IBM

developerWorks 中国 技术主题 Linux 文档库

基于 linux 平台的 libpcap 源代码分析

libpcap 是 unix/linux 平台下的网络数据包捕获函数包，大多数网络监控软件都以它为基础。Libpcap 可以在绝大多数类 unix 平台下工作，本文分析了 libpcap 在 linux 下的源代码实现，其中重点是 linux 的底层包捕获机制和过滤器设置方式，同时也简要的讨论了 libpcap 使用的包过滤机制 BPF。

施聪，成都人，高级程序员、网络设计师。从事基于 UNIX/LINUX 下的 c/c++ 程序设计和数据库建模工作已 10 年。可通过 javer@163.com 或 memncmp@yahoo.com.cn 和他联系。

2005 年 5 月 01 日

网络监控

绝大多数的现代操作系统都提供了对底层网络数据包捕获的机制，在捕获机制之上可以建立网络监控（Network Monitoring）应用软件。网络监控也常简称为sniffer,其最初的目的在于对网络通信情况进行监控，以对网络的一些异常情况进行调试处理。但随着互连网的快速普及和网络攻击行为的频繁出现，保护网络的运行安全也成为监控软件的另一个重要目的。例如，网络监控在路由器，防火墙、入侵检查等方面使用也很广泛。除此而外，它也是一种比较有效的黑客手段，例如，美国政府安全部门的"肉食动物"计划。

包捕获机制

从广义的角度上看，一个包捕获机制包含三个主要部分：最底层是针对特定操作系统的包捕获机制，最高层是针对用户程序的接口，第三部分是包过滤机制。

不同的操作系统实现的底层包捕获机制可能是不一样的，但从形式上看大同小异。数据包常规的传输路径依次为网卡、设备驱动层、数据链路层、IP 层、传输层、最后到达应用程序。而包捕获机制是在数据链路层增加一个旁路处理，对发送和接收到的数据包做过滤/缓冲等相关处理，最后直接传递到应用程序。值得注意的是，包捕获机制并不影响操作系统对数据包的网络栈处理。对用户程序而言，包捕获机制提供了一个统一的接口，使用户程序只需要简单的调用若干函数就能获得所期望的数据包。这样一来，针对特定操作系统的捕获机制对用户透明，使用户程序有比较好的可移植性。包过滤机制是对所捕获到的数据包根据用户的要求进行筛选，最终只把满足过滤条件的数据包传递给用户程序。

Libpcap 应用程序框架

Libpcap 提供了系统独立的用户级别网络数据包捕获接口，并充分考虑到应用程序的可移植性。Libpcap 可以在绝大多数类 unix 平台下工作，参考资料 A 中是对基于 libpcap 的网络应用程序的一个详细列表。在 windows 平台下，一个与libpcap 很类似的函数包 winpcap 提供捕获功能，其官方网站是http://winpcap.polito.it/。

Libpcap 软件包可从 http://www.tcpdump.org/ 下载，然后依此执行下列三条命令即可安装，但如果希望 libpcap 能在 linux 上正常工作，则必须使内核支持"packet"协议，也即在编译内核时打开配置选项 CONFIG_PACKET(选项缺省为打开)。

```
./configure
./make
./make install
```

libpcap 源代码由 20 多个 C 文件构成，但在 Linux 系统下并不是所有文件都用到。可以通过查看命令 make 的输出了解实际所用的文件。本文所针对的libpcap 版本号为 0.8.3，网络类型为常规以太网。Libpcap 应用程序从形式上看很简单，下面是一个简单的程序框架：

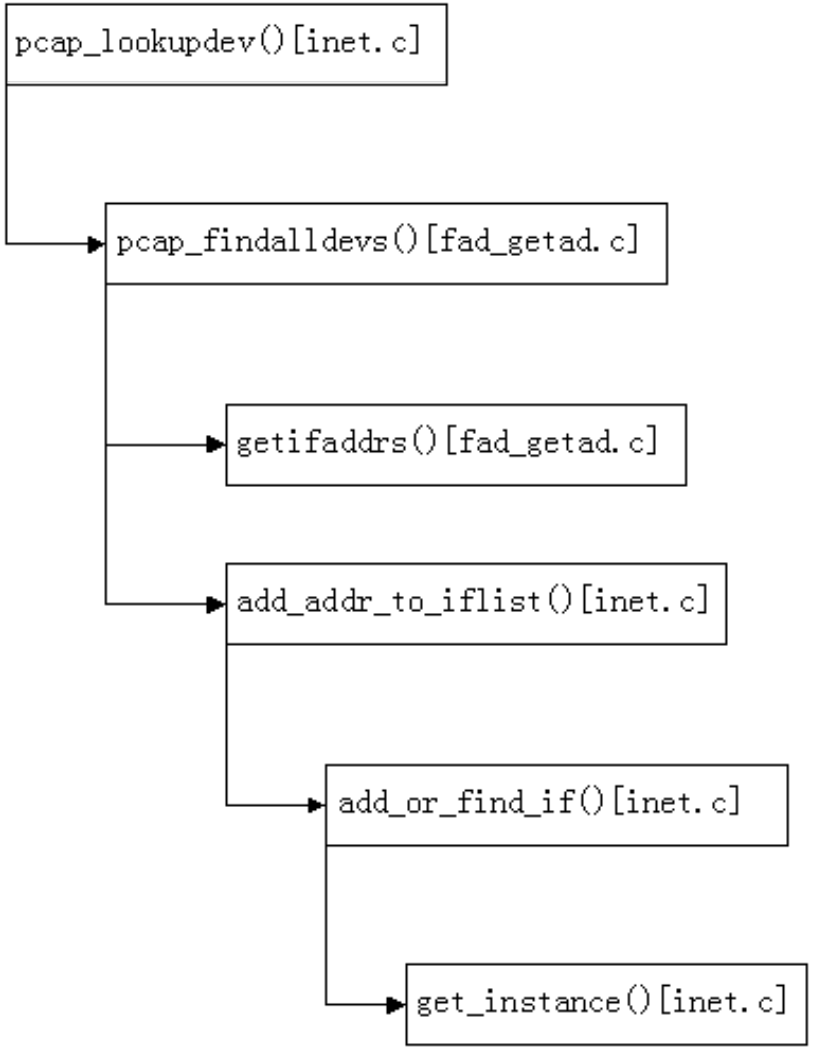
```
char * device; /* 用来捕获数据包的网络接口的名称 */
pcap_t * p; /* 捕获数据包句柄，最重要的数据结构 */
struct bpf_program fcode; /* BPF 过滤代码结构 */
/* 第一步：查找可以捕获数据包的设备 */
device = pcap_lookupdev(errbuf);
/* 第二步：创建捕获句柄，准备进行捕获 */
p = pcap_open_live(device, 8000, 1, 500, errbuf);
/* 第三步：如果用户设置了过滤条件，则编译和安装过滤代码 */
pcap_compile(p, &fcode, filter_string, 0, netmask);
pcap_setfilter(p, &fcode);
/* 第四步：进入（死）循环，反复捕获数据包 */
for( ; ; )
{
while((ptr = (char *)(pcap_next(p, &hdr))) == NULL);

/* 第五步：对捕获的数据进行类型转换，转化成以太数据包类型 */
eth = (struct libnet_ethernet_hdr *)ptr;
/* 第六步：对以太头部进行分析，判断所包含的数据包类型，做进一步的处理 */
if(eth->ether_type == ntohs(ETHERTYPE_IP))
……
if(eth->ether_type == ntohs(ETHERTYPE_ARP))
……
}

/* 最后一步：关闭捕获句柄,一个简单技巧是在程序初始化时增加信号处理函数,
以便在程序退出前执行本条代码 */
pcap_close(p);
```

检查网络设备

libpcap 程序的第一步通常是在系统中找到合适的网络接口设备。网络接口在Linux 网络体系中是一个很重要的概念，它是对具体网络硬件设备的一个抽象，在它的下面是具体的网卡驱动程序，而其上则是网络协议层。Linux 中最常见的接口设备名 eth0 和 lo。Lo 称为回路设备，是一种逻辑意义上的设备,其主要目的是为了调试网络程序之间的通讯功能。eth0 对应了实际的物理网卡，在真实网络环境下，数据包的发送和接收

都要通过 eht0。如果计算机有多个网卡，则还可以有更多的网络接口，如 eth1,eth2 等等。调用命令 ifconfig 可以列出当前所有活跃的接口及相关信息，注意对 eth0 的描述中既有物理网卡的 MAC 地址，也有网络协议的 IP 地址。查看文件 /proc/net/dev 也可获得接口信息。

Libpcap 中检查网络设备中主要使用到的函数关系如下图：



libpcap 调用 pcap_lookupdev() 函数获得可用网络接口的设备名。首先利用函数 getifaddrs() 获得所有网络接口的地址，以及对应的网络掩码、广播地址、目标地址等相关信息，再利用 add_addr_to_iflist()、add_or_find_if()、get_instance() 把网络接口的信息增加到结构链表 pcap_if 中，最后从链表中提取第一个接口作为捕获设备。其中 get_instanced() 的功能是从设备名开始,找第一个是数字的字符,做为接口的实例号。网络接口的设备号越小，则排在链表的越前面，因此，通常函数最后返回的设备名为 eth0。虽然 libpcap 可以工作在回路接口上，但显然 libpcap 开发者认为捕获本机进程之间的数据包没有多大意义。在检查网络设备操作中，主要用到的数据结构和代码如下：

```
        /* libpcap 自定义的接口信息链表 [pcap.h] */
struct pcap_if
{
struct pcap_if *next;
char *name; /* 接口设备名 */
char *description; /* 接口描述 */

/*接口的 IP 地址, 地址掩码, 广播地址,目的地址 */
struct pcap_addr addresses;
bpf_u_int32 flags;      /* 接口的参数 */
};
char * pcap_lookupdev(register char * errbuf)
{
        pcap_if_t *alldevs;
        ……
                pcap_findalldevs(&alldevs, errbuf);
                ……
                strlcpy(device, alldevs->name, sizeof(device));
        }
```

打开网络设备

当设备找到后，下一步工作就是打开设备以准备捕获数据包。Libpcap 的包捕获是建立在具体的操作系统所

提供的捕获机制上，而 Linux 系统随着版本的不同，所支持的捕获机制也有所不同。

2.0 及以前的内核版本使用一个特殊的 socket 类型 SOCK_PACKET，调用形式是 socket(PF_INET, SOCK_PACKET, int protocol)，但 Linux 内核开发者明确指出这种方式已过时。Linux 在 2.2 及以后的版本中提供了一种新的协议簇 PF_PACKET 来实现捕获机制。PF_PACKET 的调用形式为 socket(PF_PACKET, int socket_type, int protocol)，其中 socket 类型可以是 SOCK_RAW 和 SOCK_DGRAM。SOCK_RAW 类型使得数据包从数据链路层取得后，不做任何修改直接传递给用户程序，而 SOCK_DRRAM 则要对数据包进行加工(cooked)，把数据包的数据链路层头部去掉，而使用一个通用结构 sockaddr_ll 来保存链路信息。

使用 2.0 版本内核捕获数据包存在多个问题：首先，SOCK_PACKET 方式使用结构 sockaddr_pkt 来保存数据链路层信息，但该结构缺乏包类型信息；其次，如果参数 MSG_TRUNC 传递给读包函数 recvmsg()、recv()、recvfrom() 等，则函数返回的数据包长度是实际读到的包数据长度，而不是数据包真正的长度。Libpcap 的开发者在源代码中明确建议不使用 2.0 版本进行捕获。

相对 2.0 版本 SOCK_PACKET 方式，2.2 版本的 PF_PACKET 方式则不存在上述两个问题。在实际应用中，用户程序显然希望直接得到"原始"的数据包，因此使用 SOCK_RAW 类型最好。但在下面两种情况下，libpcap 不得不使用 SOCK_DGRAM 类型，从而也必须为数据包合成一个"伪"链路层头部（sockaddr_ll）。

> 某些类型的设备数据链路层头部不可用：例如 Linux 内核的 PPP 协议实现代码对 PPP 数据包头部的支持不可靠。
>
> 在捕获设备为"any"时：所有设备意味着 libpcap 对所有接口进行捕获，为了使包过滤机制能在所有类型的数据包上正常工作,要求所有的数据包有相同的数据链路头部。

打开网络设备的主函数是 pcap_open_live()[pcap-linux.c]，其任务就是通过给定的接口设备名，获得一个捕

获句柄：结构 pcap_t。pcap_t 是大多数 libpcap 函数都要用到的参数，其中最重要的属性则是上面讨论到的三种 socket 方式中的某一种。首先我们看看 pcap_t 的具体构成。

```
struct pcap [pcap-int.h]
{
        int fd; /* 文件描述字，实际就是 socket */

                /* 在 socket 上，可以使用 select() 和 poll() 等 I/O 复用类型函数 */
        int selectable_fd;
        int snapshot; /* 用户期望的捕获数据包最大长度 */
        int linktype; /* 设备类型 */
        int tzoff;              /* 时区位置，实际上没有被使用 */
        int offset;     /* 边界对齐偏移量 */
        int break_loop; /* 强制从读数据包循环中跳出的标志 */
        struct pcap_sf sf; /* 数据包保存到文件的相关配置数据结构 */
        struct pcap_md md; /* 具体描述如下 */

        int bufsize; /* 读缓冲区的长度 */
        u_char buffer; /* 读缓冲区指针 */
        u_char *bp;
        int cc;
        u_char *pkt;
        /* 相关抽象操作的函数指针，最终指向特定操作系统的处理函数 */
        int     (*read_op)(pcap_t *, int cnt, pcap_handler, u_char *);
        int     (*setfilter_op)(pcap_t *, struct bpf_program *);
        int     (*set_datalink_op)(pcap_t *, int);
        int     (*getnonblock_op)(pcap_t *, char *);
        int     (*setnonblock_op)(pcap_t *, int, char *);
        int     (*stats_op)(pcap_t *, struct pcap_stat *);
        void (*close_op)(pcap_t *);
        /*如果 BPF 过滤代码不能在内核中执行,则将其保存并在用户空间执行 */
        struct bpf_program fcode;
        /* 函数调用出错信息缓冲区 */
        char errbuf[PCAP_ERRBUF_SIZE + 1];

        /* 当前设备支持的、可更改的数据链路类型的个数 */
        int dlt_count;
        /* 可更改的数据链路类型号链表，在 linux 下没有使用 */
        int *dlt_list;
        /* 数据包自定义头部，对数据包捕获时间、捕获长度、真实长度进行描述 [pcap.h] */
        struct pcap_pkthdr pcap_header;
};
/* 包含了捕获句柄的接口、状态、过滤信息  [pcap-int.h] */
struct pcap_md {
/* 捕获状态结构  [pcap.h] */
```

```
struct pcap_stat stat;
        int use_bpf; /* 如果为1，则代表使用内核过滤*/
        u_long  TotPkts;
        u_long  TotAccepted; /* 被接收数据包数目 */
        u_long  TotDrops;       /* 被丢弃数据包数目 */
        long    TotMissed;      /* 在过滤进行时被接口丢弃的数据包数目 */
        long    OrigMissed; /*在过滤进行前被接口丢弃的数据包数目*/
#ifdef linux
        int     sock_packet; /* 如果为 1，则代表使用 2.0 内核的 SOCK_PACKET 模式 */
        int     timeout;        /* pcap_open_live() 函数超时返回时间*/
        int     clear_promisc; /* 关闭时设置接口为非混杂模式 */
        int     cooked;         /* 使用 SOCK_DGRAM 类型 */
        int     lo_ifindex;     /* 回路设备索引号 */
        char *device;   /* 接口设备名称 */

/* 以混杂模式打开 SOCK_PACKET 类型 socket 的 pcap_t 链表*/
struct pcap *next;
#endif
};
```

函数 pcap_open_live() 的调用形式是 pcap_t * pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *ebuf)，其中如果 device 为 NULL 或"any"，则对所有接口捕获，snaplen 代表用户期望的捕获数据包最大长度，promisc 代表设置接口为混杂模式（捕获所有到达接口的数据包，但只有在设备给定的情况下有意义），to_ms 代表函数超时返回的时间。本函数的代码比较简单，其执行步骤如下：

为结构 pcap_t 分配空间并根据函数入参对其部分属性进行初试化。

分别利用函数 live_open_new() 或 live_open_old() 尝试创建 PF_PACKET 方式或 SOCK_PACKET 方式的 socket，注意函数名中一个为"new"，另一个为"old"。

根据 socket 的方式，设置捕获句柄的读缓冲区长度，并分配空间。

为捕获句柄 pcap_t 设置 linux 系统下的特定函数，其中最重要的是读数据包函数和设置过滤器函数。（注意到这种从抽象模式到具体模式的设计思想在 linux 源代码中也多次出现，如 VFS 文件系统）

handle->read_op = pcap_read_linux； handle->setfilter_op = pcap_setfilter_linux；

下面我们依次分析 2.2 和 2.0 内核版本下的 socket 创建函数。

```
static int
live_open_new(pcap_t *handle, const char *device, int promisc,
   int to_ms, char *ebuf)
{
/* 如果设备给定,则打开一个 RAW 类型的套接字,否则,打开 DGRAM 类型的套接字 */
sock_fd = device ?
                        socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))
                      : socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL));
/* 取得回路设备接口的索引 */
handle->md.lo_ifindex = iface_get_id(sock_fd, "lo", ebuf);
/* 如果设备给定，但接口类型未知或是某些必须工作在加工模式下的特定类型，则使用加工模式 */
if (device) {
/* 取得接口的硬件类型 */
arptype = iface_get_arptype(sock_fd, device, ebuf);
/* linux 使用 ARPHRD_xxx 标识接口的硬件类型，而 libpcap 使用DLT_xxx
来标识。本函数是对上述二者的做映射变换，设置句柄的链路层类型为
DLT_xxx，并设置句柄的偏移量为合适的值，使其与链路层头部之和为 4 的倍数，目的是边界对齐 */
map_arphrd_to_dlt(handle, arptype, 1);
/* 如果接口是前面谈到的不支持链路层头部的类型，则退而求其次，使用 SOCK_DGRAM 模式 */
if (handle->linktype == xxx)
{
close(sock_fd);
sock_fd = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL));
}
/* 获得给定的设备名的索引 */
device_id = iface_get_id(sock_fd, device, ebuf);

/* 把套接字和给定的设备绑定，意味着只从给定的设备上捕获数据包 */
iface_bind(sock_fd, device_id, ebuf);
} else { /* 现在是加工模式 */
handle->md.cooked = 1;
/* 数据包链路层头部为结构 sockaddr_ll， SLL 大概是结构名称的简写形式 */
handle->linktype = DLT_LINUX_SLL;
                        device_id = -1;
                }

/* 设置给定设备为混杂模式 */
if (device && promisc)
{
memset(&mr, 0, sizeof(mr));
mr.mr_ifindex = device_id;
```

```
mr.mr_type = PACKET_MR_PROMISC;
setsockopt(sock_fd, SOL_PACKET, PACKET_ADD_MEMBERSHIP,
&mr, sizeof(mr));
}
/* 最后把创建的 socket 保存在句柄 pcap_t 中 */
handle->fd = sock_fd;
        }
/* 2.0 内核下函数要简单的多，因为只有唯一的一种 socket 方式 */
static int
live_open_old(pcap_t *handle, const char *device, int promisc,
              int to_ms, char *ebuf)
{
/* 首先创建一个SOCK_PACKET类型的 socket */
handle->fd = socket(PF_INET, SOCK_PACKET, htons(ETH_P_ALL));

/* 2.0 内核下，不支持捕获所有接口，设备必须给定 */
if (!device) {
strncpy(ebuf, "pcap_open_live: The \"any\" device isn't supported
         on 2.0[.x]-kernel systems", PCAP_ERRBUF_SIZE);
break;
}

/* 把 socket 和给定的设备绑定 */
iface_bind_old(handle->fd, device, ebuf);

/*以下的处理和 2.2 版本下的相似，有所区别的是如果接口链路层类型未知，则 libpcap 直接退出 */

arptype = iface_get_arptype(handle->fd, device, ebuf);
map_arphrd_to_dlt(handle, arptype, 0);
if (handle->linktype == -1) {
snprintf(ebuf, PCAP_ERRBUF_SIZE, "unknown arptype %d", arptype);
break;
}
/* 设置给定设备为混杂模式 */
if (promisc) {
memset(&ifr, 0, sizeof(ifr));
strncpy(ifr.ifr_name, device, sizeof(ifr.ifr_name));
ioctl(handle->fd, SIOCGIFFLAGS, &ifr);
ifr.ifr_flags |= IFF_PROMISC;
ioctl(handle->fd, SIOCSIFFLAGS, &ifr);
}
}
```

比较上面两个函数的代码，还有两个细节上的区别。首先是 socket 与接口绑定所使用的结构：老式的绑定使用了结构 sockaddr，而新式的则使用了 2.2 内核中定义的通用链路头部层结构 sockaddr_ll。

```
iface_bind_old(int fd, const char *device, char *ebuf)
{
struct sockaddr saddr;
memset(&saddr, 0, sizeof(saddr));
strncpy(saddr.sa_data, device, sizeof(saddr.sa_data));
bind(fd, &saddr, sizeof(saddr));
}
iface_bind(int fd, int ifindex, char *ebuf)
{
struct sockaddr_ll      sll;
memset(&sll, 0, sizeof(sll));
sll.sll_family = AF_PACKET;
sll.sll_ifindex = ifindex;
sll.sll_protocol        = htons(ETH_P_ALL);
bind(fd, (struct sockaddr *) &sll, sizeof(sll);
}
```

第二个是在 2.2 版本中设置设备为混杂模式时，使用了函数 setsockopt()，以及新的标志 PACKET_ADD_MEMBERSHIP 和结构 packet_mreq。我估计这种方式主要是希望提供一个统一的调用接口，以代替传统的（混乱的）ioctl 调用。

```
struct packet_mreq
{
int             mr_ifindex;    /* 接口索引号 */
unsigned short  mr_type;       /* 要执行的操作(号) */
unsigned short  mr_alen;       /* 地址长度 */
unsigned char   mr_address[8]; /* 物理层地址 */
};
```

用户应用程序接口

Libpcap 提供的用户程序接口比较简单，通过反复调用函数pcap_next()[pcap.c] 则可获得捕获到的数据包。下面是一些使用到的数据结构：

```
/* 单个数据包结构，包含数据包元信息和数据信息 */
struct singleton [pcap.c]
{
struct pcap_pkthdr hdr; /* libpcap 自定义数据包头部 */
const u_char * pkt; /* 指向捕获到的网络数据 */
};
```

```
/* 自定义头部在把数据包保存到文件中也被使用 */
struct pcap_pkthdr
{
                struct timeval ts; /* 捕获时间戳 */
                bpf_u_int32 caplen; /* 捕获到数据包的长度 */
                bpf_u_int32 len; /* 数据包的真正长度 */
}
/* 函数 pcap_next() 实际上是对函数 pcap_dispatch()[pcap.c] 的一个包装 */
const u_char * pcap_next(pcap_t *p, struct pcap_pkthdr *h)
{
struct singleton s;
s.hdr = h;
/*入参"1"代表收到1个数据包就返回；回调函数 pcap_oneshot() 是对结构 singleton 的属性赋值 */
if (pcap_dispatch(p, 1, pcap_oneshot, (u_char*)&s) <= 0)
return (0);
return (s.pkt); /* 返回数据包缓冲区的指针 */
}
```

pcap_dispatch() 简单的调用捕获句柄 pcap_t 中定义的特定操作系统的读数据函数：return p->read_op(p, cnt, callback, user)。在 linux 系统下，对应的读函数为 pcap_read_linux() (在创建捕获句柄时已定义 [pcap-linux.c])，而pcap_read_linux() 则是直接调用 pcap_read_packet()([pcap-linux.c])。

pcap_read_packet() 的中心任务是利用了 recvfrom() 从已创建的 socket 上读数据包数据，但是考虑到 socket 可能为前面讨论到的三种方式中的某一种，因此对数据缓冲区的结构有相应的处理，主要表现在加工模式下对伪链路层头部的合成。具体代码分析如下：

```
static int
pcap_read_packet(pcap_t *handle, pcap_handler callback, u_char *userdata)
{
/* 数据包缓冲区指针 */
u_char * bp;
/* bp 与捕获句柄 pcap_t 中 handle->buffer
之间的偏移量，其目的是为在加工模式捕获情况下，为合成的伪数据链路层头部留出空间 */
int offset;
/* PACKET_SOCKET 方式下，recvfrom() 返回 scokaddr_ll 类型，而在SOCK_PACKET 方式下，
返回 sockaddr 类型 */
#ifdef HAVE_PF_PACKET_SOCKETS
                        struct sockaddr_ll      from;
                        struct sll_header       * hdrp;
#else
                        struct sockaddr         from;
```

```
#endif
socklen_t               fromlen;
int                     packet_len, caplen;
/* libpcap 自定义的头部 */
struct pcap_pkthdr      pcap_header;
#ifdef HAVE_PF_PACKET_SOCKETS
/* 如果是加工模式，则为合成的链路层头部留出空间 */
if (handle->md.cooked)
offset = SLL_HDR_LEN;
/* 其它两中方式下，链路层头部不做修改的被返回，不需要留空间 */
else
offset = 0;
#else
offset = 0;
#endif
bp = handle->buffer + handle->offset;

/* 从内核中接收一个数据包，注意函数入参中对 bp 的位置进行修正 */
packet_len = recvfrom( handle->fd, bp + offset,
handle->bufsize - offset, MSG_TRUNC,
(struct sockaddr *) &from, &fromlen);

#ifdef HAVE_PF_PACKET_SOCKETS

/* 如果是回路设备,则只捕获接收的数据包，而拒绝发送的数据包。显然，我们只能在 PF_PACKET
方式下这样做,因为 SOCK_PACKET 方式下返回的链路层地址类型为
sockaddr_pkt，缺少了判断数据包类型的信息。*/
if (!handle->md.sock_packet &&
from.sll_ifindex == handle->md.lo_ifindex &&
from.sll_pkttype == PACKET_OUTGOING)
return 0;
#endif
#ifdef HAVE_PF_PACKET_SOCKETS
/* 如果是加工模式，则合成伪链路层头部 */
if (handle->md.cooked) {
/* 首先修正捕包数据的长度，加上链路层头部的长度 */
packet_len += SLL_HDR_LEN;
                hdrp = (struct sll_header *)bp;

/* 以下的代码分别对伪链路层头部的数据赋值 */
hdrp->sll_pkttype = xxx;
hdrp->sll_hatype = htons(from.sll_hatype);
hdrp->sll_halen = htons(from.sll_halen);
memcpy(hdrp->sll_addr, from.sll_addr,
(from.sll_halen > SLL_ADDRLEN) ?
SLL_ADDRLEN : from.sll_halen);
hdrp->sll_protocol = from.sll_protocol;
}
```

```
#endif

/* 修正捕获的数据包的长度，根据前面的讨论，SOCK_PACKET 方式下长度可能是不准确的 */
caplen = packet_len;
if (caplen > handle->snapshot)
caplen = handle->snapshot;
/* 如果没有使用内核级的包过滤,则在用户空间进行过滤*/
if (!handle->md.use_bpf && handle->fcode.bf_insns) {
if (bpf_filter(handle->fcode.bf_insns, bp,
packet_len, caplen) == 0)
{
/* 没有通过过滤，数据包被丢弃 */
return 0;
}
}
/* 填充 libpcap 自定义数据包头部数据：捕获时间,捕获的长度,真实的长度 */
ioctl(handle->fd, SIOCGSTAMP, &pcap_header.ts);
pcap_header.caplen      = caplen;
pcap_header.len         = packet_len;

/* 累加捕获数据包数目，注意到在不同内核/捕获方式情况下数目可能不准确 */
handle->md.stat.ps_recv++;
/* 调用用户定义的回调函数 */
callback(userdata, &pcap_header, bp);
}
```

数据包过滤机制

大量的网络监控程序目的不同，期望的数据包类型也不同，但绝大多数情况都都只需要所有数据包的一（小）部分。例如：对邮件系统进行监控可能只需要端口号为 25（smtp）和 110（pop3) 的 TCP 数据包，对 DNS 系统进行监控就只需要端口号为 53 的 UDP 数据包。包过滤机制的引入就是为了解决上述问题，用户程序只需简单的设置一系列过滤条件，最终便能获得满足条件的数据包。包过滤操作可以在用户空间执行，也可以在内核空间执行，但必须注意到数据包从内核空间拷贝到用户空间的开销很大，所以如果能在内核空间进行过滤，会极大的提高捕获的效率。内核过滤的优势在低速网络下表现不明显，但在高速网络下是非常突出的。在理论研究和实际应用中，包捕获和包过滤从语意上并没有严格的区分，关键在于认识到捕获数据包必然有过滤操作。基本上可以认为，包过滤机制在包捕获机制中占中心地位。

包过滤机制实际上是针对数据包的布尔值操作函数，如果函数最终返回 true，则通过过滤，反之则被丢弃。形式上包过滤由一个或多个谓词判断的并操作（AND）和或操作（OR）构成，每一个谓词判断基本上对应了数据包的协议类型或某个特定值,例如：只需要 TCP 类型且端口为 110 的数据包或 ARP 类型的数据包。包过滤机制在具体的实现上与数据包的协议类型并无多少关系，它只是把数据包简单的看成一个字节数组，而谓词判断会根据具体的协议映射到数组特定位置的值。如判断ARP类型数据包，只需要判断数组中第 13、14 个字节（以太头中的数据包类型）是否为 0X0806。从理论研究的意思上看，包过滤机制是一个数学问题，或者说是一个算法问题，其中心任务是如何使用最少的判断操作、最少的时间完成过滤处理，提高过滤效率。

BPF

Libpcap 重点使用 BPF（BSD Packet Filter）包过滤机制，BPF 于 1992 年被设计出来，其设计目的主要是解决当时已存在的过滤机制效率低下的问题。BPF的工作步骤如下：当一个数据包到达网络接口时，数据链路层的驱动会把它向系统的协议栈传送。但如果 BPF 监听接口，驱动首先调用 BPF。BPF 首先进行过滤操作，然后把数据包存放在过滤器相关的缓冲区中，最后设备驱动再次获得控制。注意到BPF是先对数据包过滤再缓冲，避免了类似 sun 的 NIT 过滤机制先缓冲每个数据包直到用户读数据时再过滤所造成的效率问题。参考资料D是关于 BPF 设计思想最重要的文献。

BPF 的设计思想和当时的计算机硬件的发展有很大联系，相对老式的过滤方式CSPF（CMU/Stanford Packet Filter）它有两大特点。1：基于寄存器的过滤机制，而不是早期内存堆栈过滤机制，2：直接使用独立的、非共享的内存缓冲区。同时，BPF 在过滤算法是也有很大进步，它使用无环控制流图（CFG control flow graph）,而不是老式的布尔表达式树（boolean expression tree）。布尔表达式树理解上比较直观，它的每一个叶子节点即是一个谓词判断，而非叶子节点则为 AND 操作或 OR操作。CSPF 有三个主要的缺

点。1：过滤操作使用的栈在内存中被模拟，维护栈指针需要使用若干的加/减等操作，而内存操作是现代计算机架构的主要瓶颈。2：布尔表达式树造成了不需要的重复计算。3：不能分析数据包的变长头部。BPF 使用的CFG 算法实际上是一种特殊的状态机，每一节点代表了一个谓词判断，而左右边分别对应了判断失败和成功后的跳转，跳转后又是谓词判断，这样反复操作，直到到达成功或失败的终点。CFG 算法的优点在于把对数据包的分析信息直接建立在图中，从而不需要重复计算。直观的看，CFG 是一种"快速的、一直向前"的算法。

过滤代码的编译

BPF 对 CFG 算法的代码实现非常复杂，它使用伪机器方式。BPF 伪机器是一个轻量级的，高效的状态机，对 BPF 过滤代码进行解释处理。BPF 过滤代码形式为"opcode jt jf k"，分别代表了操作码和寻址方式、判断正确的跳转、判断失败的跳转、操作使用的通用数据域。BPF 过滤代码从逻辑上看很类似于汇编语言，但它实际上是机器语言，注意到上述 4 个域的数据类型都是 int 和 char 型。显然，由用户来写过滤代码太过复杂，因此 libpcap 允许用户书写高层的、容易理解的过滤字符串，然后将其编译为BPF代码。

Libpcap 使用了 4 个源程序 gencode.c、optimize.c、grammar.c、scanner.c完成编译操作，其中前两个实现了对过滤字符串的编译和优化，后两个主要是为编译提供从协议相关过滤条件到协议无关(的字符数组)位置信息的映射，并且它们由词汇分析器生成器 flex 和 bison 生成。参考资料 C 有对此两个工具的讲解。

```
flex -Ppcap_ -t scanner.l > $$.scanner.c; mv $$.scanner.c scanner.c
bison -y -p pcap_ -d grammar.y
mv y.tab.c grammar.c
mv y.tab.h tokdefs.h
```

编译过滤字符串调用了函数 pcap_compile()[getcode.c]，形式为：

```
int pcap_compile(pcap_t *p, struct bpf_program *program,
             char *buf, int optimize, bpf_u_int32 mask)
```

其中 buf 指向用户过滤字符串，编译后的 BPF 代码存在在结构 bpf_program中，标志 optimize 指示是否对 BPF 代码进行优化。

```
/* [pcap-bpf.h] */
struct bpf_program {
u_int bf_len; /* BPF 代码中谓词判断指令的数目 */
struct bpf_insn *bf_insns; /* 第一个谓词判断指令 */
};

/* 谓词判断指令结构，含意在前面已描述 [pcap-bpf.h] */
struct bpf_insn {
u_short code;
u_char  jt;
u_char  jf;
bpf_int32 k;
};
```

过滤代码的安装

前面我们曾经提到，在内核空间过滤数据包对整个捕获机制的效率是至关重要的。早期使用 SOCK_PACKET 方式的 Linux 不支持内核过滤，因此过滤操作只能在用户空间执行（请参阅函数 pcap_read_packet() 代码）,在《UNIX 网络编程(第一卷)》（参考资料 B）的第 26 章中对此有明确的描述。不过现在看起来情况已经发生改变，linux 在 PF_PACKET 类型的 socket 上支持内核过滤。Linux 内核允许我们把一个名为 LPF(Linux Packet Filter) 的过滤器直接放到 PF_PACKET 类型 socket 的处理过程中，过滤器在网卡接收中断执行后立即执行。LSF 基于 BPF 机制，但两者在实现上有略微的不同。实际代码如下：

```
/* 在包捕获设备上附加 BPF 代码 [pcap-linux.c]*/
static int
pcap_setfilter_linux(pcap_t *handle, struct bpf_program *filter)
{
#ifdef SO_ATTACH_FILTER
struct sock_fprog       fcode;
int can_filter_in_kernel;
int err = 0;
#endif
```

```
/* 检查句柄和过滤器结构的正确性 */
if (!handle)
return -1;
if (!filter) {
strncpy(handle->errbuf, "setfilter: No filter specified",
sizeof(handle->errbuf));
return -1;
}
/* 具体描述如下 */
if (install_bpf_program(handle, filter) < 0)
return -1;
/* 缺省情况下在用户空间运行过滤器,但如果在内核安装成功,则值为 1 */
handle->md.use_bpf = 0;

/* 尝试在内核安装过滤器 */
#ifdef SO_ATTACH_FILTER
#ifdef USHRT_MAX
if (handle->fcode.bf_len > USHRT_MAX) {
/*过滤器代码太长，内核不支持 */
fprintf(stderr, "Warning: Filter too complex for kernel\n");
fcode.filter = NULL;
can_filter_in_kernel = 0;
} else
#endif /* USHRT_MAX */
{
/* linux 内核设置过滤器时使用的数据结构是 sock_fprog，/
/* 而不是 BPF 的结构 bpf_program ,因此应做结构之间的转换 */
switch (fix_program(handle, &fcode)) {

/* 严重错误，直接退出 */
case -1:
default:
return -1;

/* 通过检查，但不能工作在内核中 */
case 0:
can_filter_in_kernel = 0;
break;
/* BPF 可以在内核中工作 */
case 1:
can_filter_in_kernel = 1;
break;
}
}
/* 如果可以在内核中过滤，则安装过滤器到内核中 */
if (can_filter_in_kernel) {
if ((err = set_kernel_filter(handle, &fcode)) == 0)
{
```

```
/* 安装成功 !!! */
handle->md.use_bpf = 1;
}
else if (err == -1)     /* 出现非致命性错误 */
{
if (errno != ENOPROTOOPT && errno != EOPNOTSUPP) {
fprintf(stderr, "Warning: Kernel filter failed:
 %s\n",pcap_strerror(errno));
}
}
}
/* 如果不能在内核中使用过滤器，则去掉曾经可能在此 socket
上安装的内核过滤器。主要目的是为了避免存在的过滤器对数据包过滤的干扰 */
if (!handle->md.use_bpf)
reset_kernel_filter(handle);[pcap-linux.c]
#endif
}
/* 把 BPF 代码拷贝到 pcap_t 数据结构的 fcode 上 */
int install_bpf_program(pcap_t *p, struct bpf_program *fp)
{
size_t prog_size;
/* 首先释放可能已存在的 BPF 代码 */
pcap_freecode(&p->fcode);
/* 计算过滤代码的长度，分配内存空间 */
prog_size = sizeof(*fp->bf_insns) * fp->bf_len;
p->fcode.bf_len = fp->bf_len;
p->fcode.bf_insns = (struct bpf_insn *)malloc(prog_size);
if (p->fcode.bf_insns == NULL) {
snprintf(p->errbuf, sizeof(p->errbuf),
"malloc: %s", pcap_strerror(errno));
return (-1);
}
/* 把过滤代码保存在捕获句柄中 */
memcpy(p->fcode.bf_insns, fp->bf_insns, prog_size);

return (0);
}
/* 在内核中安装过滤器 */
static int set_kernel_filter(pcap_t *handle, struct sock_fprog *fcode)
{
int total_filter_on = 0;
int save_mode;
int ret;
int save_errno;
/*在设置过滤器前，socket 的数据包接收队列中可能已存在若干数据包。当设置过滤器后，
这些数据包极有可能不满足过滤条件，但它们不被过滤器丢弃。这意味着，
传递到用户空间的头几个数据包不满足过滤条件。注意到在用户空间过滤这不是问题，
因为用户空间的过滤器是在包进入队列后执行的。Libpcap
```

```
解决这个问题的方法是在设置过滤器之前，首先读完接收队列中所有的数据包。
具体步骤如下。*/

/*为了避免无限循环的情况发生（反复的读数据包并丢弃，但新的数据包不停的到达），*/
/*首先设置一个过滤器，阻止所有的包进入 */

setsockopt(handle->fd, SOL_SOCKET, SO_ATTACH_FILTER,
&total_fcode, sizeof(total_fcode);
/* 保存 socket 当前的属性 */
save_mode = fcntl(handle->fd, F_GETFL, 0);
/* 设置 socket 它为非阻塞模式 */
fcntl(handle->fd, F_SETFL, save_mode | O_NONBLOCK);
/* 反复读队列中的数据包，直到没有数据包可读。这意味着接收队列已被清空 */
while (recv(handle->fd, &drain, sizeof drain, MSG_TRUNC) >= 0);

/* 恢复曾保存的 socket 属性 */
fcntl(handle->fd, F_SETFL, save_mode);

/* 现在安装新的过滤器 */
setsockopt(handle->fd, SOL_SOCKET, SO_ATTACH_FILTER,
fcode, sizeof(*fcode));
}
/* 释放 socket 上可能有的内核过滤器 */
static int reset_kernel_filter(pcap_t *handle)
{
int dummy;
return setsockopt(handle->fd, SOL_SOCKET, SO_DETACH_FILTER,
&dummy, sizeof(dummy));
}
```

linux 在安装和卸载过滤器时都使用了函数 setsockopt()，其中标志SOL_SOCKET 代表了对 socket 进行设置，而 SO_ATTACH_FILTER 和 SO_DETACH_FILTER 则分别对应了安装和卸载。下面是 linux 2.4.29 版本中的相关代码：

```
[net/core/sock.c]
#ifdef CONFIG_FILTER
case SO_ATTACH_FILTER:
……
/* 把过滤条件结构从用户空间拷贝到内核空间 */
if (copy_from_user(&fprog, optval, sizeof(fprog)))
break;
/* 在 socket 上安装过滤器 */
ret = sk_attach_filter(&fprog, sk);
                        ……
```

```
case SO_DETACH_FILTER:
/* 使用自旋锁锁住 socket */
spin_lock_bh(&sk->lock.slock);
filter = sk->filter;
/* 如果在 socket 上有过滤器，则简单设置为空，并释放过滤器内存 */
if (filter) {
sk->filter = NULL;
spin_unlock_bh(&sk->lock.slock);
sk_filter_release(sk, filter);
break;
}
spin_unlock_bh(&sk->lock.slock);
ret = -ENONET;
break;
#endif
```

上面出现的 sk_attach_filter() 定义在 net/core/filter.c，它把结构sock_fprog 转换为结构 sk_filter, 最后把此结构设置为 socket 的过滤器：sk->filter = fp。

其他代码

libpcap 还提供了其它若干函数，但基本上是提供辅助或扩展功能，重要性相对弱一点。我个人认为，函数 pcap_dump_open() 和 pcap_open_offline() 可能比较有用，使用它们能把在线的数据包写入文件并事后进行分析处理。

总结

1994 年 libpcap 的第一个版本被发布，到现在已有 11 年的历史，如今libpcap 被广泛的应用在各种网络监控软件中。Libpcap 最主要的优点在于平台无关性，用户程序几乎不需做任何改动就可移植到其它 unix 平台上；其次，libpcap也能适应各种过滤机制，特别对BPF的支持最好。分析它的源代码，可以学习开发者优秀的设计思想和实现技巧，也能了解到（linux）操作系统的网络内核实现，对个人能力的提高有很大帮助。

参考资料

A：《Libpcap, winpcap, libdnet, and libnet applications and resources》

B：《UNIX网络编程(第一卷)》 W.Richard Stevens

C：《使用 lex 和 yacc 编译代码》 Peter Seebach

D：《The BSD Packet Filter: A New Architecture for User-level Packet Capture 》 Steven McCanne and Van Jacobson

E：linux 联机帮助手册：socket(2)、socket(7)、packet等

F：《A compiler for Packet Filters》