jasenwan88的专栏

Linux下Libpcap源码分析和包过滤机制（2）

分类： 网络基础知识 2012-07-19 13:41 213人阅读 评论(0) 收藏 举报

linux socket struct linux内核 header 数据结构

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。

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。

文章搜索

文章分类

linux内核 (25)
编辑工具 (2)
移植 (1)
网络基础知识 (16)
linux基础 (4)
服务器类 (4)
unix高级环境编程 (2)
项目需求 (1)
luci (1)

文章存档

2014年11月 (1)
2012年07月 (29)
2012年06月 (22)
2012年04月 (1)
2012年03月 (2)
展开

阅读排行

emacs 命令小结---开关、 (8295)

评论排行

推荐文章

```
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,
```

* struts2国际化
* 无需超级用户mpi多机执行
* 从视图索引说Notes数据库（下）
* Java虚拟机解析篇之---垃圾回收器
* 2015互联网校招总结—一路走来
* SSL 3.0曝出Poodle漏洞的解决方案-----开发者篇

```
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 */
};
```

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);
}
```

上一篇 Linux下Libpcap源码分析和包过滤机制（1）

下一篇 Linux下Libpcap源码分析和包过滤机制（3）

主题推荐 源码 linux 操作系统 数据结构 应用程序

猜你在找

应用层HTTP数据包的截获与还原技术的实现
Linux下Libpcap源码分析和包过滤机制（1）
Linux操作系统下如何编译安装源码包软件
深入分析Linux内核源码-Linux管道的实现机制
linux tty core 源码分析（2）
Linux下Libpcap源码分析和包过滤机制（3）
Linux下Libpcap源码分析和包过滤机制（1）
Linux下Libpcap源码分析和包过滤机制
Linux下Libpcap源码分析和包过滤机制
Linux内核源码系列（二）：探究内核基础层数据结构，

查看评论

暂无评论

您还没有登录,请[登录]或[注册]