第14章 附录

12．1 WinPcap 使用入门

本节介绍了 WinPcap 的使用入门，翻译至 WinPcap 用户手册。首先，在应用 WinPcap 之前，我们应该知道，如何使用 WinPcap。

12．1．1 获取网卡列表

用 PCAP 写应用程序的第一件事往往就是要获得本地的网卡列表。PCAP 提供了 pcap_findalldevs()这个函数来实现此功能，这个 API 返回一个 pcap_if 结构的连表，连表的每项内容都含有全面的网卡信息：尤其是字段名字和含有名字的描述以及有关驱动器的易读信息。

得到网络驱动列表的程序如下：

```
#include "pcap.h"

main()
{
    pcap_if_t *alldevs;
    pcap_if_t *d;
    int i=0;
    char errbuf[PCAP_ERRBUF_SIZE];

    /* 这个 API 用来获得网卡 的列表 */
    if (pcap_findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
```

```
        exit(1);
    }

    /*  显示列表的响应字段的内容  */
    for(d=alldevs;d;d=d->next)
    {
        printf("%d. %s", ++i, d->name);
        if (d->description)
            printf(" (%s)\n", d->description);
        else
            printf(" (No description available)\n");
    }

    if(i==0)
    {
        printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
        return;
    }

    /* We don't need any more the device list. Free it */
    pcap_freealldevs(alldevs);
}
```

程序说明：

首先 pcap_findalldevs()同其他的 libpca 函数一样有一个 errbuf 参数，当有异常情况发生时，这个参数会被 PCAP 填充为某个特定的错误字串。

再次，UNIX 也同样提供 pcap_findalldevs()这个函数，但是请注意并非所有的系统都支持 libpcap 提供的网络程序接口。所以我门要想写出合适的程序就必须考虑到这些情况（系统不能够返回一些字段的描述信息），在这种情况下我门

应该给出类似"No description available"这样的提示。

最后结束时别忘了用 pcap_freealldevs()释放掉内存资源。

12．1．2 获得已安装网络驱动器的高级信息

在前一节中演示了如何获得已存在适配器的静态信息。实际上 WinPcap 同样也提供其他的高级信息，特别是 pcap_findalldevs()这个函数返回的每个 pcap_if 结构体都同样包含一个 pcap_addr 结构的列表，他包含：
一个地址列表，一个掩码列表，一个广播地址列表和一个目的地址列表。

下面的例子通过一个 ifprint()函数打印出了 pcap_if 结构的的所有字段信息，该程序对每一个 pcap_findalldevs()所返回的 pcap_if 结构循环调用 ifprint()来显示详细的字段信息。

```
#include "pcap.h"
#ifndef WIN32
#include <sys/socket.h>
#include <netinet/in.h>
#else
#include <winsock.h>
#endif

void ifprint(pcap_if_t *d);
char *iptos(u_long in);

int main()
{
   pcap_if_t *alldevs;
   pcap_if_t *d;
   char errbuf[PCAP_ERRBUF_SIZE+1];
```

```
    /* 获得网卡的列表 */
    if (pcap_findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr,"Error in pcap_findalldevs: %s\n",errbuf);
        exit(1);
    }

    /* 循环调用 ifprint() 来显示 pcap_if 结构的信息*/
    for(d=alldevs;d;d=d->next)
    {
        ifprint(d);
    }

    return 1;
}

/* Print all the available information on the given interface */
void ifprint(pcap_if_t *d)
{
    pcap_addr_t *a;

    /* Name */
    printf("%s\n",d->name);

    /* Description */
    if (d->description)
        printf("\tDescription: %s\n",d->description);

    /* Loopback Address*/
```

```
    printf("\tLoopback: %s\n",(d->flags & PCAP_IF_LOOPBACK)?"yes":"no");

    /* IP addresses */
    for(a=d->addresses;a;a=a->next) {
        printf("\tAddress Family: #%d\n",a->addr->sa_family);

    /*关于 sockaddr_in 结构请参考其他的网络编程书*/
        switch(a->addr->sa_family)
        {
            case AF_INET:
                printf("\tAddress Family Name: AF_INET\n");//打印网络地址类型
                if (a->addr)//打印 IP 地址
                    printf("\tAddress: %s\n",iptos(((struct sockaddr_in
*)a->addr)->sin_addr.s_addr));
                if (a->netmask)//打印掩码
                    printf("\tNetmask: %s\n",iptos(((struct sockaddr_in
*)a->netmask)->sin_addr.s_addr));
                if (a->broadaddr)//打印广播地址
                    printf("\tBroadcast Address: %s\n",iptos(((struct sockaddr_in
*)a->broadaddr)->sin_addr.s_addr));
                if (a->dstaddr)//目的地址
                    printf("\tDestination Address: %s\n",iptos(((struct sockaddr_in
*)a->dstaddr)->sin_addr.s_addr));
                break;
            default:
                printf("\tAddress Family Name: Unknown\n");
                break;
        }
    }
    printf("\n");
```

```
}

/*    将一个 unsigned long  型的 IP 转换为字符串类型的 IP */
#define IPTOSBUFFERS        12
char *iptos(u_long in)
{
      static char output[IPTOSBUFFERS][3*4+3+1];
      static short which;
      u_char *p;

      p = (u_char *)&in;
      which = (which + 1 == IPTOSBUFFERS ? 0 : which + 1);
      sprintf(output[which], "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
      return output[which];
}
```

12．1．3 打开网卡捕获数据包

现在我门已经知道了如何去获得网卡的信息现在就让我们开始真正的工作：打开网卡并捕获数据流。在这一节里我们将写一个打印流经网络的每个数据包信息的程序。打开网卡的功能是通过 pcap_open_live()来实现的它有三个参数 snaplen promisc to_ms。

snaplen 用于指定所捕获包的特定部分，在一些系统上（象 xBSD and Win32 等）驱动只给出所捕获数据包的一部分而不是全部，这样就减少了拷贝数据的数量从而提高了包捕获的效率。

promisc 指明网卡处于混杂模式，在正常情况下网卡只接受去往它的包而去往其他主机的数据包则被忽略。相反当网卡处于混杂 模式时他将接收所有的流经它的数据包：这就意味着在共享介质的情况下我门可以捕获到其它主机的数据包。大部分的包捕获程序都将混杂模式设为默认，所有我们在下面的例子里也将网卡设为混杂模式。

to_ms 参数指定读数据的超时控制，超时以毫秒计算。当在超时时间内网卡上没有数据到来时对网卡的读操作将返回（如 pcap_dispatch() or pcap_next_ex() 等函数）。还有，如果网卡处于统计模式下（请查看“统计和收集网络数据流一节”）to_ms 还定义了统计的时间间隔。如果该参数为 0 那么意味着没有超时控制，对网卡的读操作在没有数据到来是将永远堵塞。如果为-1 那么对网卡的读操作将立即返回不管有没有数据可读。

```
#include "pcap.h"

/* prototype of the packet handler */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data);

main()
{
    pcap_if_t *alldevs;
    pcap_if_t *d;
    int inum;
    int i=0;
    pcap_t *adhandle;
    char errbuf[PCAP_ERRBUF_SIZE];

    /* 获得网卡的列表 */
    if (pcap_findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
        exit(1);
    }
```

```
/* 打印网卡信息 */
for(d=alldevs; d; d=d->next)
{
    printf("%d. %s", ++i, d->name);
    if (d->description)
        printf(" (%s)\n", d->description);
    else
        printf(" (No description available)\n");
}

if(i==0)
{
    printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
    return -1;
}

printf("Enter the interface number (1-%d):",i);
scanf("%d", &inum);                                    //输入要选择打开的网卡号

if(inum < 1 || inum > i)    //判断号的合法性
{
    printf("\nInterface number out of range.\n");
    /* Free the device list */
    pcap_freealldevs(alldevs);
    return -1;
}

/* 找到要选择的网卡结构 */
for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);
```

```
    /* 打开选择的网卡 */
    if ( (adhandle= pcap_open_live(d->name, // 设备名称
                                   65536,     // portion of the packet to capture.
                                   // 65536 grants that the whole packet will be
captured on all the

MACs.
                                   1,         // 混杂模式
                                   1000,      // 读超时为 1 秒
                                   errbuf     // error buffer
                                   ) ) == NULL)
    {
        fprintf(stderr,"\nUnable to open the adapter. %s is not supported by
WinPcap\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf("\nlistening on %s...\n", d->description);

    /* At this point, we don't need any more the device list. Free it */
    pcap_freealldevs(alldevs);

    /* 开始捕获包 */
    pcap_loop(adhandle, 0, packet_handler, NULL);

    return 0;
}
```

```
/* 对每一个到来的数据包调用该函数 */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data)
{
    struct tm *ltime;
    char timestr[16];

    /* 将时间戳转变为易读的标准格式*/
    ltime=localtime(&header->ts.tv_sec);
    strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);

    printf("%s,%.6d len:%d\n", timestr, header->ts.tv_usec, header->len);

}
```

一旦网卡被打开，旧可以调用 pcap_dispatch() 或 pcap_loop()进行数据的捕获，这两个函数的功能十分相似不同的是 pcap_ dispatch()可以不被阻塞，而 pcap_loop()在没有数据流到达时将阻塞。在这个简单的例子里用 pcap_loop()就足够了，而在一些复杂的程序里往往用 pcap_dispatch()。

这两个函数都有返回的参数，一个指向某个函数（该函数用来接受数据如该程序中的 packet_handler）的指针，libpcap 调用该函数对每个从网上到来的数据包进行处理和接收数据包。另一个参数是带有时间戳和包长等信息的头部，最后一个是含有所有协议头部数据报的实际数据。注意 MAC 的冗余校验码一般不出现，因为当一个桢到达并被确认后网卡就把它删除了，同样需要注意的是大多数网卡会丢掉冗余码出错的数据包，所以 WinPcap 一般不能够捕获这些出错的数据报。

上例中显示了从 pcap_pkthdr 中提取出了每个数据报的时间戳和长度并在显

示器上打印出了他们。

12．1．4 不用 loopback 捕获数据报

这节的例子很象先前的一节（获得网卡的高级信息）但是这一节中是用 pcap_next_ex()来代替 pcap_loop()来捕获数据包。基于回调包捕获机制的 pcap_loop()在某些情况下是不错的选择。但是在一些情况下处理回调并不特别好：这会使程序变的复杂并且在象多线程或 C++类这些情况下它看起来到象一块绊脚石。在这些情况下 pcap_next_ex()允许直接调用来接收包，它的参数和 pcap_loop()相同：有一个网卡描述副，和两个指针，这两个指针会被初始化并返回给用户，一个是 pcap_pkthdr 结构，另一个是接收数据的缓冲区。

下面的程序我门将循环调用前一节的例子中的回掉部分，只是把它移到了 main 里面了。

```
#include "pcap.h"

main()
{
    pcap_if_t *alldevs;
    pcap_if_t *d;
    int inum;
    int i=0;
    pcap_t *adhandle;
    int res;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct tm *ltime;
    char timestr[16];
    struct pcap_pkthdr *header;
    u_char *pkt_data;
```

```
/* Retrieve the device list */
if (pcap_findalldevs(&alldevs, errbuf) == -1)
{
    fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
    exit(1);
}

/* Print the list */
for(d=alldevs; d; d=d->next)
{
    printf("%d. %s", ++i, d->name);
    if (d->description)
        printf(" (%s)\n", d->description);
    else
        printf(" (No description available)\n");
}

if(i==0)
{
    printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
    return -1;
}

printf("Enter the interface number (1-%d):",i);
scanf("%d", &inum);

if(inum < 1 || inum > i)
{
    printf("\nInterface number out of range.\n");
```

```
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* Jump to the selected adapter */
    for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);

    /* Open the adapter */
    if ( (adhandle= pcap_open_live(d->name, // name of the device
        65536,          // portion of the packet to capture.
        // 65536 grants that the whole packet will be captured on all the MACs.
        1,              // promiscuous mode
        1000,           // read timeout
        errbuf          // error buffer
        ) ) == NULL)
    {
        fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf("\nlistening on %s...\n", d->description);

    /* At this point, we don't need any more the device list. Free it */
    pcap_freealldevs(alldevs);
```

```
    /*  此处循环调用  pcap_next_ex 来接受数据报*/
    while((res = pcap_next_ex( adhandle, &header, &pkt_data)) >= 0){

        if(res == 0)
            /* Timeout elapsed */
            continue;

        /* convert the timestamp to readable format */
        ltime=localtime(&header->ts.tv_sec);
        strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);

        printf("%s,%.6d len:%d\n", timestr, header->ts.tv_usec, header->len);
    }

    if(res == -1){
        printf("Error reading the packets: %s\n", pcap_geterr(adhandle));
        return -1;
    }

    return 0;
}
```

注：pcap_next_ex()只在 Win32 环境下才行因为它不是原始 libpcap API 中的一部分，这就意味着依赖于这个函数的代码不会在 UNIX 下工作。那么为什么我们用 pcap_next_ex()而不用 pcap_next()？因为 pcap_next()有许多限制在很多情况下并不鼓励用它。首先它的效率很低因为它隐藏了回掉方法并且还依赖于 pcap_dispatch()这个函数。再次它不能够识别文件结束标志 EOF 所以对来自文件的数据流它几乎无能为力。

注意当 pcap_next_ex()在成功，超时，出错和文件结束的情况下会返回不同的值。

12．1．5 数据流的过滤

WinPcap 或 libpca 最强大的特点之一就是数据流的过滤引擎。它提供一种高效的方法来只捕获网络数据流的某些数据而且常常和系统的捕获机制相集成。过滤数据的函数是 pcap_compile() 和 pcap_setfilter()来实现的。

pcap_compile()用来编译一个过滤设备，它通过一个高层的 boolean 型变量和字串产生一系列的能够被底层驱动所解释的二进制编码。boolean 表示语法能够在这个文件的过滤表示语法中找到。

pcap_setfilter() 用来联系一个在内核驱动上过滤的过滤器，这时所有网络数据包都将流经过滤器，并拷贝到应用程序中。

下面的代码展示了如何编译并社定一个过滤设备。注意我们必须从 pcap_if 结构中获得掩码，因为一些过滤器的创建需要这个参数。

下面的代码段中的 pcap_compile()的"ip and tcp"参数说明只有 IPV4 和 TCP 数据才会被内核保存并被传递到应用程序。

```
if(d->addresses != NULL)
    /* 获得第一个接口地址的掩码 */
    netmask=((struct sockaddr_in
*)(d->addresses->netmask))->sin_addr.S_un.S_addr;
else
    /* 如果这个接口没有地址那么我们假设他为 C 类地址 */
    netmask=0xffffff;


//compile the filter
if(pcap_compile(adhandle, &fcode, "ip and tcp", 1, netmask) <0 ){
```

```
        fprintf(stderr,"\nUnable to compile the packet filter. Check the syntax.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    //set the filter
    if(pcap_setfilter(adhandle, &fcode)<0){
        fprintf(stderr,"\nError setting the filter.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }
```

如何你想进一步查看本节中用过滤器过滤数据流的例子可以查看下一节《数据的解包》。

12．1．6 解析数据包

现在经过上几节的学习能够进行数据报的捕获和过滤了，我们想用一个简单的"real world"程序将我们所学的知识应用于实际。

这一节里我们将利用以前的代码并将其引申从而建立一个更实用的程序。该程序的主要目的是如何显示出所捕获的数据报的内容，尤其是对它的协议头的分析和说明。这个程序名叫 UDPdump 它将在屏幕上显示出我们网络上 UDP 数据的信息。

在此我们选择解析 UDP 而不用 TCP 因为他比 TCP 简单更加的直观明了。下面让我们来看看原代码。

```
#include "pcap.h"
```

```
/* 4 BIT 的 IP 头定义 */
typedef struct ip_address{
    u_char byte1;
    u_char byte2;
    u_char byte3;
    u_char byte4;
}ip_address;

/* IPv4 头的定义 */
typedef struct ip_header{
    u_char  ver_ihl;        // 4 bit 的版本信息 + 4 bits 的头长
    u_char  tos;            // TOS 类型
    u_short tlen;           // 总长度
    u_short identification; // Identification
    u_short flags_fo;       // Flags (3 bits) + Fragment offset (13 bits)
    u_char  ttl;            // 生存期
    u_char  proto;          // 后面的协议信息
    u_short crc;            // 校验和
    ip_address  saddr;      // 源 IP
    ip_address  daddr;      // 目的 IP
    u_int   op_pad;         // Option + Padding
}ip_header;

/* UDP header*/
typedef struct udp_header{
    u_short sport;          // Source port
    u_short dport;          // Destination port
    u_short len;            // Datagram length
    u_short crc;            // Checksum
}udp_header;
```

```
/* 定义处理包的函数 */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data);


main()
{
    pcap_if_t *alldevs;
    pcap_if_t *d;
    int inum;
    int i=0;
    pcap_t *adhandle;
    char errbuf[PCAP_ERRBUF_SIZE];
    u_int netmask;
    char packet_filter[] = "ip and udp";
    struct bpf_program fcode;

    /* Retrieve the device list */
    if (pcap_findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
        exit(1);
    }

    /* Print the list */
    for(d=alldevs; d; d=d->next)
    {
        printf("%d. %s", ++i, d->name);
        if (d->description)
```

```
            printf(" (%s)\n", d->description);
        else
            printf(" (No description available)\n");
    }

    if(i==0)
    {
        printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
        return -1;
    }

    printf("Enter the interface number (1-%d):",i);
    scanf("%d", &inum);

    if(inum < 1 || inum > i)
    {
        printf("\nInterface number out of range.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* Jump to the selected adapter */
    for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);

    /* Open the adapter */
    if ( (adhandle= pcap_open_live(d->name, // name of the device
                                   65536,     // portion of the packet to capture.
                                              // 65536 grants that the whole
packet will be captured on
```

all the MACs.

```
                                    1,                  // promiscuous mode
                                    1000,               // read timeout
                                    errbuf              // error buffer
                                    ) ) == NULL)
    {
        fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* Check the link layer. We support only Ethernet for simplicity. */
    if(pcap_datalink(adhandle) != DLT_EN10MB)
    {
        fprintf(stderr,"\nThis program works only on Ethernet networks.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    if(d->addresses != NULL)
        /* Retrieve the mask of the first address of the interface */
        netmask=((struct sockaddr_in *)(d->addresses->netmask))->sin_addr.S_un.S_addr;
    else
        /* If the interface is without addresses we suppose to be in a C class network */
```

```
        netmask=0xffffff;


    //compile the filter
    if(pcap_compile(adhandle, &fcode, packet_filter, 1, netmask) <0 ){
        fprintf(stderr,"\nUnable to compile the packet filter. Check the syntax.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    //set the filter
    if(pcap_setfilter(adhandle, &fcode)<0){
        fprintf(stderr,"\nError setting the filter.\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf("\nlistening on %s...\n", d->description);

    /* At this point, we don't need any more the device list. Free it */
    pcap_freealldevs(alldevs);

    /* start the capture */
    pcap_loop(adhandle, 0, packet_handler, NULL);

    return 0;
}
```

```
/* Callback function invoked by libpcap for every incoming packet */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char
*pkt_data)
{
    struct tm *ltime;
    char timestr[16];
    ip_header *ih;
    udp_header *uh;
    u_int ip_len;
    u_short sport,dport;

    /* convert the timestamp to readable format */
    ltime=localtime(&header->ts.tv_sec);
    strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);

    /* print timestamp and length of the packet */
    printf("%s.%.6d len:%d ", timestr, header->ts.tv_usec, header->len);

    /*  找到 IP 头的位置  */
    ih = (ip_header *) (pkt_data +
        14); //14 为以太头的长度

    /*  找到 UDP 的位置  */
    ip_len = (ih->ver_ihl & 0xf) * 4;
    uh = (udp_header *) ((u_char*)ih + ip_len);

    /*  将端口信息从网络型转变为主机顺序  */
    sport = ntohs( uh->sport );
    dport = ntohs( uh->dport );
```

```
    /* print ip addresses and udp ports */
    printf("%d.%d.%d.%d.%d -> %d.%d.%d.%d.%d\n",
        ih->saddr.byte1,
        ih->saddr.byte2,
        ih->saddr.byte3,
        ih->saddr.byte4,
        sport,
        ih->daddr.byte1,
        ih->daddr.byte2,
        ih->daddr.byte3,
        ih->daddr.byte4,
        dport);
}
```

首先我门设置 UDP 过滤，用这种方法我们确保 packet_handler()只接受到基于 IPV4 的 UDP 数据。我们同样定义了

两个数据结构来描述 IP 和 UDP 的头部信息，packet_handler()用这两个结构来定位头部的各种字段。packet_handler()虽然只是限于处理一些 UDP 数据但却显示了复杂的嗅探器如 tcpdump/WinDump 的工作原理。

首先我们对 MAC 地址的头部并不感兴趣所以我们跳过它。不过在开始捕获之前我们用 pcap_datalink()来检查 MAC 层，所以以上的程序只能够工作在 Ethernet networks 上，再次我们确保 MAC 头为 14 bytes。MAC 头之后是 IP 头，我们从中提取出了目的地址。IP 之后是 UDP，在确定 UDP 的位置时有点复杂，因为 IP 的长度以为版本的不同而不同，所以我们用头长字段来定位 UDP，一旦我们确定了 UDP 的起始位置，我们就可以解析出原和目的端口。

下面是我们打印出来的一些结果：

```
1. {A7FD048A-5D4B-478E-B3C1-34401AC3B72F} (Xircom t 10/100 Adapter)
Enter the interface number (1-2):1
```

listening on Xircom CardBus Ethernet 10/100 Adapter...

16:13:15.312784 len:87 130.192.31.67.2682 -> 130.192.3.21.53

16:13:15.314796 len:137 130.192.3.21.53 -> 130.192.31.67.2682

16:13:15.322101 len:78 130.192.31.67.2683 -> 130.192.3.21.53

上面每一行都显示出不同的数据包的内容.

12．1．7 处理脱机的堆文件

通过以前的学习我门已经熟悉了从网卡上捕获数据包，现在我门将学习如何处理数据包。WINPCAP 为我们提供了很多 API 来将流经网络的数据包保存到一个堆文件并读取堆的内容。这一节将讲述如何使用所有的这些 API。

这种文件的格式很简单，但包含了所捕获的数据报的二进制内容，这种文件格式也是很多网络工具的标准如 WinDump, Ethereal 还有 Snort 等.

关于如何将数据包保存到文件：

首先我们看看如何以 LIBPCAP 的格式写数据包。

下面的例子演示了如何从指定的接口上捕获数据包并将它们存储到一个指定的文件。

```
#include "pcap.h"

/* 定义处理数据的函数原形 */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data);

main(int argc, char **argv)
{
    pcap_if_t *alldevs;
```

```
pcap_if_t *d;
int inum;
int i=0;
pcap_t *adhandle;//定义文件句柄
char errbuf[PCAP_ERRBUF_SIZE];
pcap_dumper_t *dumpfile;

/* 检查命令行参数 是否带有文件名*/
if(argc != 2){
	printf("usage: %s filename", argv[0]);
	return -1;
}

/* 获得驱动列表 */
if (pcap_findalldevs(&alldevs, errbuf) == -1)
{
	fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
	exit(1);
}

/* 打印 list */
for(d=alldevs; d; d=d->next)
{
	printf("%d. %s", ++i, d->name);
	if (d->description)
		printf(" (%s)\n", d->description);
	else
		printf(" (No description available)\n");
}
```

```
if(i==0)
{
    printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
    return -1;
}

printf("Enter the interface number (1-%d):",i);
scanf("%d", &inum);

if(inum < 1 || inum > i)
{
    printf("\nInterface number out of range.\n");
    /* Free the device list */
    pcap_freealldevs(alldevs);
    return -1;
}

/* 跳转到指定的网卡 */
for(d=alldevs, i=0; i< inum-1 ;d=d->next, i++);

/* Open the adapter */
if ( (adhandle = pcap_open_live(d->name, // name of the device
                                65536,      // portion of the packet to capture.
                                            // 65536 grants that the whole packet will be captured on all the MACs.
                                1,          // promiscuous mode
                                1000,       // read timeout
                                errbuf      // error buffer
                                ) ) == NULL)
{
```

```
        fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n");
        /* Free the device list */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* 打开文件 */
    dumpfile = pcap_dump_open(adhandle, argv[1]);
    if(dumpfile==NULL){
        fprintf(stderr,"\nError opening output file\n");
        return -1;
    }

    printf("\nlistening on %s...\n", d->description);

    /* At this point, we don't need any more the device list. Free it */
    pcap_freealldevs(alldevs);

    /* 循环捕获数据并调用 packet_handler 函数把数据存储到堆文件 */
    pcap_loop(adhandle, 0, packet_handler, (unsigned char *)dumpfile);

    return 0;
}

/* Callback function invoked by libpcap for every incoming packet */

void packet_handler(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *pkt_data)
{
```

```
    /* 此函数功能将数据报存储到堆文件 */
    pcap_dump(dumpfile, header, pkt_data);
}
```

正如你看到的那样该程序的结构非常类似与以前的例子，区别是：

一旦打开网卡就调用 pcap_dump_open()来打开一个文件，该调用将文件和某个网卡相关联。

packet_handler()内部通过调用 pcap_dump()来将捕获的数据报存储到文件。

pcap_dump()的参数和 packet_handler()一样，所以用起来比较方便。

从文件读数据包：

下面我们来看如何从文件读取数据内容。下面的代码打开了 一个堆文件并打印了其中的每个包内容。

pcap_open_offline()用来打开一个堆文件，之后用 pcap_loop()来循环从文件中读取数据。你能发现读取脱机的数据几乎和实时的从网卡上读取一摸一样。

```
#include <stdio.h>
#include <pcap.h>

#define LINE_LEN 16

void dispatcher_handler(u_char *, const struct pcap_pkthdr *, const u_char *);

main(int argc, char **argv) {

    pcap_t *fp;
    char errbuf[PCAP_ERRBUF_SIZE];

    if(argc != 2){
```

```
        printf("usage: %s filename", argv[0]);
        return -1;

    }

    /* 打开一个存储有数据的堆文件 */
    if ( (fp = pcap_open_offline(argv[1], errbuf) ) == NULL)
    {
        fprintf(stderr,"\nError opening dump file\n");
        return -1;
    }

    // 读取数据直到遇到 EOF 标志。
    pcap_loop(fp, 0, dispatcher_handler, NULL);

    return 0;
}

void dispatcher_handler(u_char *temp1,
                        const struct pcap_pkthdr *header, const u_char
*pkt_data)
{
    u_int i=0;

    /* print pkt timestamp and pkt len */
    printf("%ld:%ld (%ld)\n", header->ts.tv_sec, header->ts.tv_usec, header->len);

    /* Print the packet */
    for (i=1; (i < header->caplen + 1 ) ; i++)
```

```
        {
            printf("%.2x ", pkt_data[i-1]);
            if ( (i % LINE_LEN) == 0) printf("\n");
        }

        printf("\n\n");

}
```

下面的代码具有一样的作用，只不过是用 pcap_next_ex()来代替 pcap_loop()循环读取数据而已。

```
#include <stdio.h>
#include <pcap.h>

#define LINE_LEN 16

main(int argc, char **argv) {

        pcap_t *fp;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct pcap_pkthdr *header;
        u_char *pkt_data;
        u_int i=0;
        int res;

        if(argc != 2){

            printf("usage: %s filename", argv[0]);
            return -1;
```

```
    }

    /* Open a capture file */
    if ( (fp = pcap_open_offline(argv[1], errbuf) ) == NULL)
    {
        fprintf(stderr,"\nError opening dump file\n");
        return -1;
    }

    /* Retrieve the packets from the file */
    while((res = pcap_next_ex( fp, &header, &pkt_data)) >= 0){
        /* print pkt timestamp and pkt len */
        printf("%ld:%ld (%ld)\n", header->ts.tv_sec, header->ts.tv_usec,
header->len);

        /* Print the packet */
        for (i=1; (i < header->caplen + 1 ) ; i++)
        {
            printf("%.2x ", pkt_data[i-1]);
            if ( (i % LINE_LEN) == 0) printf("\n");
        }

        printf("\n\n");
    }

    if(res == -1){
        printf("Error reading the packets: %s\n", pcap_geterr(fp));
    }
```

```
    return 0;
}
```

用 pcap_live_dump 将数据写到文件：

WinPcap 的最新版本提供了一个进一步的方法来将数据包存储到磁盘，就是使用 pcap_live_dump()函数。他需要三个参数：一个文件名，和一个该文件允许的最大长度还有一个参数是该文件所允许的最大包的数量。对这些参数来说 0 意味着没有最大限制。注：我们可以在调用 pcap_live_dump()前设置一个过滤器来定义哪些数据报需要存储。

pcap_live_dump() 是非阻塞的，所以他会立刻返回：数据的存储过程将会异步的进行，直到文件到达了指定的最大长度或最大数据报的数目为止。
应用程序能够用 pcap_live_dump_ended()来等检查是否数据存储完毕，如果你指定的最大长度参数和数据报数量为 0，那么该操作将永远阻塞。

pcap_live_dump() 和 pcap_dump()的不同从设置的最大极限来说就是性能的问题。pcap_live_dump()采用 WinPcap NPF 驱动来从内核级的层次上向文件中写数据，从而使内存拷贝最小化。

显然，这些特点当前在其他的操作系统下是不能够实现的，pcap_live_dump()是 WinPcap 所特有的，而且只能够应用于 Win32 环境.

12．1．8 发送数据包

尽管 WinPcap 从名字上来看表明他的主要目的是捕获数据包，但是他还为原始网络提供了一些其他的功能，其中之一就是用户可以发送数据包，这也就是本节的主要内容。需要指出的是原来的 libpcap 并不提供数据包 的发送功能，这里所说的功能都是 WinPcap 的扩展功能，所以并不能够工作在 UNIX 下。

用 pcap_sendpacket 来发送一个数据包：

下面的代码是一个最简单的发送数据的方法。打开一个适配器后就可以用pcap_sendpacket（）来手工发送一个数据包了。这个函数需要的参数：一个装有要发送数据的缓冲区，要发送的长度，和一个适配器。注意缓冲区中的数据将不被内核协议处理，只是作为最原始的数据流被发送，所以我门必须填充好正确的协议头以便正确的将数据发送。

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

#include <pcap.h>

void usage();

void main(int argc, char **argv) {
	pcap_t *fp;
	char error[PCAP_ERRBUF_SIZE];
	u_char packet[100];
	int i;

	/* Check the validity of the command line */
	if (argc != 2)
	{
		printf("usage: %s inerface", argv[0]);
		return;
	}

	/* 打开指定网卡 */
	if((fp = pcap_open_live(argv[1], 100, 1, 1000, error) ) == NULL)
	{
		fprintf(stderr,"\nError opening adapter: %s\n", error);
```

```
        return;
}

/* 假设网络环境为 ethernet，我门把目的 MAC 设为 1:1:1:1:1:1*/
packet[0]=1;
packet[1]=1;
packet[2]=1;
packet[3]=1;
packet[4]=1;
packet[5]=1;

/* 假设源 MAC 为  2:2:2:2:2:2 */
packet[6]=2;
packet[7]=2;
packet[8]=2;
packet[9]=2;
packet[10]=2;
packet[11]=2;

/* 填充发送包的剩余部分 */
for(i=12;i<100;i++){
        packet[i]=i%256;
}

/* 发送包 */
pcap_sendpacket(fp,
        packet,
        100);

return;
```

}

发送队列：

pcap_sendpacket()只是提供一个简单的直接的发送数据的方法，而发送队列提供一个高级的强大的和最优的机制来发送一组数据包，队列实际上是一个装有要发送数据的一个容器，他有一个最大值来表明他所能够 容纳的最大比特数。

pcap_sendqueue_alloc()用来创建一个队列，并指定该队列的大小。一旦队列被创建就可以调用 pcap_sendqueue_queue()来将数据存储到队列中，这个函数接受一个带有时间戳和长度的 pcap_pkthdr 结构和一个装有数据报的缓冲区。这些参数同样也应用于 pcap_next_ex() 和 pcap_handler()中，所以给要捕获的数据包或要从文件读取的数据包排队就是 pcap_sendqueue_queue()的事情了。

WinPcap 调用 pcap_sendqueue_transmit()来发送数据包，注意，第三个参数如果非零，那么发送将是同步的，这将占用很大的 CPU 资源，因为发生在内核驱动的同步发送是通过"brute force"loops 的，但是一般情况下能够精确到微秒。

需要指出的是用 pcap_sendqueue_transmit()来发送比用 pcap_sendpacket()来发送一系列的数据要高效的多，因为他的数据是在内核级上被缓冲。当不再需要队列时可以用 pcap_sendqueue_destroy()来释放掉所有的队列资源。

下面的代码演示了如何用发送队列来发送数据，该示例用 pcap_open_offline()打开了一个文件，然后将数据从文件移动到已分配的队列，这时就同步地传送队列（如果用户指定为同步的话）。

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

#include <pcap.h>

void usage();

void main(int argc, char **argv) {
```

```
pcap_t *indesc,*outdesc;
char error[PCAP_ERRBUF_SIZE];
FILE *capfile;
int caplen,
    sync;
u_int res;
pcap_send_queue *squeue;
struct pcap_pkthdr *pktheader;
u_char *pktdata;

/* Check the validity of the command line */
if (argc <= 2 || argc >= 5)
{
    usage();
    return;
}

/*  得到文件长度  */
capfile=fopen(argv[1],"rb");
if(!capfile){
    printf("Capture file not found!\n");
    return;
}

fseek(capfile , 0, SEEK_END);
caplen= ftell(capfile)- sizeof(struct pcap_file_header);
fclose(capfile);

/*  检查确保时间戳被忽略  */
if(argc == 4 && argv[3][0] == 's')
```

```
        sync = TRUE;
    else
        sync = FALSE;

    /* Open the capture */
    if((indesc = pcap_open_offline(argv[1], error)) == NULL){
        fprintf(stderr,"\nError opening the input file: %s\n", error);
        return;
    }

    /* Open the output adapter */
    if((outdesc = pcap_open_live(argv[2], 100, 1, 1000, error) ) == NULL)
    {
        fprintf(stderr,"\nError opening adapter: %s\n", error);
        return;
    }

    /*  检测 MAC 类型  */
    if(pcap_datalink(indesc) != pcap_datalink(outdesc)){
        printf("Warning: the datalink of the capture differs from the one of the
selected

interface.\n");
        printf("Press a key to continue, or CTRL+C to stop.\n");
        getchar();
    }

    /*  给对列分配空间  */
    squeue = pcap_sendqueue_alloc(caplen);
```

```
    /* 从文件获得包来填充队列 */
    while((res = pcap_next_ex( indesc, &pktheader, &pktdata)) == 1){
        if(pcap_sendqueue_queue(squeue, pktheader, pktdata) == -1){
            printf("Warning: packet buffer too small, not all the packets will be
sent.\n");
            break;
        }
    }

    if(res == -1){
        printf("Corrupted input file.\n");
        pcap_sendqueue_destroy(squeue);
        return;
    }

    /* 传送队列数据 */

    if((res = pcap_sendqueue_transmit(outdesc, squeue, sync)) < squeue->len)
    {
        printf("An error occurred sending the packets: %s. Only %d bytes were
sent\n", error,

res);
    }

    /* free the send queue */
    pcap_sendqueue_destroy(squeue);

    return;
}
```

```
void usage()
{

    printf("\nSendcap, sends a libpcap/tcpdump capture file to the net. Copyright (C)
2002 Loris

Degioanni.\n");
    printf("\nUsage:\n");
    printf("\t sendcap file_name adapter [s]\n");
    printf("\nParameters:\n");
    printf("\nfile_name: the name of the dump file that will be sent to the
network\n");
    printf("\nadapter: the device to use. Use \"WinDump -D\" for a list of valid
devices\n");
    printf("\ns: if present, forces the packets to be sent synchronously, i.e. respecting
the

timestamps in the dump file. This option will work only under Windows NTx.\n\n");

    exit(0);
}
```

12．1．9 收集并统计网络流量

这一节将展示 WinPcap 的另一高级功能：收集网络流量的统计信息。WinPcap 的统计引擎在内核层次上对到来的数据进行分类。如果你想了解更多的细节请查看 NPF 驱动指南。

为了利用这个功能来监视网络，我们的程序必须打开一个网卡并用

pcap_setmode()将其设置为统计模式。注意 pcap_setmode()要用 MODE_STAT 来将网卡设置为统计模式。

在统计模式下编写一个程序来监视 TCP 流量只是几行代码的事情，下面的例子说明了如何来实现该功能的。

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

#include <pcap.h>

void usage();

void dispatcher_handler(u_char *,
    const struct pcap_pkthdr *, const u_char *);


void main(int argc, char **argv) {
    pcap_t *fp;
    char error[PCAP_ERRBUF_SIZE];
    struct timeval st_ts;
    u_int netmask;
    struct bpf_program fcode;

    /* Check the validity of the command line */
    if (argc != 2)
    {
        usage();
        return;
    }
```

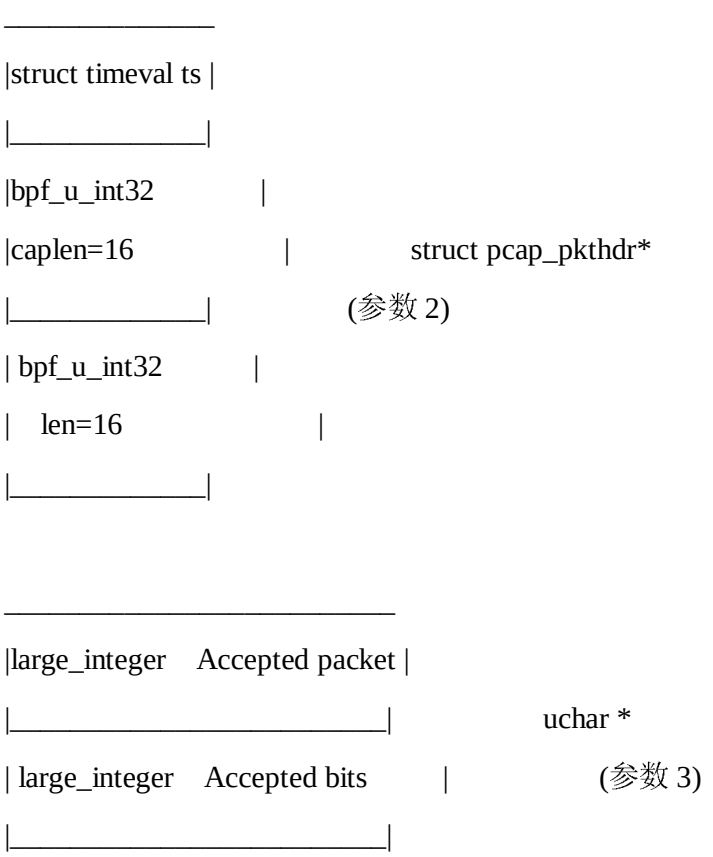
```
/* Open the output adapter */
if((fp = pcap_open_live(argv[1], 100, 1, 1000, error) ) == NULL)
{
	fprintf(stderr,"\nError opening adapter: %s\n", error);
	return;
}

/* Don't care about netmask, it won't be used for this filter */
netmask=0xffffff;

//compile the filter
if(pcap_compile(fp, &fcode, "tcp", 1, netmask) <0 ){
	fprintf(stderr,"\nUnable to compile the packet filter. Check the syntax.\n");
	/* Free the device list */
	return;
}

//set the filter
if(pcap_setfilter(fp, &fcode)<0){
	fprintf(stderr,"\nError setting the filter.\n");
	/* Free the device list */
	return;
}

/* 将网卡设置为统计模式 */
pcap_setmode(fp, MODE_STAT);

printf("TCP traffic summary:\n");

/* Start the main loop */
```

```
    pcap_loop(fp, 0, dispatcher_handler, (PUCHAR)&st_ts);

    return;
}

void dispatcher_handler(u_char *state, const struct pcap_pkthdr *header, const u_char
*pkt_data)
{
    struct timeval *old_ts = (struct timeval *)state;
    u_int delay;
    LARGE_INTEGER Bps,Pps;
    struct tm *ltime;
    char timestr[16];

    /* 从最近一次的采样以微秒计算延迟时间 */
    delay=(header->ts.tv_sec - old_ts->tv_sec) * 1000000 - old_ts->tv_usec +
header->ts.tv_usec;
    /* 获得每秒的比特数 */
    Bps.QuadPart=(((*(LONGLONG*)(pkt_data + 8)) * 8 * 1000000) / (delay));
    /*                                                    ^         ^
                                                          |         |
                                                          |         |
                                                          |         |
                                  converts bytes in bits --         |
                                                                    |
                              delay is expressed in microseconds --
    */

    /* 获得每秒的数据包数 */
    Pps.QuadPart=(((*(LONGLONG*)(pkt_data)) * 1000000) / (delay));
```

```
	/*  将时间戳转变为可读的标准格式  */
	ltime=localtime(&header->ts.tv_sec);
	strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);

	/* Print timestamp*/
	printf("%s ", timestr);

	/* Print the samples */
	printf("BPS=%I64u ", Bps.QuadPart);
	printf("PPS=%I64u\n", Pps.QuadPart);

	//store current timestamp
	old_ts->tv_sec=header->ts.tv_sec;
	old_ts->tv_usec=header->ts.tv_usec;
}


void usage()
{

	printf("\nShows the TCP traffic load, in bits per second and packets per second.\nCopyright (C) 2002 Loris Degioanni.\n");
	printf("\nUsage:\n");
	printf("\t tcptop adapter\n");
	printf("\t You can use \"WinDump -D\" if you don't know the name of your adapters.\n");

	exit(0);
}
```

在设置为统计模式前可以设置一个过滤器来指定要捕获的协议包。如果没有设置过滤器那么整个网络数据都将被监视。一旦设置了 过滤器就可以调用 pcap_setmode()来设置为统计模式，之后网卡开始工作在统计模式下。

需要指出的是 pcap_open_live()的第四个参数(to_ms)定义了采样的间隔，回调函数 pcap_loop()每隔一定间隔就获取一次采样统计，这个采样被装入 pcap_loop()的第二和第三个参数，过程如下图所示：



用两个 64 位的计数器分别记录最近一次间隔数据包数量和比特数量。

本例子中，网卡打开时设置超时为 1000 毫秒，也就是说 dispatcher_handler()每隔 1 秒就被调用一次。过滤器也设置为只监视 TCP 包，然后 pcap_setmode() and pcap_loop()被调用，注意一个指向 timeval 的指针 作为参数传送到 pcap_loop()。这个 timeval 结构将用来存储个时间戳以计算两次采样的时间间隔。

dispatcher_handler()用该间隔来获取每秒的比特数和数据包数，并把着两个数显示在显示器上。

最后指出的是目前这个例子是比任何一个利用传统方法在用户层统计的包捕获程序都高效。因为统计模式需要最小数量的数据拷贝和上下环境交换，同时还有最小的内存需求，所以 CPU 是最优的。

12．2 在 Wind2000 上搭建 Snort 入侵检测平台

12．3 Snort 版本升级历程

12．3 参考资料