

4 网络信息被动获取与处理

2019年4月24日 16:26

0 网络流监测技术—系统架构及原理

- [0.1 网络流监控模式](#)
- [0.2 旁路监测技术](#)
- [0.3 关键技术介绍](#)

1 TCP/IP协议与以太网通讯

2 基于Libpcap的网络编程技术

3 基于Libnet的网络编程技术

4 Linux网络数据流解析技术

- [4.1 libnids背景介绍](#)
- [4.2 libnids数据结构介绍](#)
- [4.3 libnids函数介绍](#)
- [4.4 libnids中TCP协议检测](#)
- [4.5 libnids开发示例](#)
- [4.6 libnids升级换代](#)

0 网络流监测技术—系统架构及原理

- [0.1 网络流监控模式](#)
- [0.2 旁路监测技术](#)
- [0.3 关键技术介绍](#)

0.1 网络流监控模式

- 串联监控模式
 - 一般是通过网关或者网桥的模式来进行监控
- 旁路监控模式
 - 一般是指通过交换机等网络设备的“端口镜像”功能来实现监控

网络流监控模式比较

- 旁路部署起来比较灵活方便，不会影响现有的网络结构，串行需要对现有网络结构进行变动
- 旁路模式对原始传递的数据包不会造成延时，不会对网速造成任何影响。而串联模式是串联在网络中的，那么所有的数据必须先经过监控系统，通过监控系统的分析检查之后，才能够发送到各个客户端，所以会对网速有一定的延时。
- 旁路监控设备一旦故障或者停止运行，不会影响现有网络的正常原因。而串联监控设备如果出现故障，会导致网络中断，导致网络单点故障。

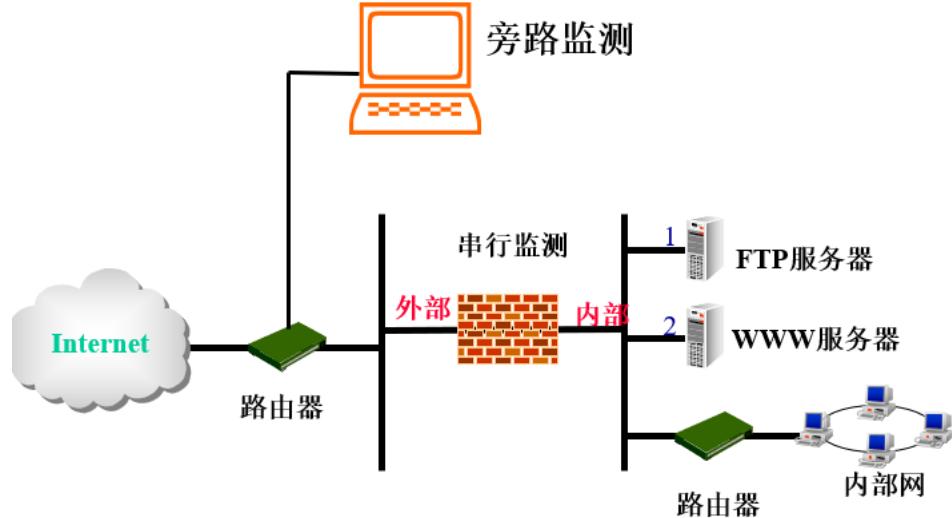
旁路监控模式局限性

- 数据获取：旁路需要交换机支持端口镜像才可以实现监控。
- 数据管控：旁路模式采用发送RST包的方式来断开TCP连接，不能禁止UDP通讯。对于UDP应用，一般还需要在路由器上面禁止UDP端口进行配合。而串联模式不存在该问题。

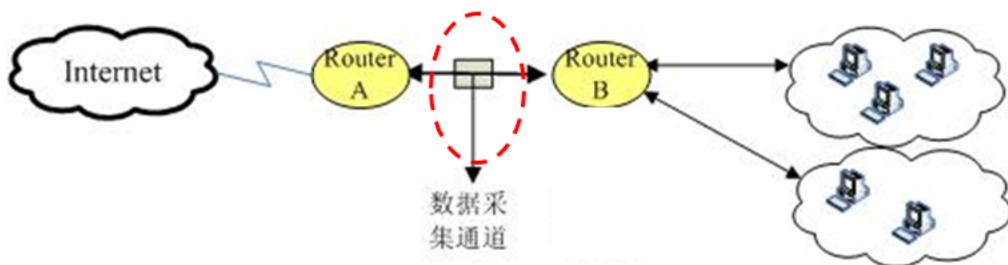
网络流监测产品

- 串行
 - NetworkFirewall
 - UTM
 - NIPS
- 旁路
 - NIDS
 - ContentFilter
 - WebFilter
 - 网络行为审计

设备部署环境比较

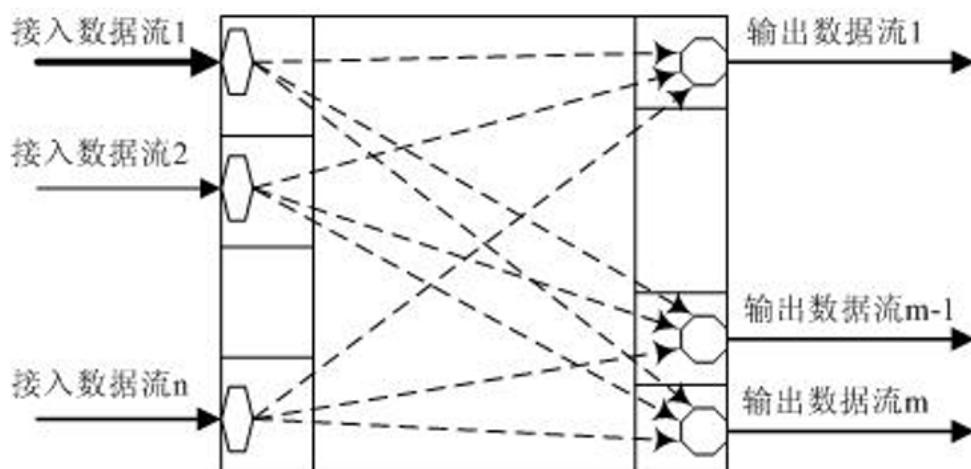


0.2 旁路监测技术 数据分流

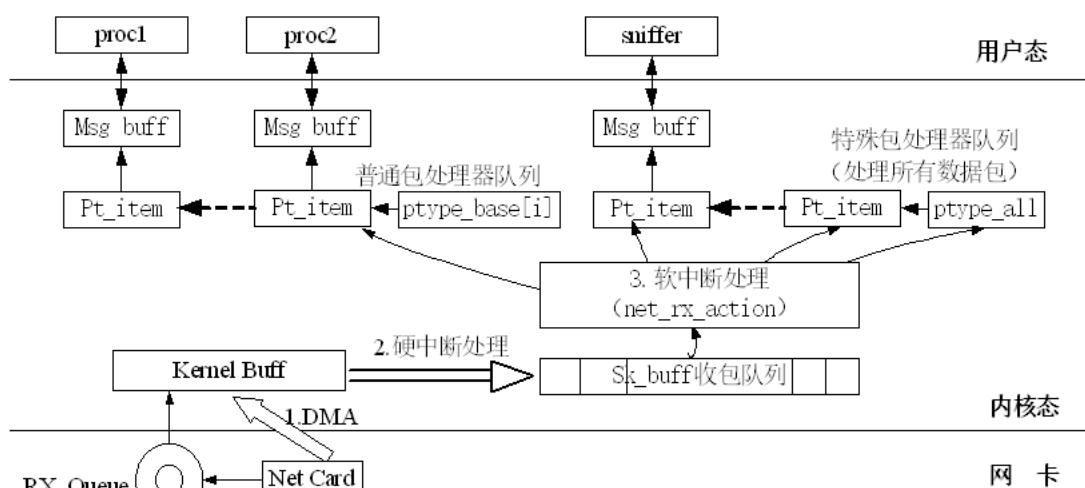


- 1. 分光器**
- 2. 路由交换**
- 3. HUB**

数据汇聚



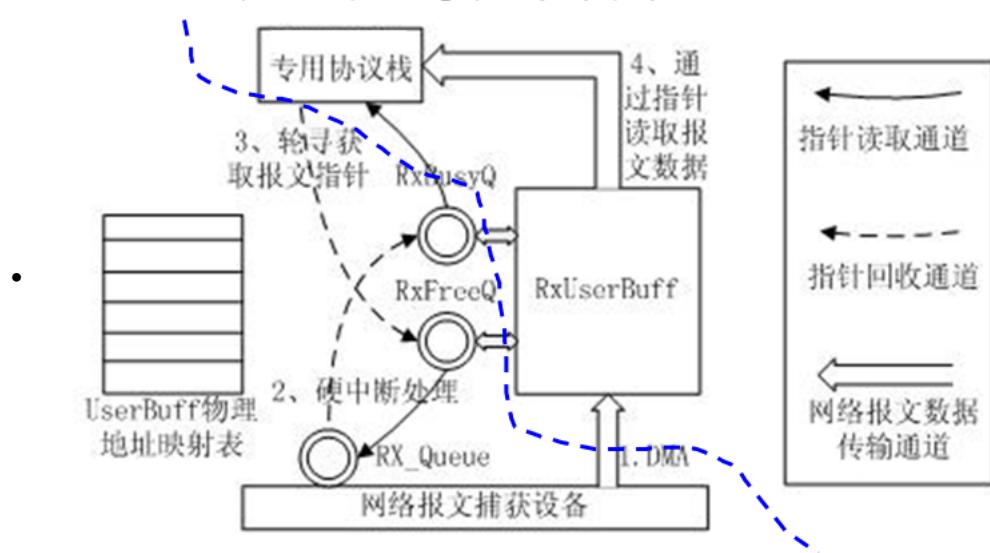
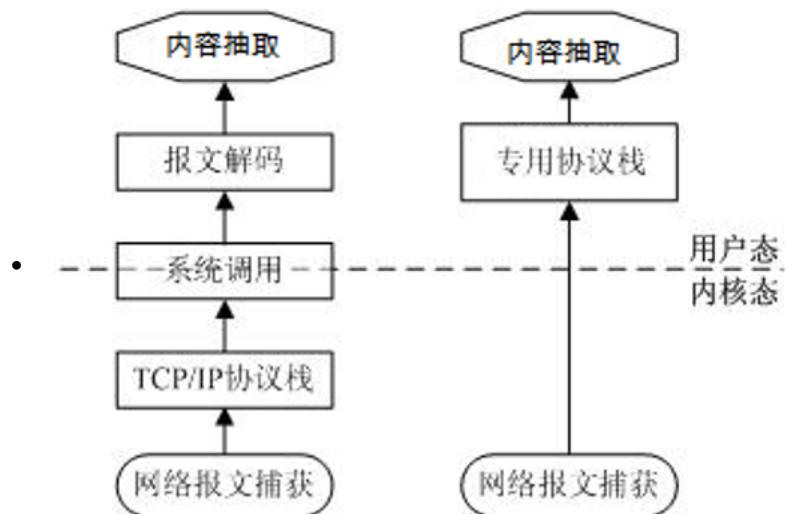
报文抓取



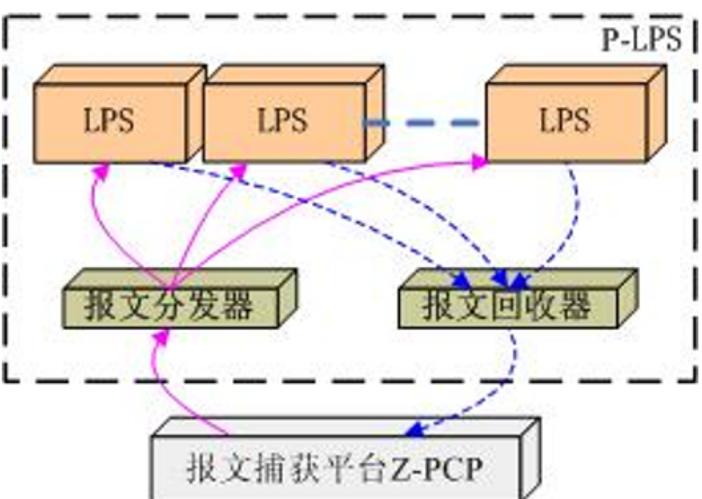
报文抓取性能影响因素

- 系统调用
- 数据拷贝
- 计算校验和
- 网卡中断

报文抓取过程优化

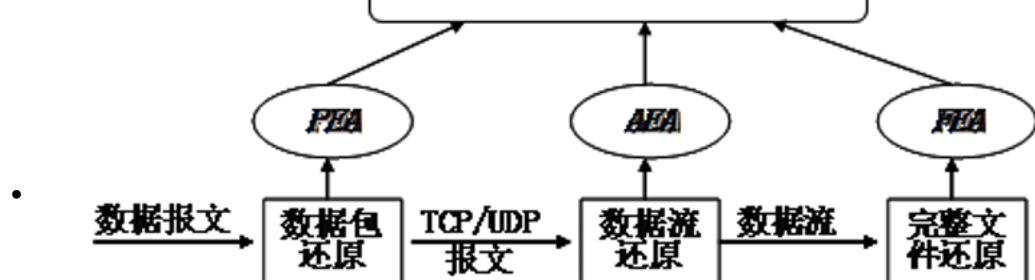


协议还原

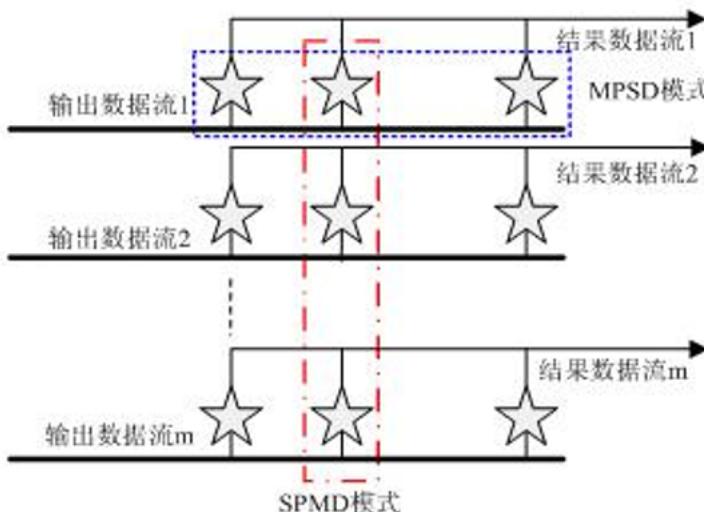


内容抽取

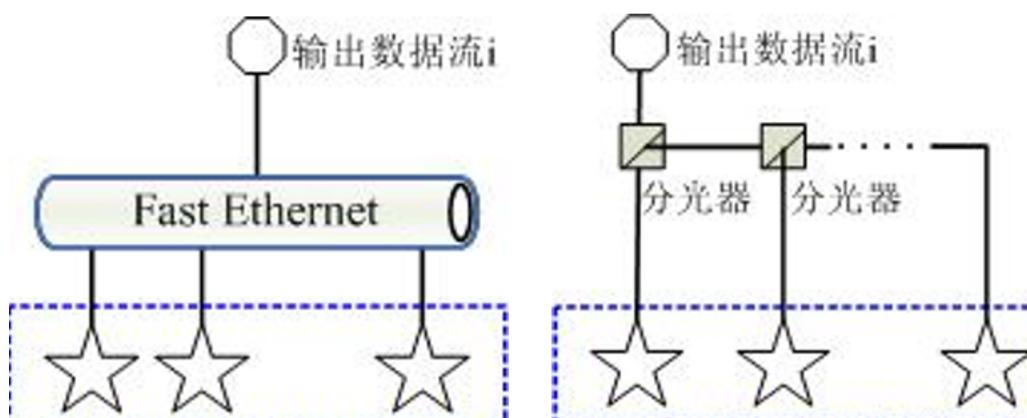
内容抽取器



系统架构设计



数据耦合方法



0.3 关键技术介绍

内容过滤平台关键技术

- Intel网卡零拷贝技术
 - Linux系统内存映射技术
 - Linux内核模块加载技术
 - 共享内存无锁访问技术
 - “写写” => “读写”
- 并行协议栈技术
 - 网络数据检测协议栈
 - 共享内存无锁访问技术
 - “写写” => “读写”
- 快速应用协议识别技术
- 高速串匹配技术

进程通信技术——UNIX

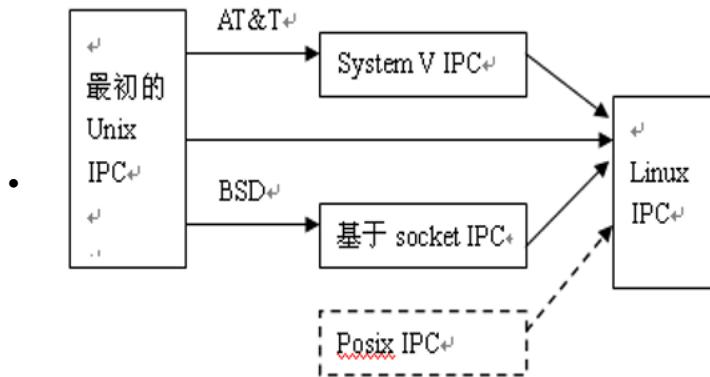
进程通信方法 (IPC)

- linux下的进程通信手段基本上是从Unix平台上的进程通信手段继承而来
- AT&T的贝尔实验室： Unix早期的进程间通信手段进行了系统的改进和扩充，形成了“systemVIPC”，通信进程局限在单个计算机内

- BSD (加州大学伯克利分校的伯克利软件发布中心) : 形成了基于套接口 (socket) 的进程间通信机制
- Unix IPC包括: 管道、FIFO、信号
- System V IPC包括: System V消息队列、System V信号灯、
- System V共享内存区
- Posix IPC包括: Posix消息队列、Posix信号灯、Posix共享内存区

Linux系统进程通信技术

- 管道 (Pipe) : 管道可用于具有亲缘关系进程间的通信
- 有名管道 (namedpipe) : 有名管道克服了管道没有名字的限制, 因此, 除具有管道所具有的功能外, 它还允许无亲缘关系进程间的通信
- 信号 (Signal) : 用于通知接受进程有某种事件发生, 用于进程内、进程间通信
- 报文 (Message) 队列 (消息队列) : 消息队列是消息的链接表, 有足够权限的进程可以向队列中添加消息, 被赋予读权限的进程则可以读走队列中的消息
- 共享内存: 多个进程可以访问同一块内存空间, 最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的
- 信号量 (semaphore) : 主要作为进程间以及同一进程不同线程之间的同步手段
- 套接口 (Socket) : 更为一般的进程间通信机制, 可用于不同机器之间的进程间通信



linux进程包含以下关键要素

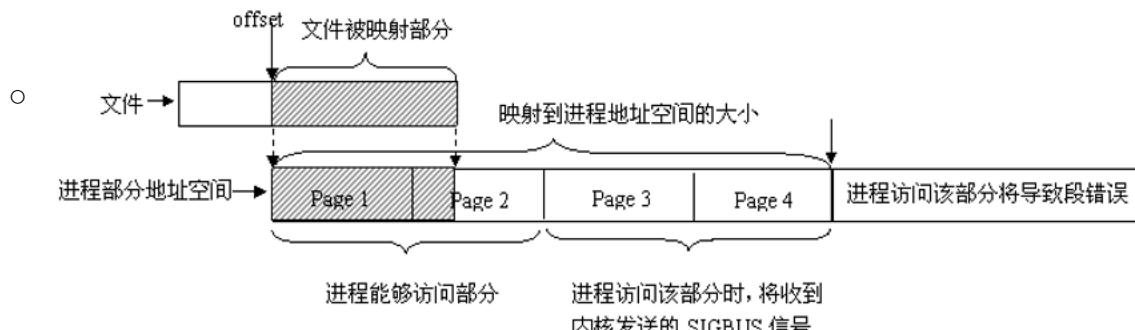
- 有一段可执行程序;
- 有专用的系统堆栈空间;
- 内核中有它的控制块 (进程控制块), 描述进程所占用的资源, 这样进程才能接受内核的调度;
- 具有独立的存储空间;

Linux内存映射技术

- 共享内存: 最有用的进程间通信方式, 也是最快的IPC形式。
- 两个不同进程A、B共享内存的意思是, 同一块物理内存被映射到进程A、B各自的进程地址空间。
- 进程A可以即时看到进程B对共享内存中数据的更新, 反之亦然。
- 由于多个进程共享同一块内存区域, 必然需要同步机制, 互斥锁和信号量都可以
- mmap()系统调用

● `void* mmap (void * addr , size_t len , int prot , int flags , int fd , off_t offset)`

● `int munmap(void * addr, size_t len)`



不同进程寻址同一个共享内存区域

- pagecache及swapcache中页面的区分
 - 一个被访问文件的物理页面都驻留在pagecache或swapcache中，一个页面的所有信息由structpage来描述。structpage中有一个域为指针mapping，它指向一个structaddress_space类型结构。pagecache或swapcache中的所有页面就是根据address_space结构以及一个偏移量来区分的。
- 文件与address_space结构的对应
 - 一个具体的文件在打开后，内核会在内存中为之建立一个structinode结构，其中的i_mapping域指向一个address_space结构。这样，一个文件就对应一个address_space结构，一个address_space与一个偏移量能够确定一个pagecache或swapcache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。
- 调用mmap()
 - 只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。
- 对于共享内存映射情况，缺页异常处理
 - 首先在swapcache中寻找目标页（符合address_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区(swaparea)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到pagecache中。进程最终将更新进程页表。
 - 注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在pagecache中根据address_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新。
- 多进程映射同一个共享内存区域
 - 情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。
 - 注：一个共享内存区域可以看作是特殊文件系统shm中的一个文件，shm的安装点在交换区上。

mmap()用于共享内存的两种方式

- 使用普通文件提供的内存映射
 - 适用于任何进程之间
 - 需要打开或创建一个文件，然后再调用mmap()
- 使用特殊文件提供匿名内存映射
 - 适用于具有亲缘关系的进程之间；
 - 由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。

Linux内核模块技术

- 基本原理
- 内核模块
- 设备驱动的结构
- Linux26内核设备模型
- 中断处理
- 各种接口设计与驱动开发实例

LINUX设备的分类

- 字符设备
 - 串口，终端，触摸屏
 - ls -l /dev/ttys0
crw-rw-rw- 1 root uucp 4, 64 4月 1 19:56 /dev/ttys0
- 块设备
 - FLASH, RAMDISK, 硬盘
 - ls -l /dev/mtdblock3
brw-r--r-- 1 505 505 31, 3 Feb 19 2005 /dev/mtdblock3
- 网络设备
 - eth0 Link encap:Ethernet HWaddr 00:30:48:21:D6:26
inet addr:172.16.24.143 Bcast:172.16.24.255 Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:4654989 errors:0 dropped:0 overruns:0 frame:0
TX packets:38456 errors:0 dropped:0 overruns:0 carrier:0
 - ifconfig -a

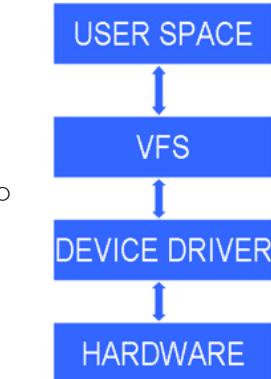
```
collisions:0 txqueuelen:1000
RX bytes:233261105 (222.4 Mb) TX bytes:5481796 (5.2 Mb)
Interrupt:31 Base address:0xd800 Memory:feafe000-afe038
```

设备文件与设备号

- 用户通过设备文件访问设备
- 每个设备用一个主设备号和次设备号标识

设备驱动的功能

- 管理I/O设备
- 上层软件的抽象操作与设备操作的转换



内核模块

- Linux内核运行时动态扩展的一种技术
- 一组可以动态加载/卸载的代码
- Linux驱动以内核模块的方式实现

LINUX内核模块的框架

```
static int init_routine (void)
{
    ...
}

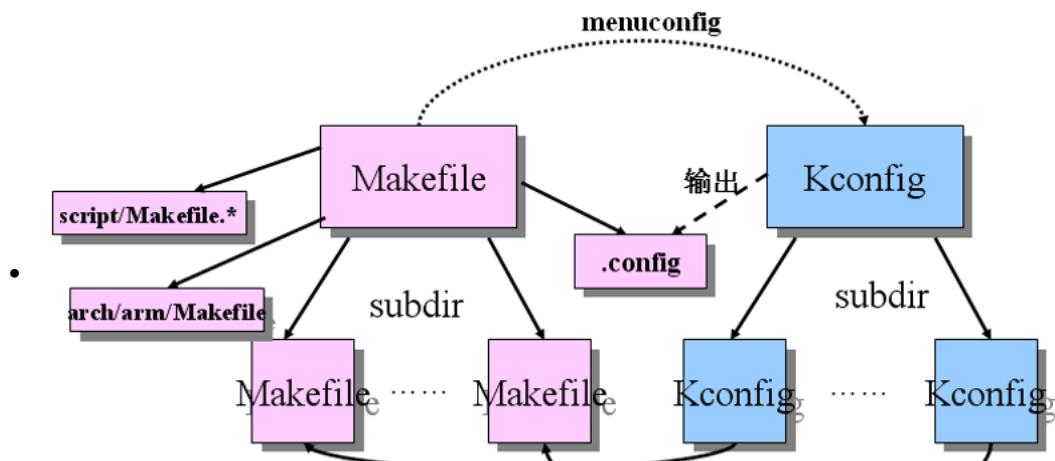
void cleanup_routine(void)
{
    ...
}

module_init(init_routine);
module_exit(cleanup_routine);
MODULE_LICENSE("GPL");
```

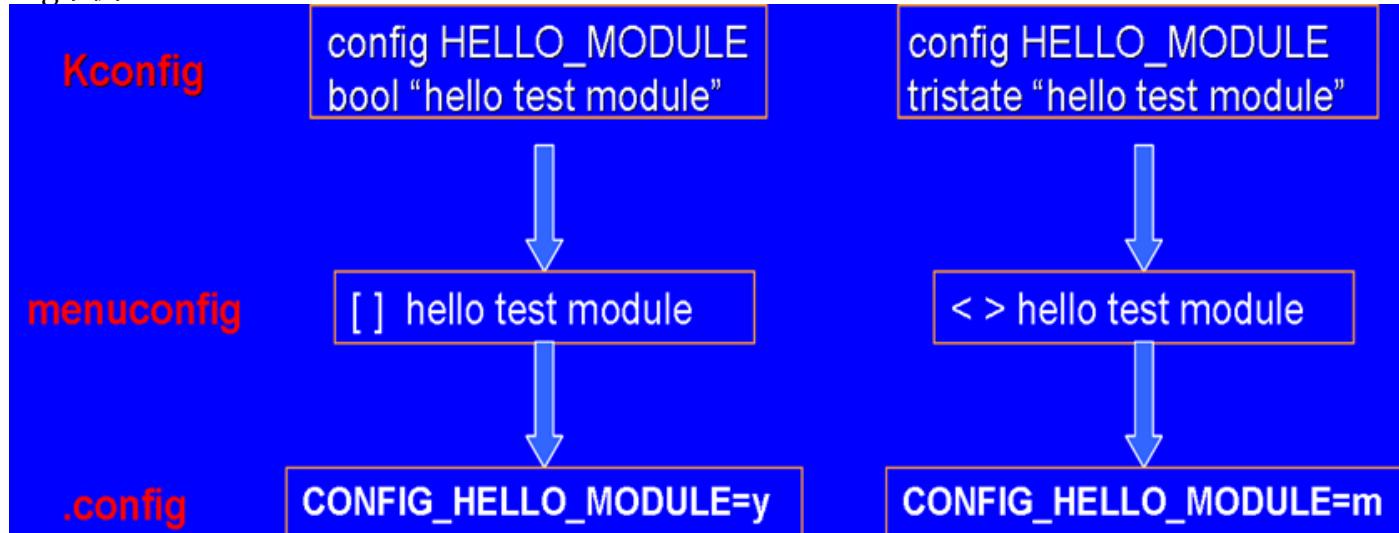
内核模块的编译和加载

- Kconfig
- Makefile

Kconfig与Makefile的关系



Kconfig示例



Makefile示例

- 定义示例
 - obj-y += hello.o
 - obj-m += hello.o
 - obj-\$(CONFIG_HELLO_MODULE) += hello.o
- 编译
 - make -C <PATH_TO_KERNEL> M = \$PWD modules

设备驱动的结构

- 驱动与内核的接口
 - 注册/卸载
 - VFS接口
 - 数据交互
 - 中断注册
- 硬件设备接口
 - 硬件探测
 - 初始化
 - 读写访问
 - 设备控制

设备驱动与VFS的接口

```
struct file_operations {  
    struct module *owner;  
    loff_t (*llseek)(struct file *, loff_t, int);  
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);  
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);  
    ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t);  
    int (*readdir)(struct file *, void *, filldir_t);  
    unsigned int (*poll)(struct file *, struct poll_table_struct *);  
    int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);  
    int (*mmap)(struct file *, struct vm_area_struct *);  
    int (*open)(struct inode *, struct file *);  
    int (*flush)(struct file *);  
    int (*release)(struct inode *, struct file *);  
    int (*fsync)(struct file *, struct dentry *, int datasync);  
    int (*aio_fsync)(struct kiocb *, int datasync);  
    int (*fasync)(int, struct file *, int);  
    int (*lock)(struct file *, int, struct file_lock *);  
    ssize_t (*readv)(struct file *, const struct iovec *, unsigned long, loff_t *);  
    ssize_t (*writev)(struct file *, const struct iovec *, unsigned long, loff_t *);  
    ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void __user *);  
    ssize_t (*sendpage)(struct file *, struct page *, int, size_t, loff_t *);  
};
```

```

    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    long (*fcntl)(int fd, unsigned int cmd,
        unsigned long arg, struct file *filp);
} ? end_file_operations ? ;

```

简单字符设备驱程的框架

```

static struct file_operations driver_fops = {
    ...
};

int __init init_routine(void)
{
    ...
    cdev_add(...);
    return 0;
}

void cleanup_routine(void)
{
    cdev_del(...);
}

```

内核与用户空间数据交换方法

- `copy_to_user(void __user *to, const void *from, unsigned long n);`
- `copy_from_user(void *to, const void __user *from, unsigned long n);`
- `mmap(.....)`

示例——kernel代码

```

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Wheelz");
MODULE_DESCRIPTION("mmap demo");

static unsigned long p = 0;

static int __init init(void)
{
    //分配共享内存（一个页面）
    p = __get_free_pages(GFP_KERNEL, 0); //得到的当然是一个虚拟地址了
    SetPageReserved(virt_to_page(p)); //#define virt_to_page(kaddr) (mem_map + (_pa(kaddr)
        >> PAGE_SHIFT)) / 12
    printk("<1> p = 0x%08x\n", p);
    //在共享内存中写上一个字符串
    strcpy(p, "Hello world!\n");
    return 0;
}

static void __exit exit(void)
{
    ClearPageReserved(virt_to_page(p));
    free_pages(p, 0);
}

```

示例——用户空间代码

```

#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

```

```

int main()
{
    char *buf;
    int fd;
    unsigned long phy_addr;

    fd=open("/dev/mem",O_RDWR);
    if(fd == -1)

```

```

#define PAGE_SIZE (4*1024)

#define PAGE_OFFSET      0xc0000000
#define KERNEL_VIRT_ADDR
0xcf9e5000 //这里是硬编址的， 可以通过ioctl 或者proc来实现的。

```

•

```

n_addr = phy_addr;
perror("open");
phy_addr=KERNEL_VIRT_ADDR -
PAGE_OFFSET;

buf=mmap(0, PAGE_SIZE,
PROT_READ|PROT_WRITE,
MAP_SHARED,
fd, phy_addr);
if(buf == MAP_FAILED)
perror("mmap");
puts(buf);//打印共享内存的内容

munmap(buf,PAGE_SIZE);

close(fd);
return 0;
}

```

Linux内核模块常用方法与命令

● Linux常用命令

- insmod
- rmmod
- lsmod
- dmesg

● 内核空间函数

- printk
- kmalloc
- kfree
- copy_to_user
- copy_from_user

● 用户空间函数

- printf
- malloc
- free
- ioctl(.....)

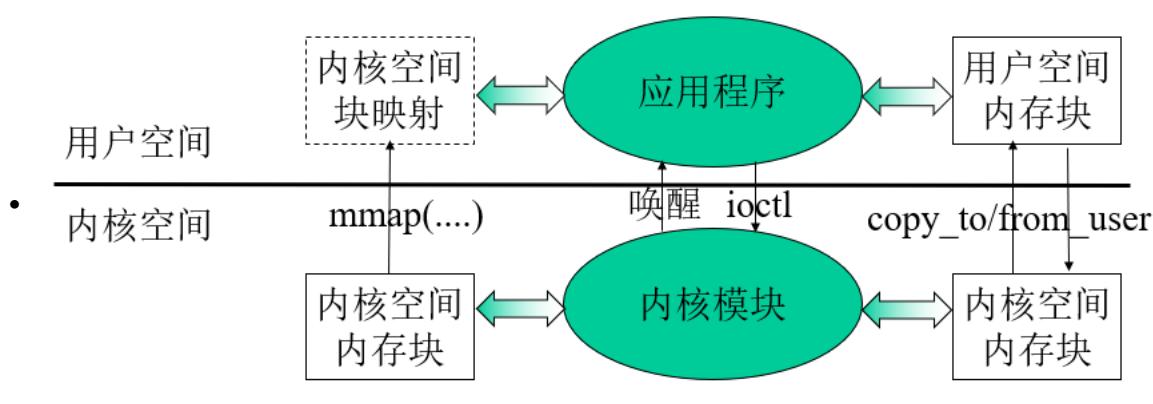
● 系统与内核模块交互方法

- model_init()
- model_exit()
- driver API

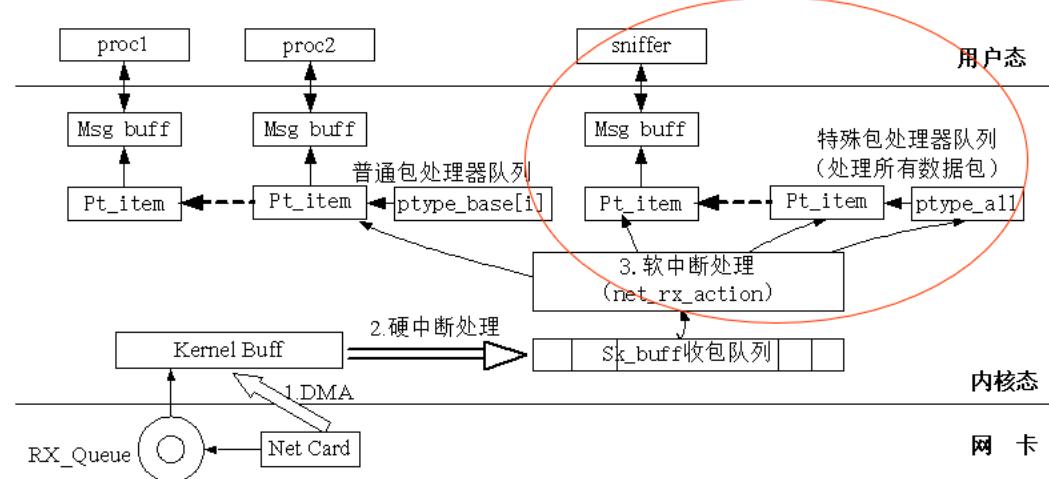
● 用户与内核模块交互方法

- ioctl(.....)
- copy_to/from_user(.....)
- put/get_user(.....)
- mmap(.....)

用户、内核、系统、驱动交互图



练习



- 宽带网络内容过滤平台整体架构
- 零拷贝技术
 - Linux系统内存映射技术
 - Linux内核模块加载技术
 - 共享内存无锁访问技术
- Linux网络报文处理
 - 传统报文获取技术
 - 网络报文检测协议栈
 - 网络动态阻断
- 快速应用协议识别技术
- 高速串匹配技术

TCP/IP协议与以太网通讯

Linux网络数据包监测技术

- libpcap提供的接口函数主要实现和封装了与数据包截获有关的过程。

Linux网络数据包组装与发送技术

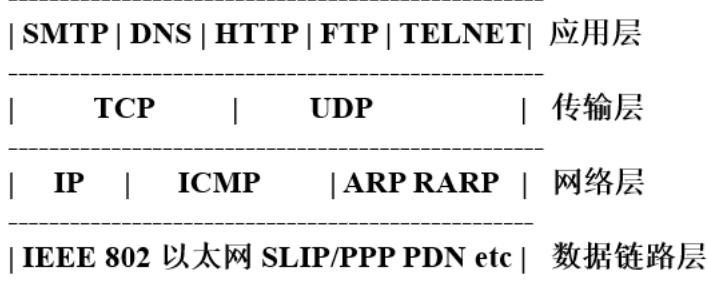
- libnet提供的接口函数主要实现和封装了数据包的构造和发送过程。

Linux网络数据流解析技术

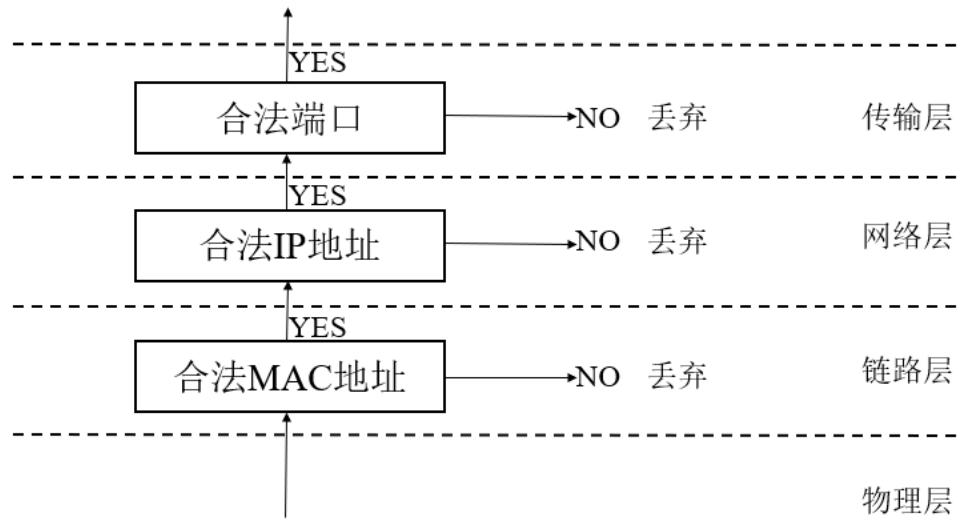
- libnids提供的接口函数主要实现了开发网络入侵监测系统所必须的一些结构框架。

1 TCP/IP协议与以太网通讯

TCP/IP体系结构



Linux数据包接收过程



以太网通信

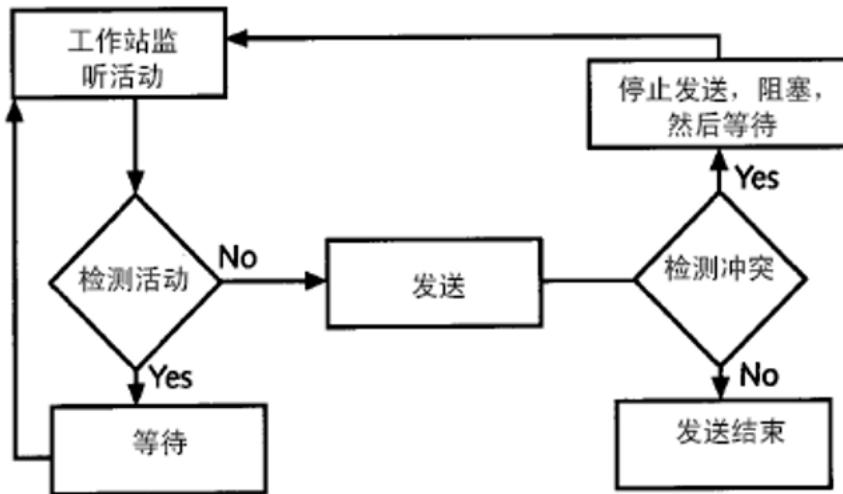
- 以太网最初是由XEROX公司研制,并且在1980年由数据设备公司DEC(DIGITAL EQUIPMENT CORPORATION)、INTEL公司和XEROX公司共同使之规范成形。后来它被作为802.3标准为电气与电子工程师协会(IEEE)所采纳。
- 以太网是最为流行的网络传输系统之一。以太网的基本特征是采用一种称为载波监听多路访问/冲突检测CSMA/CD (Carrier Sense Multiple Access/ Collision Detection)的共享访问方案。

TCP/IP与以太网

- 以太网和TCP/IP可以说是相辅相成的。
 - 以太网在一、二层提供物理上的连线, 使用48位的MAC地址

- TCP/IP工作在上层，使用32位的IP地址
- 两者间使用ARP和RARP协议进行相互转换。
- 载波监听
 - 指在以太网中的每个站点都具有同等的权利，在传输自己的数据时，首先监听信道是否空闲，如果空闲，就传输自己的数据，如果信道被占用，就等待信道空闲。
- 冲突检测
 - 为了防止发生两个站点同时监测到网络没有被使用时而产生冲突。以太网采用广播机制，所有与网络连接的工作站都可以看到网络上传递的数据

CSMA/CD过程



以太网的广播通讯

- 在以太网中，所有的通讯都是广播的，
 - 通常在同一个网段的所有网络接口都可以访问在物理媒体上传输的所有数据
- 网卡的MAC地址
 - 每一个网络接口都有一个唯一的硬件地址，这个硬件地址也就是网卡的MAC地址。
 - 大多数系统使用48比特的地址，这个地址用来表示网络中的每一个设备
 - 一般来说每一块网卡上的MAC地址都是不同的
 - 每个网卡厂家得到一段地址，然后用这段地址分配给其生产的每个网卡一个地址。
- 在正常的情况下，网络接口应该只响应这样的两种数据帧：
 - 与自己硬件地址相匹配的数据帧。
 - 发向所有机器的广播数据帧。
- 数据的收发是由网卡来完成的
 - 网卡接收到传输来的数据，网卡内的单片程序接收数据帧的目的MAC地址，根据计算机上的网卡驱动程序设置的接收模式判断该不该接收。
 - 认为该接收就接收后产生中断信号通知CPU
 - 认为不该接收就丢掉不管，所以不该接收的数据网卡就截断了，计算机根本就不知道
 - CPU得到中断信号产生中断，操作系统就根据网卡的驱动程序设置的网卡中断程序地址调用驱动程序接收数据
 - 驱动程序接收数据后放入信号堆栈让操作系统处理。
- 网卡来说一般有四种接收模式：
 - 广播方式：该模式下的网卡能够接收网络中的广播信息。
 - 组播方式：设置在该模式下的网卡能够接收组播数据。
 - 直接方式：在这种模式下，只有目的网卡才能接收该数据。
 - 混杂模式：在这种模式下的网卡能够接收一切通过它的数据，而不管该数据是否是传给它的。
- 总结一下
 - 首先，我们知道了在以太网中是基于广播方式传送数据的，也就是说，所有的物理信号都要经过我的机器，
 - 其次，广播模式下，网卡的一种模式叫混杂模式（promiscuous），在这种模式下工作的网卡能够接收到一切通过它的数据，而不管实际上数据的目的地址是不是它。
 - 再次，通过设置交换机监听端口。监听端口带宽要大于所监听的端口带宽，防止丢包。

2 基于Libpcap的网络编程技术

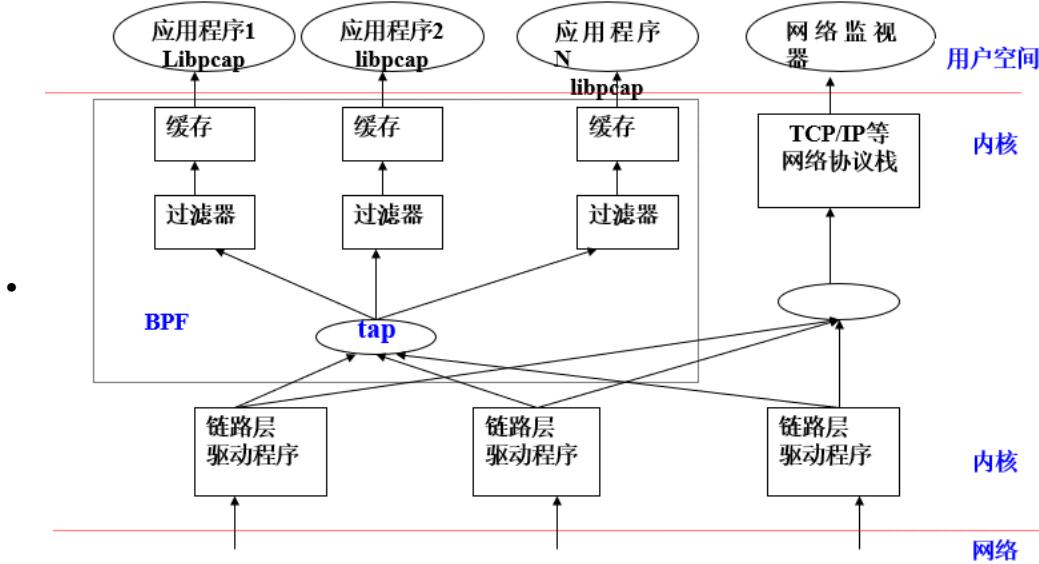
libpcap函数库介绍

- Libpcap(PacketCapture library), 即数据包捕获函数库。该库提供的C函数接口可用于捕获经过网络接口(只要经过该接口, 目标地址不一定为本机)的数据包
- 它是由洛伦兹伯克利(Berkeley)试验室的研究人员StevenMcCanne和VanJacobson于1993年在Usenix ‘93会议上正式提出的一种用于Unix内核数据包过滤体制。该函数库支持Linux、 Solaris 和BSD系统平台。采用libpcap可以捕获本地网络数据链路层上的数据。
- 广泛应用于:
 - 网络参数统计
 - 协议分析
 - 防火墙
 - 入侵检测系统
 - 网络调试
- libpcap库是基于BPF(BerkeleyPacketFilter: BSD包过滤器)系统的。
- BPF是BSD系统在TCP/IP软件实现时所提供的一个接口, 通过这个接口, 外部程序可以获取到达本机的数据链路层网络数据, 同时也可以设置过滤器, 嵌入到网络软件中, 获得过滤后的数据包。

BPF工作过程

- BPF是一种内核包捕获的体系结构, 它使得UNIX下的应用程序通过一个高度优化的方法读取流经网络适配器的数据包。
- BPF主要由两部分组成: Networktap和PacketFilter。
- Networktap是一个回调函数 (callbackfunction) , 它实时监视共享网络中的所有数据包, 从网络设备驱动程序中搜集数据拷贝并转发给包过滤器。
- 包过滤器决定是否接收该数据包, 以及接收该数据包的那些部分 (Slice(分片)技术) 。

BPF过滤机制



Libpcap库函数接口简介

libpcap所提供的主要函数如下:

- libpcap所提供的主要函数如下:
 - ① `char *pcap_lookupdev()`; 返回一个适于`pcap_open_live()`和`pcap_lookupnet()`函数使用的指向网络设备的指针
 - ② `pcap_t *pcap_open_live()`; 用于获取一个包捕获描述符
 - ③ `int pcap_lookupnet()`; 用于返回与网络设备相关的网络号和掩码
 - ④ `int pcap_dispatch()`或`int pcap_loop()`; 收集和处理数据包
 - ⑤ `void pcap_dump()`; 将一个包输出到由`pcap_dump_open()`打开的文件中保存
 - ⑥ `int pcap_compile()`; 用于将过滤规则字符串编译成一个内核过滤程序
 - ⑦ `int pcap_setfilter()`; 设定一个过滤程序
 - ⑧ `int pcap_datalink()`; 返回数据链路层类型, 如10M以太网, SLIP, PPP, FDDI, ATM, IEEE802.3等
 - ⑨ `void pcap_close()`; 关闭关联设备(文件)并回收资源
 - ⑩ `int pcap_stats(pcap_t *, struct pcap_stat *)`; 参数统计
 - ⑪ `int pcap_read(pcap_t *, int cnt, pcap_handler, u_char *)`; 打开设备

Libpcap: dump文件格式

dump文件头:

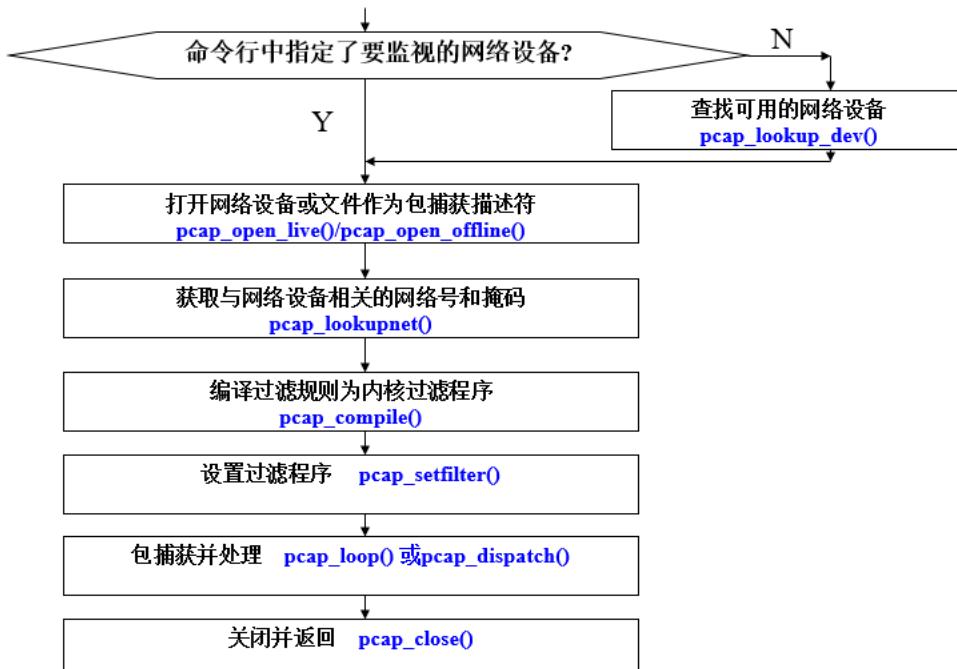
```
struct pcap_file_header {  
    bpf_u_int32 magic; // 0xa1b2c3d4  
    u_short version_major;  
    u_short version_minor;  
    bpf_int32 thiszone; /*本地时间*/  
    bpf_u_int32 sigfigs; /*时间戳*/  
    bpf_u_int32 snaplen; /*数据长度*/  
    bpf_u_int32 linktype; /*链路类型*/  
};
```

每一个包的包头和数据

```
struct pcap_pkthdr {  
    struct timeval ts;  
    bpf_u_int32 caplen;  
    bpf_u_int32 len;  
};
```

其中数据部分的长度为caplen

Libpcap应用步骤



Libpcap应用步骤及关键的函数使用方法

1. 获取设备名

`char *pcap_lookupdev(char *errbuf)`

该函数用于返回可被pcap_open_live()或pcap_lookupnet()函数调用的网络设备名（一个字符串指针）。如果函数出错，则返回NULL，同时errbuf中存放相关的错误消息。

2. 获取网络号和掩码

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

获得指定网络设备的网络号和掩码。netp参数和maskp参数都是bpf_u_int32指针。如果函数出错，则返回-1，同时errbuf中存放相关的错误消息。

3. 打开设备

`pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)`

获得用于捕获网络数据包的数据包捕获描述字。

✓ device参数为指定打开的网络设备名。

✓ snaplen参数定义捕获数据的最大字节数。

- ✓ `promisc`指定是否将网络接口置于混杂模式： `promisc=1` 混杂模式，`0`正常模式；。
- ✓ `to_ms`参数指定超时时间（毫秒）。
- ✓ `ebuf`参数仅在 `pcap_open_live()` 函数出错返回 `NULL` 时用于传递错误消息。

4. 编译和设置过滤器

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

将 `str` 参数指定的字符串编译到过滤程序中。 `fp` 是一个 `bpf_program` 结构的指针，在 `pcap_compile()` 函数中被赋值。`optimize` 参数控制结果代码的优化。`netmask` 参数指定本地网络的网络掩码。

```
int pcap_setfilter(pcap_t *p, struct
```

`bpf_program *fp)` 指定一个过滤程序。 `fp` 参数是 `bpf_program` 结构指针，通常取自 `pcap_compile()` 函数调用。出错时返回 `-1`；成功时返回 `0`。

• 5. 抓取数据包

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

`cnt` 参数指定函数返回前所处理数据包的最大值。`cnt=-1` 表示在一个缓冲区中处理所有的数据包。`cnt=0` 表示处理所有数据包，直到产生以下错误之一：读取到 EOF；超时读取。

`callback` 参数指定一个带有三个参数的回调函数，这三个参数为：`typedef void (*pcap_handler) (u_char *args, const struct pcap_pkthdr *header, const u_char *packet);`

`args`: `pcap_dispatch()` 函数传递过来的第四个形参，一般我们自己的包捕捉程序不需要提供它，总是为 `NULL`；

`header`: 指向 `pcap_pkthdr` 结构，该结构位于真正的物理帧前面，用于消除不同链路层支持的差异；

`packet`: 指向所捕获报文的物理帧。

参数 `user` 为用户传递给回调函数的指针。

如果成功则返回读取到的字节数。读取到 EOF 时则返回零值。出错时则返回 `-1`，此时可调用 `pcap_perror()` 或 `pcap_geterr()` 函数获取错误消息。

- `int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)` 功能基本与 `pcap_dispatch()` 函数相同，只不过此函数在 `cnt` 个数据包被处理或出现错误时才返回，但读取超时不会返回。而如果为 `pcap_open_live()` 函数指定了一个非零值的超时设置，然后调用 `pcap_dispatch()` 函数，则当超时发生时 `pcap_dispatch()` 函数会返回。`cnt` 参数为负值时 `pcap_loop()` 函数将始终循环运行，除非出现错误。

例如：`pcap_loop(pd, 10, eth_printer, NULL);`

主循环，开始抓包，共抓 10 个（由第二个参数指定），抓到包后就进入回调函数 `eth_printer` 处理

- `u_char *pcap_next(pcap_t *p, struct`

`pcap_pkthdr *h)` 返回指向下一个数据包的 `u_char` 指针。

- 6. 关闭
- **void pcap_close(pcap_t *p)**关闭p参数相应的文件，并释放资源。
- 7. 其他的辅助函数
- **FILE *pcap_file(pcap_t *p)**返回被打开文件的文件名。
- **int pcap_fileno(pcap_t *p)**返回被打开文件的文件描述字号码。
- **pcap_read()**这个函数从包捕获驱动器中读取一组数据包并针对每一个包运行包过滤程序,然后把过滤后的数据送应用程序缓冲器。
- **pcap_next()**循环包捕获

其他的辅助函数

- 脱机方式监听部分：Libpcap 支持脱机监听。即先将网络上的数据截获下来，存到磁盘上，在以后方便的时候再从磁盘上获取数据来做分析。主要函数为
 - **pcap_open_offline()**
 - **pcap_offline_read()**
- **int pcap_datalink(pcap_t *)**;返回网络类型,如 DLT_EN10MB 就是10M以太网
- **int pcap_major_version(pcap_t *)**;返回libpcap主版本号
- **int pcap_minor_version(pcap_t *)**;返回libpcap次版本号
- **int pcap_snapshot(pcap_t *)**；
返回最长抓多少字节,就是在pcap_open_live中第二个参数设置
- **int pcap_stats(pcap_t *, struct pcap_stat *)**；
计数,共抓了多少,过滤掉了多少

```
struct pcap_stat {
    u_int ps_recv; /* number of packets received */
    u_int ps_drop; /* number of packets dropped */
    u_int ps_ifdrop; /* drops by interface XXX not yet
supported */
};
```

出错处理

- 象其它库一样, libpcap 也有自己的错误处理机制。
基本上每个函数都有返回值,出错时返回值 < 0

```
void pcap_perror(pcap_t *, char *);
char *pcap_strerror(int);
char *pcap_geterr(pcap_t *);
```
- 在 pcap_t 中有一个成员存放着错误字串

```
struct pcap {
```

```
    ...
    char errbuf[PCAP_ERRBUF_SIZE];
};
```

安装配置

安装前需下载下列文件：

1) libpcap.tar.z 公用函数库：

下载地址：

<ftp://ftp.ee.lbl.gov/libpcap.tar.z>

2) libnet-1.0.2a.tar.gz底层支持：

<http://www.packetfactory.net>

3) NDIS Packet Capture Driver for windows：

<http://www.datanerds.net/~mike>

For Linux用户安装

*安装libpcap：先检查文件下列文件是否存在， /usr/local/lib/libpcap.a，

• /usr/include/pcap/pcap.h , pcap-namedb.h和 /usr/include/pcap/net/bpf.h，如都存在表明系统中已安装libpcap（注意：Linux Redhat 7.0 及之后版本已经集成了该包）。

如没安装则操作：

#cd /tmp

#tar -zxvf libpcap.tar.z

• #cd libpcap

 #./configure; make; make check; make

install

如无任何报错信息，表明安装成功。

例一个简单的Libpcap测试程序

```
• #ifdef __cplusplus
    extern "C" {
#endif
#include <pcap.h>
#ifdef __cplusplus
}
#endif

void printer(u_char * user, const struct pcap_pkthdr * h,
const u_char * p)
{
```

```

        printf("I get one packet! ");
    }

#define DEFAULT_SNAPLEN 68

int main()
{
    char ebuf[PCAP_ERRBUF_SIZE];
    char *device = pcap_lookupdev(ebuf);

    bpf_u_int32 localnet, netmask;
    pcap_lookupnet(device, &localnet, &netmask, ebuf);
    printf("%u.%u.%u.%u", localnet&0xff, localnet>>8&0xff,
           localnet>>16&0xff, localnet>>24&0xff);
    printf(":%d.%d.%d.%d ", netmask&0xff,
           netmask>>8&0xff,
           netmask>>16&0xff, netmask>>24&0xff);

    struct pcap_t *pd = pcap_open_live(device,
                                       DEFAULT_SNAPLEN, 0, 1000, ebuf);

    if(pcap_datalink(pd) == DLT_EN10MB)
        printf("10Mb以太网 ");

    struct bpf_program fcode;
    pcap_compile(pd, &fcode, NULL, 1, 0);
    pcap_setfilter(pd, &fcode);

    pcap_loop(pd, 10, printer, NULL);

    struct pcap_stat stat;
    pcap_stats(pd, &stat);
    printf("recv %d, drop %d. ", stat.ps_recv, stat.ps_drop);

    pcap_close(pd);
}

```

3 基于Libnet的网络编程技术

Unix系统平台上的API library

- 在众多的APIlibrary中，对于类Unix系统平台而言，目前最为流行的C API library有：
 - Libnet
 - Libpcap
 - Libnids
 - Libicmp等
- 它们分别从不同层次和角度提供了不同的网络功能函数。使网络开发人员能够忽略网络底层细节的实现，从而专注于程序本身具体功能的设计与开发。其中：

Unix系统平台上的APIlibrary

- libnet提供的接口函数主要用于实现数据包的构造和发送。
- libpcap提供的接口函数主要用于实现数据包截获(接收)。
- libnids提供的接口函数主要实现了开发网络入侵监测系统(nids)所必须的一些结构框架。
- libicmp封装的是ICMP数据包的主要处理过程(构造、发送、接收等)。
- 利用这些C函数库的接口，开发人员可以很方便地编写出具有结构化强、健壮性好、可移植性高等特点的网络通信程序。
- 这些函数库在网络应用开发中具有很大的实用价值，在scanner、sniffer、firewall、IDS等领域都获得了极其广泛的应用，著名的抓包软件如tcpdump、ethereal等就是在libpcap的基础上开发的。著名的网络嗅探器软件Sniffer就是在libpcap的基础上开发的。

- **Libnet概述**
- **功能:数据包构造和发送**
- **libnet**是一个高层API (toolkit)，主要用C语言写成，为应用程序设计人员提供了低层的网络数据报的构造、处理和发送等功能接口。
- **libnet**使得程序员从乏味的报文创建工作（如多路复用、缓冲区管理、神秘的报头信息(域、位置、长度)、字节顺序、操作系统相关问题等）中解脱出来，将精力集中在解决关键问题上。
- 利用**libnet**，可以方便、快速、简单地完成报文组装工作，稍加扩展，还可以编写出复杂的应用程序（如Traceroute(路由跟踪)和ping就可以方便地通过**libnet**和**libpcap**来实现）。
- **libnet**库提供的接口函数包含**15种数据包生成器**和**两种数据包发送器**(IP层和数据链路层)。
- 提供了**50多个C API**函数，功能涵盖
 - 内存管理(分配和释放)函数
 - 地址解析函数
 - 各种协议类型的数据包构造函数，包括应用层（如DNS, RIP, SNMP等）传输层（如TCP, UDP等）、网络层（如IP, ARP, ICMP, IGMP, OSPF等）和数据链路层（如Ethernet帧等）
 - 数据包发送函数(IP层和链路层)
 - 一些辅助函数，如产生随机数、错误报告、端口列表管理等

内存管理函数

- **单数据包内存初始化:**
`int libnet_init_packet(u_short packet_size, u_char **buf);`
- **单数据包内存释放:**
`void libnet_destroy_packet(u_char **buf);`
- **多数据包（缓冲池）内存初始化:**
`Int libnet_init_packet_arena(struct libnet_arena **arena, u_short packet_num, u_short packet_size);`
- **访问多数据包（缓冲池）内存中的下一个数据包:**
`u_char *libnet_next_packet_from_arena(struct libnet_arena **arena, u_short packet_size);`
- **多数据包内存释放:**
`void libnet_destroy_packet_arena(struct libnet_arena **arena);`

地址解析函数

- **解析主机名:**
- `u_char *libnet_host_lookup(u_long ip, u_short use_name);`
- **解析主机名(可重入函数):**
- `void libnet_host_lookup_r(u_long ip, u_short use_name, u_char *buf);`
- **域名解析:**
- `u_long libnet_name_resolve(u_char *ip, u_short use_name);`
- **获取接口设备IP地址:**

- **获取接口设备IP地址:**
 - `u_long libnet_get_ipaddr(struct libnet_link_int *l,`
 - `const u_char *device, const u_char *ebuf);`
- **获取接口设备硬件地址:**
 - `struct ether_addr *libnet_get_hwaddr(struct libnet_link_int *l,`
 - `const u_char *device,`
 - `const u_char *ebuf);`

数据包构造函数

- ARP协议数据包
 - DNS协议数据包
 - 以太网帧数据包
 - IGMP协议数据包
 - IP协议数据包
 - IP协议数据包选项
 - UDP协议数据包
 - TCP协议数据包
 - TCP协议数据包选项
-
- ICMP协议数据包(ICMP_ECHO / ICMP_ECHOREPLY 回声请求/应答)----*Ping*
 - ICMP协议数据包(ICMP_MASKREQ / ICMP_MASKREPLY 地址掩码请求/应答)
 - ICMP协议数据包(ICMP_UNREACH 目的地不可达)
 - ICMP协议数据包(ICMP_TIMEXCEED 超时)
 - ICMP协议数据包(ICMP_REDIRECT 路由重定向)
 - ICMP协议数据包(ICMP_TSTAMP / ICMP_TSTAMPREPLY 时间戳请求/应答)
-
- RIP路由协议数据包
 - OSPF路由协议数据包
 - OSPF路由协议数据包(Hello)
 - OSPF路由协议数据包(DataBase Description (DBD))
 - OSPF路由协议数据包(Link State Request (LSR))
 - OSPF路由协议数据包(Link State Update (LSU))
 - OSPF路由协议数据包(Link State Acknowledgement (LSA))
 - OSPF路由协议数据包(Link State Router)
 - OSPF路由协议数据包(Link State Summary)
 - OSPF路由协议数据包(Link State AS External)

- 常量名 数值(字节数)
- LIBNET_ETH_H 14
- LIBNET_IP_H 20
- LIBNET_RIP_H 24
- LIBNET_TCP_H 20
- LIBNET_UDP_H 8
- LIBNET_ARP_H 28
- LIBNET_DNS_H 12
- LIBNET_ICMP_H 4
- LIBNET_ICMP_ECHO_H 8
- LIBNET_ICMP_MASK_H 12
- LIBNET_ICMP_UNREACH_H 8
- LIBNET_ICMP_TIMXCEED_H 8
- LIBNET_ICMP_REDIRECT_H 8
- LIBNET_ICMP_TS_H 20
- LIBNET_IGMP_H 8

• 数据包内存常量：

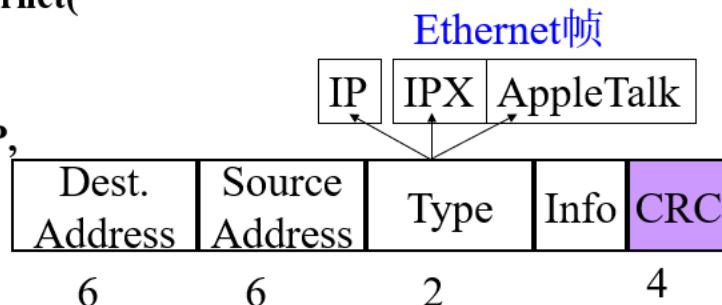
- LIBNET_PACKET TCP/UDP数据包头 + IP数据包头使用的内存
- LIBNET_OPTS IP或TCP选项使用的内存
- LIBNET_MAX_PACKET IP_MAXPACKET (65535字节)使用的内存

构造以太帧数据包

```
int libnet_build_ether(u_char *daddr, u_char *saddr,
    u_short type, const u_char *payload, int payload_len,
    u_char *packet_buf);
```

例：构造一个不含数据的以太帧

- /* Packet construction (ethernet header). */
libnet_build_ether(
 enet_dst,
 enet_src,
 ETHERTYPE_IP,
 NULL,
 0,
 packet);



ARP协议数据包

```
int libnet_build_arp(u_short hrdw, u_short
    prot, u_short h_len, u_short p_len,
    u_short op, u_char *s_ha, u_char *s_pa,
```

```
u_char *t_ha, u_char *t_pa,const u_char  
*payload, int payload_len,  
u_char *packet_buf);
```

构造ARP协议数据包

```
t = libnet_build_arp(  
ARPHRD_ETHER, /* hardware addr */  
ETHERTYPE_IP, /* protocol addr */  
6, /* hardware addr size */  
4, /* protocol addr size */  
ARPOP_REPLY, /* operation type */  
enet_src, /* sender hardware addr */  
ip_src, /* sender protocol addr */  
enet_dst, /* target hardware addr */  
ip_dst, /* target protocol addr */  
NULL, /* payload */  
0, /* payload size */  
packet); /* packet buffer */  
if (t == -1){ fprintf(stderr, "Can't build ARP header: %s\n",  
libnet_geterror(l)); goto bad; }
```

IP协议数据包

构造IP分组数据包：

```
int libnet_build_ip(u_short len, u_char tos,  
u_short ip_id, u_short frag,  
u_char ttl, u_char protocol, u_long saddr,  
u_long daddr, const u_char *payload, int  
payload_len,u_char *packet_buf);
```

构造IP协议数据包

- `libnet_build_ip(LIBNET_IP_H, /* size of the IP header */
IPTOS_LOWDELAY, /* IP tos */
242, /* IP ID */
0, /* frag stuff */
48, /* TTL */
 IPPROTO_TCP, /* transport control protocol */
src_ip, /* source IP address */
dst_ip, /* destination IP address */
NULL, /* payload (none) */
0, /* payload length */
packet); /* packet header memory */`

计算IP校验和

- 调用函数来计算TCP/IP协议的校验和
`libnet_do_checksum(u_char *packet,int protocol,int packet_size)`进行校验。
- 校验的协议由参数`protocol`决定,例:

```
if (libnet_do_checksum(packet, IPPROTO_IP,
LIBNET_IP_H) == -1)
{
    libnet_error(LIBNET_ERR_FATAL,
"libnet_do_checksum failed\n");
}
```

TCP/UDP协议数据包

- TCP协议数据包构造:

```
int libnet_build_tcp(u_short th_sport, u_short
th_dport, u_long th_seq,
u_long th_ack, u_char th_flags, u_short th_win,
u_short th_urg, const u_char *payload,
int payload_len, u_char *packet_buf);
```

- UDP协议数据包构造:

```
int libnet_build_udp(u_short sport, u_short dport,
const u_char *payload,int payload_len,
u_char *packet_buf);
```

构造TCP协议数据包

```
/* Packet construction (TCP header). */
libnet_build_tcp(src_prt, /* source TCP port */
dst_prt, /* destination TCP port */
0xa1d95, /* sequence number */
0x53, /* acknowledgement number */
TH_SYN, /* control flags */
1024, /* window size */
0, /* urgent pointer */
NULL, /* payload (none) */
0, /* payload length */
packet + LIBNET_IP_H); /* packet header memory */
```

数据包发送函数

- 打开raw socket:
`int libnet_open_raw_sock(int protocol);`
- 关闭raw socket:
`int libnet_close_raw_sock(int socket);`
- 选择接口设备:
`int libnet_select_device(struct sockaddr_in *sin, u_char **device, u_char *ebuf);`
- 打开链路层接口设备:
`struct libnet_link_int *libnet_open_link_interface (char *device,char *ebuf);`
- 关闭链路层接口设备:
`int libnet_close_link_interface(struct libnet_link_int *l);`

辅助函数

- 随机数种子生成器
- 获得随机数
- 16进制数据输出显示
- 端口列表链初始化
- 获得端口列表链的下一项(端口范围)
- 端口列表链输出显示
- 获得端口列表链
- 端口列表链内存释放

使用Libnet的基本过程

- | | |
|--------------|--|
| 1) 数据包内存初始化 | <code>libnet_init_packet(...);</code> |
| 2) 网络接口初始化 | <code>libnet_open_raw_sock(...);</code> |
| 3) 构造所需的数据包 | <code>libnet_build_ip(...);</code> |
| 4) 计算数据包的校验和 | <code>libnet_build_tcp(...);</code> |
| 5) 发送数据包 | <code>libnet_do_checksum(...);</code> |
| 6) 关闭网络接口 | <code>libnet_write_ip(...);</code> |
| 7) 释放数据包内存 | <code>libnet_close_raw_sock(...);</code> |
| | <code>libnet_destroy_packet(...);</code> |

Libnet的安装

- 以[libnet1.1.2.1](#)为例，安装步骤为：
- `# tar -zxf libnet.tar.gz`

- ```
tar -zXvf libnet.tar.gz
```
- # cd libnet
  - # ./configure
  - # make
  - # make install

## 应用实例

Snort

libnids

tcpdump

ssldump

sniffer

## 4 Linux网络数据流解析技术

- [4.1 libnids背景介绍](#)
- [4.2 libnids数据结构介绍](#)
- [4.3 libnids函数介绍](#)
- [4.4 libnids中TCP协议检测](#)
- [4.5 libnids开发示例](#)
- [4.6 libnids升级换代](#)

### 4.1 libnids背景介绍

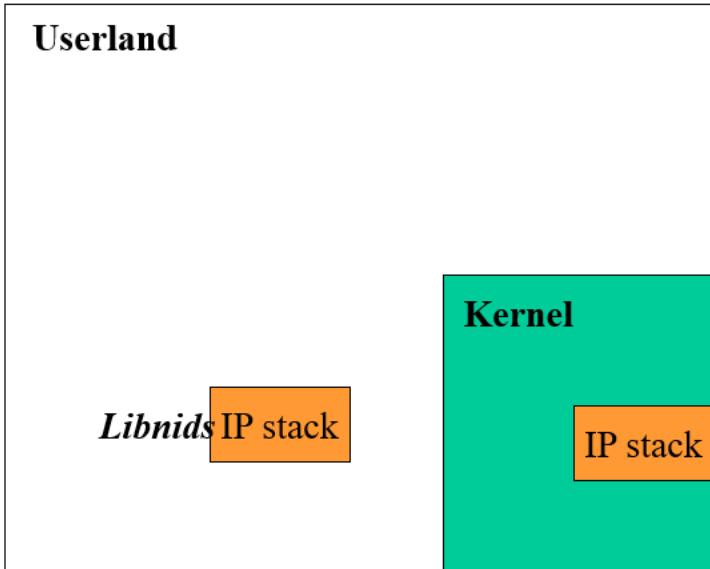
#### Libnids简介

- NetworkIntrusionDetectSystemlibrary, 即网络入侵监测系统函数库。
- 在libnet和libpcap的基础上开发的，封装了开发NIDS所需的许多通用型函数
- linids提供的接口函数监视流经本地的所有网络通信，检查数据包等。
- 除此之外，还具有重组TCP数据段、处理IP分片包和监测TCP端口扫描的功能
- libnids支持Linux、Solaris和\*BSD系统平台
- NIDS “E” box (event generation box)
- Userland TCP/IP stack
- Based on Linux 2.0.36 IP stack

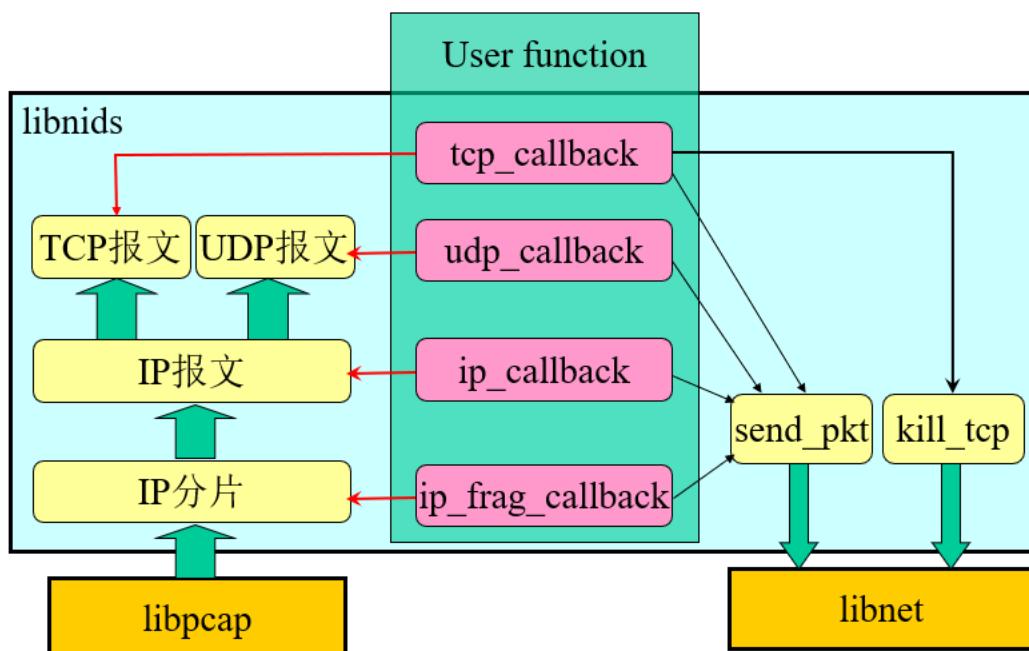
#### libnids背景介绍UserlandvsKernel

#### **Userland**

## libnids背景介绍UserlandIPStackvsKernelIPStack



## libnids背景介绍Libnids原理



## libnids背景介绍回调函数

- libnids的每级回调函数可以是多个
  - 用户可以注册多个ip\_callback
  - 用户可以注册多个tcp\_callback
  - 用户可以注册多个udp\_callback
- libnids的IP回调函数已经进行分片重组
  - ip\_callback回调函数中不会接收到IP分片

## 4.2 libnids数据结构介绍

### libnids主要数据结构

#### 1. tuple4：此数据结构是Libnids中最基本的一种数据结构

功能描述：用于描述一个地址端口对，它表示发送方IP和端口以及接收方IP和端口

#### struct tuple4

{

  u\_short source; //源端口

```

u_short dest;//目标端口
u_int saddr; //源IP
u_int daddr;//目的IP
};

```

2.half\_stream：此数据结构用来描述在tcp连接中一端的所有信息，可以使客户端也可以是服务端。

```

struct half_stream
{
char state;//表示套接字的状态
char collect;//表示是否保留数据
char collect_urg;//是否保留紧急数据
char *data;//存储正常接收的数据
int offset;//存储在data中的偏移量
int count;//表示data中数据字节数
int count_new;//表示新数据大小

```

```

int bufsize;
int rmem_alloc;
int urg_count;
u_int acked;
u_int seq;
u_int ack_seq;
u_int first_data_seg;
u_char urgdata;//用来存储紧急数据
u_char count_new_urg;//表示是否有新的紧急数据到达
u_char urg_seen;
u_int urg_ptr;
u_short window;
u_char ts_on;
u_int curr_ts;
struct skbuff *list;
struct skbuff *listtail;
}

```

3.tcp\_stream:描述的是一个TCP连接的所有信息

```

struct tcp_stream
{
struct tuple4 addr;//tcp连接四元组
char nids_state;//逻辑连接状态
struct lurker_node *listeners;
struct half_stream client;
struct half_stream server;
struct tcp_stream *next_node;
struct tcp_stream *prev_node;
int hash_index;
struct tcp_stream *next_time;
struct tcp_stream *prev_time;
int read;
struct tcp_stream *next_free;
};

```

4.nids\_prm:描述libnids的一些全局参数信息

```

struct nids_prm
{
int n_tcp_streams;//tcp流哈西表大小
int n_hosts;//ip碎片哈西表的大小
char *device;
char *filename;//已存储报文文件
int sk_buff_size;
int dev_addon;

```

```

int syslog_level;//表示日志等级
int scan_num_hosts;//表示存储端口扫描信息的哈西表的大小
int scan_delay;//表示在扫描检测中两端口扫描的间隔时间
int scan_num_ports;//表示相同源地址必须扫描的tcp端口数目
void (*no_mem)(char *);
int (*ip_filter)();
char *pcap_filter;
int promisc;

```

```
void (*syslog)(); //函数指针，可以
 检测入侵攻击,如:网络扫描攻击 int one_loop_less; int pcap_timeout;
};

函数定义类型为nids_syslog(int
 type,int errnum,struct
 ip_header * iph,void *data)
```

## 4.3 libnids函数介绍

### libnids基本函数

- (1)int nids\_init(void): 对libnids进行初始化
- (2)void nids\_run(void): 运行Libnids进入循环捕获数据包状态。
- (3)int nids\_getfd(void): 获得文件描述号
- (4)int nids\_dispatch(int cnt):  
 功能是调用Libpcap中的捕获数据包函数pcap\_dispatch().
- (5)int nids\_next(void):  
 调用Libpcap中的捕获数据包函数pcap\_next()
- (6)void nids\_register\_chksum\_ctl(struct nids\_chksum\_ctl  
 \*ptr,int nr):  
 决定是否计算校验和,它是根据数据结构nids\_chksum\_ctl  
 中的action进行决定的

### IP碎片函数

- (1)void nids\_register\_ip\_frag(void(\*))
  - 此函数的功能是注册一个能够检测所有IP数据包的回调函数,包括IP碎片
  - nids\_register\_ip\_frag(ip\_frag\_function)这样就定义了一个回调函数ip\_frag\_function的定义类型如下:
  - void ip\_frag\_function(struct ip \*a\_packet,int len)
- (2)void nids\_register\_ip(void(\*))
  - 此函数定义一个回调函数,此回调函数可以接受正常的IP数据包
  - nids\_register\_ip(ip\_function)此回调函数的定义类型如下:
  - void ip\_function(struct ip \* a\_packet)

### UDP报文函数

- (1)void nids\_register\_udp(void(\*));
  - 此函数的功能注册一个分析UDP协议的回调函数,回调函数的类型定义如下:
    - void udp\_callback(struct tuple4 \*addr,char  
 \*buf,int len,struct ip \* iph);
      - 其中参数addr表示的是端口的信息,
      - 参数buf表示UDP协议负载数据内容,

- 参数len 表示UDP负载数据的长度;
- 参数iph表示一个IP数据包,包括IP首部,
- UDP首部以及UDP负载内容

TCP流函数

(1)void nids\_register\_tcp(void(\*))回调函数的功能是  
注册一个TCP连接的回调函数

回调函数的类型定义如下:

- void tcp\_callback(struct tcp\_stream \*ns,void \*\*param);
- 其中参数ns表示一个tcp连接的所有信息,它的类型是tcp\_stream数据结构;
- 参数param表示要传递的连接参数信息,可以指向一个TCP连接的私有数据

(2)void nids\_killtcp(struct tcp\_stream \* a\_tcp)//此函数功能是终止TCP连接

(3)void nids\_discard(struct tcp\_stream \*a\_tcp,int num)//丢弃num字节TCP数据,用于存储更多的数据

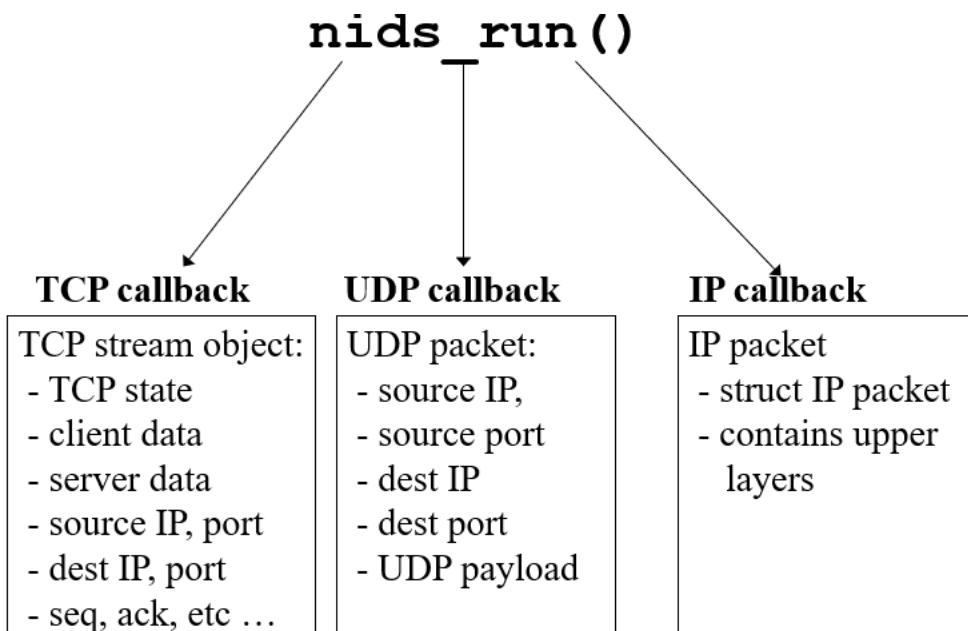
Libnids开发流程

• Initialize

- nids\_init()
- Register callbacks
- nids\_register\_tcp(*tcp\_callback*)
  - nids\_register\_ip(*ip\_callback*)
  - nids\_register\_udp(*udp\_callback*)

• Run!

- nids\_run()
- React
- 在tcp\_callback中调用nids\_kill\_tcp()



## TCP协议检测tcp\_callback

- **libnids的工作**

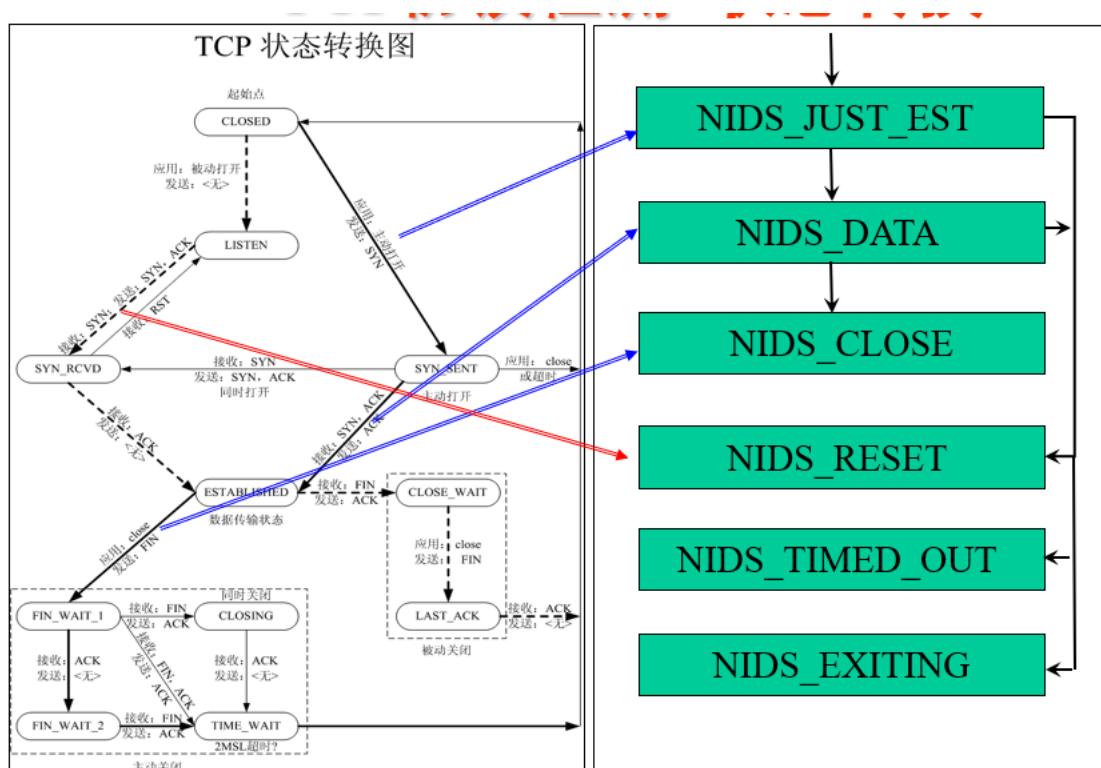
- TCP报文数据缓存

- 客户-服务器建立链接
- 客户端->服务器的数据
- 服务器->客户端的数据
- 客户-服务器链接终止

- TCP报文排序

- TCP状态管理

- 用户回调接口



## TCP协议检测tcp\_callback

```

tcp_callback(struct tcp *s)
{
 switch(s.state)
 {
 case NIDS JUST EST:
 what can we do here?
 case NIDS DATA:
 what can we do here?
 case NIDS CLOSE:
 what can we do here?
 case NIDS RESET:
 case NIDS TIMED OUT:
 what can we do here?
 }
}

tcp_callback(struct tcp *stream)
{
 switch(stream->state)

```

•新的TCP链接(3WHS)  
 •设置stream->{client,server}.collect=1

|                                                                                                                                                                                                                                                         |                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <pre>         case NIDS JUST EST:             what can we         case NIDS DATA:             what can we         case NIDS CLOSE:         case NIDS RESET:             what can we         case NIDS TIMED OUT:             what can we     } } </pre> | <ul style="list-style-type: none"> <li>• 某个TCP链接有新数据了</li> <li>• 根据stream结构体中相应字段判断</li> </ul> |
|                                                                                                                                                                                                                                                         | <ul style="list-style-type: none"> <li>• 某个TCP链接关闭了</li> <li>• 释放内存</li> </ul>                 |
|                                                                                                                                                                                                                                                         | <ul style="list-style-type: none"> <li>• 超时，无法继续跟踪该TCP</li> <li>• 释放内存</li> </ul>              |

## 4.5 libnids开发示例

开发示例：jflow

- **jflow**
  - Pcap to NetFlow v1 summaries
  - Daemonizes
  - Sends to a receiving host over UDP
- **Limitations of jflow**
  - Not very lightweight
  - Inaccurate for some things

<http://monkey.org/~jose/software/jflow/>

文件名称：tcpc

## 4.4.5 开发示例：jflow主函数

```

<includes>
<export record>
<ip callback>

int main (int argc, char *argv[])
{
 <getopt handler>
 <UDP socket connect>
 nids_init();
 nids_register_ip(monitor_ip);
 nids_run();
 return(0);
}

```

## 4.4.5 开发示例：jflow回调函数

```

void monitor_ip(struct ip *pkt)
{
 struct ip record rec;

```

```

int i;

rec.srcaddr = (u_int)(pkt->ip_src.s_addr);
rec.dstaddr = (u_int)(pkt->ip_dst.s_addr);
rec.nexthop = inet_addr("0.0.0.0");
rec.dOctets = htonl(pkt->ip_len);
rec.pad = 0x0;
rec.prot = pkt->ip_p;
rec.tos = 0x0;
rec.tcp_flags = 0x0;
rec.pad_2 = 0x0;
rec.pad_3 = 0x0;
for (i = 0; i < 4; i++)
 rec.reserved[i] = 0x0;
export_ip_record(&rec);
return;
}

```

## 4.4.5 开发示例：jflow发送函数

```

void export_ip_record(struct ip_record *rec)
{
 time_t now;
 /* fill out the header */
 now = time(NULL);
 rec->hdr.version = htons(1);
 rec->hdr.count = htons(1);
 rec->hdr.SysUptime = htonl(get_uptime());
 rec->hdr.unix_secs = htonl(now);
 rec->hdr.unix_nsecs = 0; /* XXX */
 if (write(ctx.u, rec, sizeof(struct ip_record))
 < sizeof(struct ip_record))
 warn("ip_export_record(): short write()");
 else ctx.count += 1;
 return;
}

```

## 4.4.5 开发示例：tcp回调函数

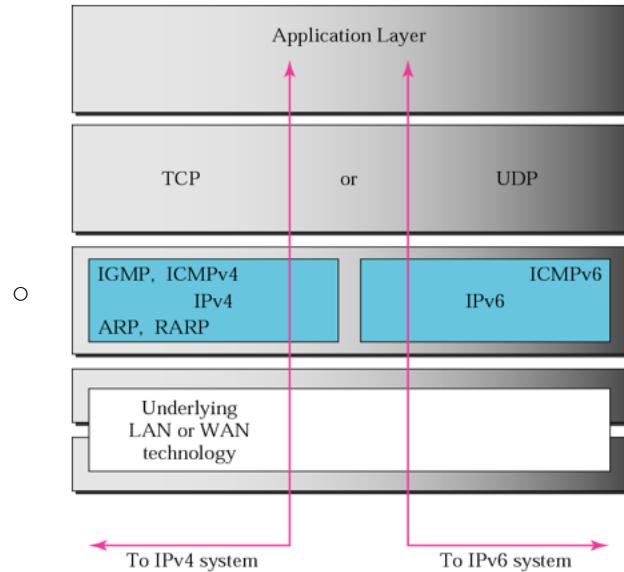
- 文件名称：tcp.c
- 功能描述：通过libnids还原tcp数据流
- 学习内容：
  - libnids库函数的基本函数及数据结构使用方法
  - Linux环境下编译方法、调试方法
  - tcp网络流解析方法

参考资料第62页

### 4.6 libnids升级换代

- 双协议栈检测方案

- 主机运行IPv4和IPv6两个协议栈。源节点发送分组前先查询DNS，若DNS返回IPv4地址，源节点就发送IPv4分组；若返回IPv6地址，就发送IPv6分组。



- 自动隧道检测技术

- 接收主机使用兼容的IPv6地址（运行双协议栈）
- 发送端使用IPv6兼容地址作为目的地址
- IPv6/IPv4边界路由器提取嵌入在IPv6地址中的IPv4地址进行数据包封装
- 目的主机接收IPv4数据包，取出IPv6分组交给IPv6协议软件处理。