

libpcap原理及使用

原创

ptmozhu

于 2017-12-07 16:43:30 发布


27343

收藏

87

版权

分类专栏: 网络 文章标签: pcap libpcap gopacket

 网络 专栏收录该内容

1 订阅 6 篇文章 订阅专栏

2.9 libpcap

本文最初整理在我的github上SDN-Learning-notes

libpcap（Packet Capture Library）即数据包捕获函数库，是Unix/Linux平台下的网络数据包捕获函数库。它是一个独立于系统的用户层包捕获的API接口，为底层网络监测提供了一个可移植的框架。著名的软件 **TCPDUMP** 就是在libpcap的的基础上开发而成的

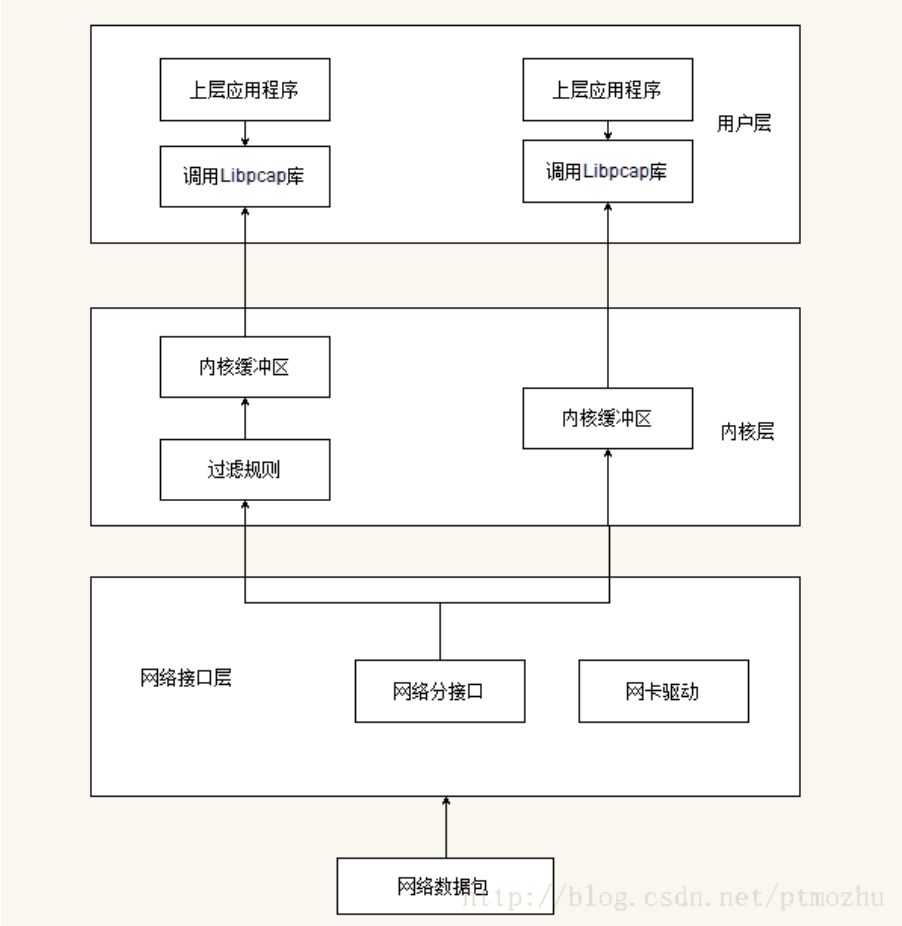
libpcap可以实现以下功能：

- 数据包捕获：捕获流经网卡的原始数据包
- 自定义数据包发送：任何构造格式的原始数据包
- 流量采集与统计：网络采集的中流量信息
- 规则过滤：提供自带规则过滤功能，按需要选择过滤规则

libpcap工作原理

libpcap主要由两部份组成：网络分接口(Network Tap)和数据过滤器(Packet Filter)。

网络分接口从网络设备驱动程序中收集数据拷贝（旁路机制），过滤器决定是否接收该数据包。Libpcap利用BSD Packet Filter(BPF)算法对网卡接收到的链路层数据包进行过滤。BPF算法的基本思想是在有BPF监听的网络中，网卡驱动将接收到的数据包复制一份交给BPF过滤器，过滤器根据用户定义的规则决定是否接收此数据包以及需要拷贝该数据包的那些内容，然后将过滤后的数据给与过滤器相关联的上层应用程序。如果没有定义规则，则把全部数据交给上层应用程序。



如图所示一个数据包的捕捉分为三个主要部分：

- 面向底层包捕获、
- 面向中间层的数据包过滤
- 面向应用层的用户接口

libpcap的包捕获机制就是在数据链路层加一个旁路处理。当一个数据包到达网络接口时，libpcap首先利用链路层PF_PACKET原始套接字从链路层驱动程序中获得该数据包的拷贝，再通过Tap函数将数据包发给BPF过滤器。BPF过滤器根据用户已经定义好的过滤规则对数据包进行逐一匹配，匹配成功则放入内核缓冲区，并传递给用户缓冲区，匹配失败则直接丢弃。如果没有设置过滤规则，所有数据包都将放入内核缓冲区，并传递给用户层缓冲区。所以整个过程并不干扰系统自身的网路协议栈的处理。

libpcap使用流程

1. 决定对那一个接口进行嗅探，如eth0。我们也可以用一串字符串来定义这个设备。
2. 初始化pcap。使用文件句柄传入需要嗅探的设备。同时支持多个设备的嗅探。
3. 设置BPF. 创建一个规则集合，编译并且使用它。这个过程分为三个阶段: 1.规则集合被置于一个字符串内，并且被转换成能被pcap读的格式。2.编译该规则（就是调用一个不被外部程序使用的函数）。3.告诉pcap使用它来过滤数据包。
4. pcap进入它的主循环。在这个阶段内pcap一直工作到它接收了所有我们想要的包为止。每当它收到一个包（或者多个数据包）就调用另一个已经定义好的函数，这个函数可以做我们想要的任何工作，比如它可以剖析包的上层协议信息并给用户打印出结果，它可以将结果保存为一个文件，或者什么也不作。
5. 在嗅探到所需的数据后，我们要关闭会话并结束。

tcpdump

tcpdump命令：

```
1 $ tcpdump --help
2 tcpdump: invalid option -- '-'
3
```

```

4  tcpdump version 4.5.1
5  libpcap version 1.5.3
6  Usage: tcpdump [-aAbdDefhHIJKlLnNOpqRStuUvxxX] [-B size] [-c count]
7      [-C file_size] [-E algo:secret] [-F file] [-G seconds]
8      [-i interface] [-j tstamptype] [-M secret]
9      [-P in|out|inout]
10     [-r file] [-s snaplen] [-T type] [-V file] [-w file]
      [-W filecount] [-y datalinktype] [-z command] [-Z user] [expression]

```

tcpdump选项可划分为四大类型：控制tcpdump程序行为，控制数据怎样显示，控制显示什么数据，以及过滤命令。

控制程序行为

这一类命令行选项影响tcpdump程序行为，如包括数据收集的方式：-w选项允许用户将输出重定向到一个文件，之后可通过-r选项将捕获数据显示出来。

如果用户知道需要捕获的报文数量或对于数量有一个上限，可使用-c选项。则当达到该数量时程序自动终止，而无需使用kill命令或Ctrl-C。下例中，收集到100个报文之后tcpdump终止：

```
bsd1# tcpdump -c 100
```

如果用户在多余一个网络接口上运行tcpdump，用户可以通过-i选项指定接口。在不确定的情况下，可使用ifconfig -a来检查哪一个接口可用及对应哪一个网络。例如，一台机器有两个接口，eth0接口IP地址192.168.0.1，eth1接口IP地址192.168.0.2。要捕捉192.168.0.1网络的数据流，使用以下命令：

```
1  bsd1# tcpdump -i eth0
```

没有指定接口时，tcpdump默认为最低编号接口。

-p 选项将网卡接口设置为非混杂模式。这一选项理论上将限制为捕获接口上的正常数据流——来自或发往主机，多播数据，以及广播数据。

-s 选项控制数据的截取长度。通常，tcpdump默认为一最大字节数量并只会从单一报文中截取到该数量长度。实际字节数取决于操作系统的设备驱动。通过默认值来截取合适的报文头，而舍弃不必要的报文数据。减少数据量。

如果用户需截取更多数据，通过-s选项来指定字节数。也可以用-s来减少截取字节数。对于少于或等于200字节的报文，以下命令会截取完整报文：

```
1  # tcpdump -s200
```

更长的报文会被缩短为200字节。

控制信息如何显示

-a，-n，-N 和 -f 选项决定了地址信息是如何显示的。-a选项强制将网络地址显示为名称，-n阻止将地址显示为名字，-N阻止将域名转换。-f选项阻止远端名称解析。下例中，从sloan.lander.edu (205.153.63.30) ing远程站点，分别不加选项，-a，-n，-N，-f。（选项-c1限制抓取1个报文）

```

1  # tcpdump -c1 host 192.31.7.130
2  tcpdump: listening on xl0
3  14:16:35.897342 sloan.lander.edu > cio-sys.cisco.com: icmp: echo request
4  # tcpdump -c1 -a host 192.31.7.130
5  tcpdump: listening on xl0
6  14:16:14.567917 sloan.lander.edu > cio-sys.cisco.com: icmp: echo request
7  # tcpdump -c1 -n host 192.31.7.130
8  tcpdump: listening on xl0
9  14:17:09.737597 205.153.63.30 > 192.31.7.130: icmp: echo request
10 # tcpdump -c1 -N host 192.31.7.130
11

```

```

11 tcpdump: listening on x10
12 14:17:28.891045 sloan > cio-sys: icmp: echo request
13 # tcpdump -c1 -f host 192.31.7.130
14 tcpdump: listening on x10
15 14:17:49.274907 sloan.lander.edu > 192.31.7.130: icmp: echo request

```

默认为-a选项。

-t 和 -tt 选项控制时间戳的打印。-t选项不显示时间戳而-tt选项显示无格式的时间戳。以下命令显示了tcpdump命令无选项，-t选项，-tt选项的同一报文：

```

1 12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)
2 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)
3 934303014.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)

```

控制显示什么数据

可以通过-v和-vv选项来打印更多详细信息。例如，-v选项将会打印TTL字段。要显示较少信息，使用-q，或quiet选项。一下为同一报文分别使用-q选项，无选项，-v选项，和-vv选项的输出。

```

1 12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: tcp 0 (DF)
2 12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)
3 12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF) (ttl 128, id 4583)
4 12:36:54.772066 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF) (ttl 128, id 4583)

```

-e 选项用于显示链路层头信息。上例中-e选项的输出为：

```

1 12:36:54.772066 0:10:5a:a1:e9:8 0:10:5a:e3:37:c ip 60:
2 sloan.lander.edu.1174 > 205.153.63.238.telnet: . ack 3259091394 win 8647 (DF)

```

0:10:5a:a1:e9:8是sloan.lander.edu中3Com卡的以太网地址，0:10:5a:e3:37:c是205.153.63.238中3Com卡的以太网地址。

-x 选项将报文以十六进制形式dump出来，排除了链路层报文头。-x和-vv选项报文显示如下：

```

1 13:57:12.719718 bsd1.lander.edu.1657 > 205.153.60.5.domain: 11587+ A? www.microsoft.com. (35) (ttl 64, id 41353)
2 4500 003f a189 0000 4011 c43a cd99 3db2
3 cd99 3c05 0679 0035 002b 06d9 2d43 0100
4 0001 0000 0000 0000 0377 7777 096d 6963
5 726f 736f 6674 0363 6f6d 0000 0100 01

```

过滤

要有效地使用tcpdump，掌握过滤器非常必要的。过滤允许用户指定想要抓取的数据流，从而用户可以专注于感兴趣的数据。

如果用户很清楚对何种数据流不感兴趣，可以将这部分数据排除在外。如果用户不确定需要什么数据，可以将源数据收集到文件之后在读取时应用过滤器。实际应用中，需要经常在两种方式之间转换。

简单的过滤器是加在命令行之后的关键字。但是，复杂的命令是由逻辑和关系运算符构成的。对于这样的情况，通常最好用-F选项将过滤器存储在文件中。例如，假设testfilter 是一个包含过滤主机205.153.63.30的文本文件，之后输入tcpdump -F testfilter等效于输入命令tcpdump host 205.153.63.30。通常，这一功能只在复杂过滤器时使用。但是，同一命令中命令行过滤器和文件过滤器不能混用。

地址过滤

过滤器可以按照地址选择数据流。例如，考虑如下命令：

```
1 # tcpdump host 205.153.63.30
```

该命令抓取所有来自以及发往IP地址205.153.63.30的主机。主机可以通过名称或IP地址来选定。虽然指定的是IP地址，但抓取数据流并不限于IP数据流，实际上，过滤器也会抓到ARP数据流。如果要限定仅抓取特定协议的数据流就需要更复杂的过滤器了。

有若干种方式可以指定和限制地址，下例是通过机器的以太网地址来选择数据流：

```
1 # tcpdump ether host 0:10:5a:e3:37:c
```

数据流可进一步限制为单向，分别用src或dst指定数据流的来源或目的地。下例显示了发送到主机205.153.63.30的数据流：

```
1 # tcpdump dst 205.153.63.30
```

注意到本例中host被省略了。在某些例子中省略是没问题的，但添加这些关键字通常更安全些。

广播和多播数据相应可以使用broadcast和multicast。由于多播和广播数据流在链路层和网络层所指定的数据流是不同的，所以这两种过滤器各有两种形式。过滤器 `ether multicast` 抓取以太网多播地址的数据流，`ip multicast` 抓取IP多播地址数据流。广播数据流也是类似的使用方法。

注意多播过滤器也会抓到广播数据流。

除了抓取特定主机以外，还可以抓取特定网络。例如，以下命令限制抓取来自或发往205.153.60.0的报文：

```
1 # tcpdump net 205.153.60
```

以下命令也可以做同样的事情：

```
1 # tcpdump net 205.153.60.0 mask 255.255.255.0
```

而以下命令由于最后的.0就无法正常工作：

```
1 # tcpdump net 205.153.60.0
```

协议及端口过滤

限制抓取指定协议如IP或TCP。还可以限制建立在这些协议之上的服务，如DNS或RIP。这类抓取可以通过三种方式进行：使用tcpdump关键字，通过协议关键字proto，或通过服务使用port关键字。

一些协议名能够被tcpdump识别到因此可通过关键字来指定。以下命令限制抓取IP数据流：

```
1 # tcpdump ip
```

当然，IP数据流包括TCP数据流，UDP数据流，等等。

如果仅抓取TCP数据流，可以使用：

```
1 # tcpdump tcp
```

tcpdump可识别的关键字包括ip, igmp, tcp, udp, and icmp。

有很多传输层服务没有可以识别的关键字。在这种情况下，可以使用关键字proto或ip proto加上/etc/protocols能够找到的协议名或相应的协议编号。例如，以下两种方式都会查找OSPF报文：

```
1  bsd1# tcpdump ip proto ospf
2  bsd1# tcpdump ip proto 89
```

内嵌的关键字可能会造成问题。下面的例子中，无法使用tcp关键字，或必须使用数字。例如，下面的例子是正常工作的：

```
1  bsd#1 tcpdump ip proto 6
```

另一方面，不能使用proto加上tcp：

```
1  bsd#1 tcpdump ip proto tcp
```

会产生问题。

对于更高层级的建立于底层协议之上的服务，必须使用关键字port。以下两者会采集DNS数据流：

```
1  #1 tcpdump port domain
2  #1 tcpdump port 53
```

第一条命令中，关键字domain能够通过查找/etc/services来解析。在传输层协议有歧义的情况下，可以将端口限制为指定协议。考虑如下命令：

```
1  #1 tcpdump udp port domain
```

这会抓取使用UDP的DNS名查找但不包括使用TCP的DNS zone传输数据。而之前的两条命令会同时抓取这两种数据。

报文特征

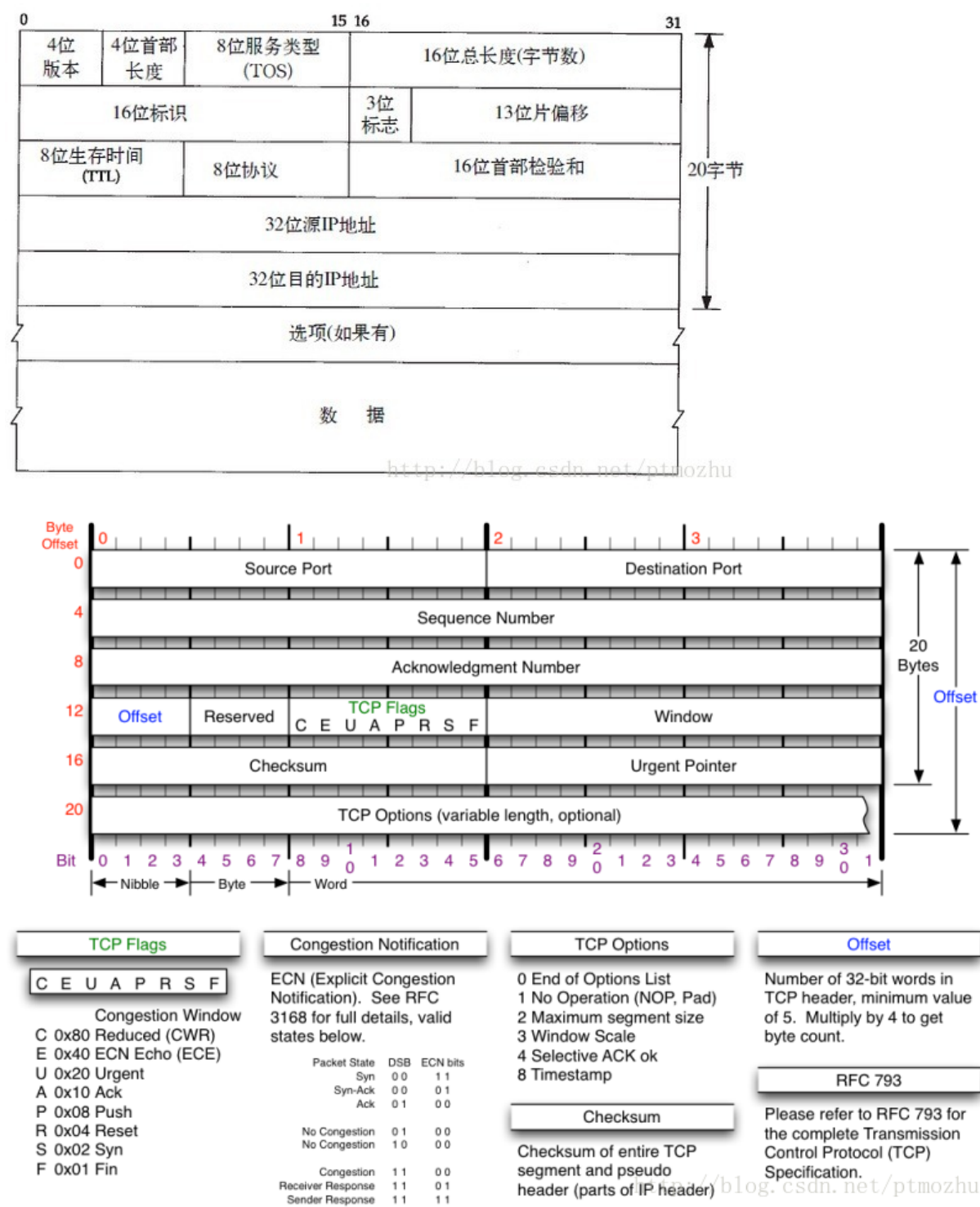
过滤器也可以基于报文特征比如报文长度或特定字段的内容，过滤器必须包含关系运算符。要指定长度，使用关键字less或greater。如下例所示：

```
1  # tcpdump greater 200
```

该命令收集长度大于200字节的报文。

根据报文内容过滤更加复杂，因为用户必须理解报文头的结构。但是尽管如此，或者说正因如此，这一方式能够使用户最大限度的控制抓取的数据。

我们先来回顾下ip,tcp的报文头：



根据协议报文头抓取数据的一般使用语法 `proto [expr : size]`。字段proto指定要查看的报文头——ip则查看IP头，tcp则查看TCP头，以此类推。expr字段给出从报文头索引0开始的位移。即：报文头的第一个字节为0，第二字节为1，以此类推。size字段是可选的，指定需要使用的字节数，1，2或4。

```
1 # tcpdump "ip[9] = 6"
```

查看第十字节的IP头，协议值为6。

注意这里必须使用引号。撇号或引号都可以，但反引号将无法正常工作。

```
1 # tcpdump tcp
```

也是等效的，因为TCP协议编号为6。

这一方式常常作为掩码来选择特定比特位。值可以是十六进制。可通过语法&加上比特掩码来指定。下例提取从以太网头第一字节开始（即目的地址第一字节），提取低阶比特位，并确保该位不为0：

```
1 # tcpdump 'ether[0] & 1 != 0'
```

该条件会选取广播和多播报文。

以上两个例子都有更好的方法来匹配报文。作为一个更实际的例子，考虑以下命令：

```
1 # tcpdump "tcp[13] & 0x03 != 0"
```

该过滤器跳过TCP头的13个字节，提取flag字节。掩码0x03选择第一和第二比特位，即FIN和SYN位。如果其中一位不为0则报文被抓取。此命令会抓取TCP连接建立及关闭报文。

不要将逻辑运算符与关系运算符混淆。比如想 `tcp src port > 23` 这样的表达式就无法正常工作。因为tcp src port表达式返回值为true或false，而不是一个数值，所以无法与数值进行比较。如果需要查找端口号大于23的所有TCP数据流，必须从报文头提取端口字段，使用表达式 `tcp[0:2] & 0xffff > 0x0017`。

gopacket包

使用go语言的gopacket包可以非常容易地实现抓包，gopacket包构建在libpcap的之上。下面介绍下gopacket的常见用法。

```
1 # Get the gopacket package from GitHub
2 go get github.com/google/gopacket
3 # Pcap dev headers might be necessary
4 sudo apt-get install libpcap-dev
```

1. 获取所有的网络设备信息

```
1 package main
2
3 import (
4     "fmt"
5     "log"
6     "github.com/google/gopacket/pcap"
7 )
8
9 func main() {
10     // Find all devices
11     devices, err := pcap.FindAllDevs()
12     if err != nil {
13         log.Fatal(err)
14     }
15
16     // Print device information
17     fmt.Println("Devices found:")
18     for _, device := range devices {
19         fmt.Println("\nName: ", device.Name)
20         fmt.Println("Description: ", device.Description)
21         fmt.Println("Devices addresses: ", device.Addresses)
22         for _, address := range device.Addresses {
23             fmt.Println("- IP address: ", address.IP)
24             fmt.Println("- Subnet mask: ", address.Netmask)
25         }
26     }
27 }
```


1. 打开设备实时捕捉

```

1  package main
2
3  import (
4      "fmt"
5      "github.com/google/gopacket"
6      "github.com/google/gopacket/pcap"
7      "log"
8      "time"
9  )
10
11  var (
12      device      string = "eth0"
13      snapshot_len int32  = 1024
14      promiscuous bool   = false
15      err          error
16      timeout      time.Duration = 30 * time.Second
17      handle       *pcap.Handle
18  )
19
20  func main() {
21      // Open device
22      handle, err = pcap.OpenLive(device, snapshot_len, promiscuous, timeout)
23      if err != nil {log.Fatal(err) }
24      defer handle.Close()
25
26      // Use the handle as a packet source to process all packets
27      packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
28      for packet := range packetSource.Packets() {
29          // Process packet here
30          fmt.Println(packet)
31      }
32  }

```

1. 抓取结果保存为PCAP格式文件

我们必须使用gapacket/pcapgo包来输出pcap的格式文件。

```

1  package main
2
3  import (
4      "fmt"
5      "os"
6      "time"
7
8      "github.com/google/gopacket"
9      "github.com/google/gopacket/layers"
10     "github.com/google/gopacket/pcap"
11     "github.com/google/gopacket/pcapgo"
12 )
13
14  var (
15      deviceName string = "eth0"
16      snapshotLen int32  = 1024
17      promiscuous bool   = false
18      err          error
19      timeout      time.Duration = -1 * time.Second
20      handle       *pcap.Handle
21      packetCount int = 0
22  )

```

```

22 )
23
24 func main() {
25     // Open output pcap file and write header
26     f, _ := os.Create("test.pcap")
27     w := pcapgo.NewWriter(f)
28     w.WriteHeader(snapshotLen, layers.LinkTypeEthernet)
29     defer f.Close()
30
31     // Open the device for capturing
32     handle, err = pcap.OpenLive(deviceName, snapshotLen, promiscuous, timeout)
33     if err != nil {
34         fmt.Printf("Error opening device %s: %v", deviceName, err)
35         os.Exit(1)
36     }
37     defer handle.Close()
38
39     // Start processing packets
40     packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
41     for packet := range packetSource.Packets() {
42         // Process packet here
43         fmt.Println(packet)
44         w.WritePacket(packet.Metadata().CaptureInfo, packet.Data())
45         packetCount++
46
47         // Only capture 100 and then stop
48         if packetCount > 100 {
49             break
50         }
51     }
52 }

```

1. 读取PCAP格式文件来查看分析网络数据包

我们可以使用tcpdump创建的文件。

```

1  # Capture packets to test.pcap file
2  sudo tcpdump -w test.pcap
3  package main
4
5  // Use tcpdump to create a test file
6  // tcpdump -w test.pcap
7  // or use the example above for writing pcap files
8
9  import (
10     "fmt"
11     "github.com/google/gopacket"
12     "github.com/google/gopacket/pcap"
13     "log"
14 )
15
16 var (
17     pcapFile string = "test.pcap"
18     handle    *pcap.Handle
19     err       error
20 )
21
22 func main() {
23     // Open file instead of device
24

```

```

25     handle, err = pcap.OpenOffline(pcapFile)
26     if err != nil { log.Fatal(err) }
27     defer handle.Close()
28
29     // Loop through packets in file
30     packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
31     for packet := range packetSource.Packets() {
32         fmt.Println(packet)
33     }
34 }

```

1. 设置过滤器

```

1  #只抓取TCP协议80端口的数据
2  package main
3
4  import (
5      "fmt"
6      "github.com/google/gopacket"
7      "github.com/google/gopacket/pcap"
8      "log"
9      "time"
10 )
11
12 var (
13     device      string = "eth0"
14     snapshot_len int32  = 1024
15     promiscuous bool   = false
16     err          error
17     timeout      time.Duration = 30 * time.Second
18     handle       *pcap.Handle
19 )
20
21 func main() {
22     // Open device
23     handle, err = pcap.OpenLive(device, snapshot_len, promiscuous, timeout)
24     if err != nil {
25         log.Fatal(err)
26     }
27     defer handle.Close()
28
29     // Set filter
30     var filter string = "tcp and port 80"
31     err = handle.SetBPFFilter(filter)
32     if err != nil {
33         log.Fatal(err)
34     }
35     fmt.Println("Only capturing TCP port 80 packets.")
36
37     packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
38     for packet := range packetSource.Packets() {
39         // Do something with a packet here.
40         fmt.Println(packet)
41     }
42 }
43 }

```

1. 解码抓取的数据

我们可以使用原始数据包，并且可将其转换为已知格式。它与不同的层兼容，所以我们可以轻松访问以太网，IP和TCP层。layers包是进入库中新增加的，在底层PCAP库中不可用。这是一个令人难以置信的有用的包，它是gopacket库的一部分。它允许我们容易地识别包是否包含特定类型的层。该代码示例将显示如何使用层包来查看数据包是以太网，IP和TCP，并轻松访问这些头文件中的元素。查询有效载荷取决于所涉及的所有层。每个协议是不同的，必须相应地计算。这就是layer包的魅力所在。gopacket的作者花了时间为诸如以太网，IP，UDP和TCP等众多已知层创建了相应类型。有效载荷是应用层的一部分。

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/google/gopacket"
6     "github.com/google/gopacket/layers"
7     "github.com/google/gopacket/pcap"
8     "log"
9     "strings"
10    "time"
11 )
12
13 var (
14     device      string = "eth0"
15     snapshotLen int32  = 1024
16     promiscuous bool   = false
17     err          error
18     timeout      time.Duration = 30 * time.Second
19     handle       *pcap.Handle
20 )
21
22 func main() {
23     // Open device
24     handle, err = pcap.OpenLive(device, snapshotLen, promiscuous, timeout)
25     if err != nil {log.Fatal(err) }
26     defer handle.Close()
27
28     packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
29     for packet := range packetSource.Packets() {
30         printPacketInfo(packet)
31     }
32 }
33
34 func printPacketInfo(packet gopacket.Packet) {
35     // Let's see if the packet is an ethernet packet
36     ethernetLayer := packet.Layer(layers.LayerTypeEthernet)
37     if ethernetLayer != nil {
38         fmt.Println("Ethernet layer detected.")
39         ethernetPacket, _ := ethernetLayer.(*layers.Ethernet)
40         fmt.Println("Source MAC: ", ethernetPacket.SrcMAC)
41         fmt.Println("Destination MAC: ", ethernetPacket.DstMAC)
42         // Ethernet type is typically IPv4 but could be ARP or other
43         fmt.Println("Ethernet type: ", ethernetPacket.EthernetType)
44         fmt.Println()
45     }
46
47     // Let's see if the packet is IP (even though the ether type told us)
48     ipLayer := packet.Layer(layers.LayerTypeIPv4)
49     if ipLayer != nil {
50         fmt.Println("IPv4 layer detected.")
51         ip, _ := ipLayer.(*layers.IPv4)
52
53         // IP layer variables:
54         // Version (Either 4 or 6)
55         // IHL (IP Header Length in 32-bit words)
56     }
```

```

57      // TOS, Length, Id, Flags, FragOffset, TTL, Protocol (TCP?),
58      // Checksum, SrcIP, DstIP
59      fmt.Printf("From %s to %s\n", ip.SrcIP, ip.DstIP)
60      fmt.Println("Protocol: ", ip.Protocol)
61      fmt.Println()
62  }
63
64      // Let's see if the packet is TCP
65      tcpLayer := packet.Layer(layers.LayerTypeTCP)
66      if tcpLayer != nil {
67          fmt.Println("TCP layer detected.")
68          tcp, _ := tcpLayer.(*layers.TCP)
69
70          // TCP Layer variables:
71          // SrcPort, DstPort, Seq, Ack, DataOffset, Window, Checksum, Urgent
72          // Bool flags: FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, NS
73          fmt.Printf("From port %d to %d\n", tcp.SrcPort, tcp.DstPort)
74          fmt.Println("Sequence number: ", tcp.Seq)
75          fmt.Println()
76      }
77
78      // Iterate over all layers, printing out each layer type
79      fmt.Println("All packet layers:")
80      for _, layer := range packet.Layers() {
81          fmt.Println("- ", layer.LayerType())
82      }
83
84      // When iterating through packet.Layers() above,
85      // if it lists Payload Layer then that is the same as
86      // this applicationLayer. applicationLayer contains the payload
87      applicationLayer := packet.ApplicationLayer()
88      if applicationLayer != nil {
89          fmt.Println("Application layer/Payload found.")
90          fmt.Printf("%s\n", applicationLayer.Payload())
91
92          // Search for a string inside the payload
93          if strings.Contains(string(applicationLayer.Payload()), "HTTP") {
94              fmt.Println("HTTP found!")
95          }
96      }
97
98      // Check for errors
99      if err := packet.ErrorLayer(); err != nil {
100          fmt.Println("Error decoding some part of the packet:", err)
101      }
102  }

```

1. 构造发送数据包

这个例子做了几件事情。首先将显示如何使用网络设备发送原始字节。这样就可以像串行连接一样使用它来发送数据。这对于真正的低层数据传输非常有用，但如果您想与应用程序进行交互，您应该构建可以识别该数据包的其他硬件和软件。接下来，它将显示如何使用以太网，IP和TCP层创建一个数据包。一切都是默认空的。要完成它，我们创建另一个数据包，但实际上填写了以太网层的一些MAC地址，IPv4的一些IP地址和TCP层的端口号。你应该看到如何伪装数据包和仿冒网络设备。TCP层结构体具有可读取和可设置的SYN，FIN，ACK标志。这有助于操纵和模糊TCP三次握手，会话和端口扫描。pcap库提供了一种发送字节的简单方法，但gopacket中的图元可帮助我们为多层创建字节结构。

```

1  package main
2
3  import (
4      "github.com/google/gopacket"
5

```

```
-
6      "github.com/google/gopacket/layers"
7      "github.com/google/gopacket/pcap"
8      "log"
9      "net"
10     "time"
11 )
12
13 var (
14     device      string = "eth0"
15     snapshot_len int32  = 1024
16     promiscuous bool   = false
17     err          error
18     timeout      time.Duration = 30 * time.Second
19     handle       *pcap.Handle
20     buffer       gopacket.SerializeBuffer
21     options      gopacket.SerializeOptions
22 )
23
24 func main() {
25     // Open device
26     handle, err = pcap.OpenLive(device, snapshot_len, promiscuous, timeout)
27     if err != nil {log.Fatal(err)}
28     defer handle.Close()
29
30     // Send raw bytes over wire
31     rawBytes := []byte{10, 20, 30}
32     err = handle.WritePacketData(rawBytes)
33     if err != nil {
34         log.Fatal(err)
35     }
36
37     // Create a properly formed packet, just with
38     // empty details. Should fill out MAC addresses,
39     // IP addresses, etc.
40     buffer = gopacket.NewSerializeBuffer()
41     gopacket.SerializeLayers(buffer, options,
42         &layers.Ethernet{},
43         &layers.IPv4{},
44         &layers.TCP{},
45         gopacket.Payload(rawBytes),
46     )
47     outgoingPacket := buffer.Bytes()
48     // Send our packet
49     err = handle.WritePacketData(outgoingPacket)
50     if err != nil {
51         log.Fatal(err)
52     }
53
54     // This time Lets fill out some information
55     ipLayer := &layers.IPv4{
56         SrcIP: net.IP{127, 0, 0, 1},
57         DstIP: net.IP{8, 8, 8, 8},
58     }
59     ethernetLayer := &layers.Ethernet{
60         SrcMAC: net.HardwareAddr{0xFF, 0xAA, 0xFA, 0xAA, 0xFF, 0xAA},
61         DstMAC: net.HardwareAddr{0xBD, 0xBD, 0xBD, 0xBD, 0xBD, 0xBD},
62     }
63     tcpLayer := &layers.TCP{
64         SrcPort: layers.TCPPort(4321),
65         DstPort: layers.TCPPort(80),
66     }
67     // And create the packet with the layers
68     buffer = gopacket.NewSerializeBuffer()
```

```

69     gopacket.SerializeLayers(buffer, options,
70         ethernetLayer,
71         ipLayer,
72         tcpLayer,
73         gopacket.Payload(rawBytes),
74     )
75     outgoingPacket = buffer.Bytes()
}

```

1. 更多的解码/构造数据包的例子

```

1  package main
2
3  import (
4      "fmt"
5      "github.com/google/gopacket"
6      "github.com/google/gopacket/layers"
7  )
8
9  func main() {
10     // If we don't have a handle to a device or a file, but we have a bunch
11     // of raw bytes, we can try to decode them in to packet information
12
13     // NewPacket() takes the raw bytes that make up the packet as the first parameter
14     // The second parameter is the lowest level layer you want to decode. It will
15     // decode that layer and all layers on top of it. The third layer
16     // is the type of decoding: default(all at once), lazy(on demand), and NoCopy
17     // which will not create a copy of the buffer
18
19     // Create an packet with ethernet, IP, TCP, and payload layers
20     // We are creating one we know will be decoded properly but
21     // your byte source could be anything. If any of the packets
22     // come back as nil, that means it could not decode it in to
23     // the proper layer (malformed or incorrect packet type)
24     payload := []byte{2, 4, 6}
25     options := gopacket.SerializeOptions{}
26     buffer := gopacket.NewSerializeBuffer()
27     gopacket.SerializeLayers(buffer, options,
28         &layers.Ethernet{},
29         &layers.IPv4{},
30         &layers.TCP{},
31         gopacket.Payload(payload),
32     )
33     rawBytes := buffer.Bytes()
34
35     // Decode an ethernet packet
36     ethPacket :=
37         gopacket.NewPacket(
38             rawBytes,
39             layers.LayerTypeEthernet,
40             gopacket.Default,
41         )
42
43     // with Lazy decoding it will only decode what it needs when it needs it
44     // This is not concurrency safe. If using concurrency, use default
45     ipPacket :=
46         gopacket.NewPacket(
47             rawBytes,
48             layers.LayerTypeIPv4,
49             gopacket.Lazy,
50         )
51 }

```

```

51
52     // With the NoCopy option, the underlying slices are referenced
53     // directly and not copied. If the underlying bytes change so will
54     // the packet
55     tcpPacket :=
56         gopacket.NewPacket(
57             rawBytes,
58             layers.LayerTypeTCP,
59             gopacket.NoCopy,
60         )
61
62     fmt.Println(ethPacket)
63     fmt.Println(ipPacket)
64     fmt.Println(tcpPacket)
65 }

```

1. 自定义协议

使用gopacket layer包不包含的协议。比如如果您要创建自己的I33t协议，甚至不使用TCP/IP或以太网，这是很有用的。

```

1  package main
2
3  import (
4      "fmt"
5      "github.com/google/gopacket"
6  )
7
8  // Create custom layer structure
9  type CustomLayer struct {
10     // This layer just has two bytes at the front
11     SomeByte    byte
12     AnotherByte byte
13     restOfData  []byte
14 }
15
16 // Register the Layer type so we can use it
17 // The first argument is an ID. Use negative
18 // or 2000+ for custom layers. It must be unique
19 var CustomLayerType = gopacket.RegisterLayerType(
20     2001,
21     gopacket.LayerTypeMetadata{
22         "CustomLayerType",
23         gopacket.DecodeFunc(decodeCustomLayer),
24     },
25 )
26
27 // When we inquire about the type, what type of layer should
28 // we say it is? We want it to return our custom layer type
29 func (l CustomLayer) LayerType() gopacket.LayerType {
30     return CustomLayerType
31 }
32
33 // LayerContents returns the information that our layer
34 // provides. In this case it is a header layer so
35 // we return the header information
36 func (l CustomLayer) LayerContents() []byte {
37     return []byte{l.SomeByte, l.AnotherByte}
38 }
39
40 // LayerPayload returns the subsequent layer built
41

```



```

42 // on top of our layer or raw payload
43 func (l CustomLayer) LayerPayload() []byte {
44     return l.restOfData
45 }
46
47 // Custom decode function. We can name it whatever we want
48 // but it should have the same arguments and return value
49 // When the Layer is registered we tell it to use this decode function
50 func decodeCustomLayer(data []byte, p gopacket.PacketBuilder) error {
51     // AddLayer appends to the list of layers that the packet has
52     p.AddLayer(&CustomLayer{data[0], data[1], data[2:]})
53
54     // The return value tells the packet what layer to expect
55     // with the rest of the data. It could be another header layer,
56     // nothing, or a payload layer.
57
58     // nil means this is the last layer. No more decoding
59     // return nil
60
61     // Returning another layer type tells it to decode
62     // the next layer with that layer's decoder function
63     // return p.NextDecoder(layers.LayerTypeEthernet)
64
65     // Returning payload type means the rest of the data
66     // is raw payload. It will set the application layer
67     // contents with the payload
68     return p.NextDecoder(gopacket.LayerTypePayload)
69 }
70
71 func main() {
72     // If you create your own encoding and decoding you can essentially
73     // create your own protocol or implement a protocol that is not
74     // already defined in the layers package. In our example we are just
75     // wrapping a normal ethernet packet with our own layer.
76     // Creating your own protocol is good if you want to create
77     // some obfuscated binary data type that was difficult for others
78     // to decode
79
80     // Finally, decode your packets:
81     rawBytes := []byte{0xF0, 0x0F, 65, 65, 66, 67, 68}
82     packet := gopacket.NewPacket(
83         rawBytes,
84         CustomLayerType,
85         gopacket.Default,
86     )
87     fmt.Println("Created packet out of raw bytes.")
88     fmt.Println(packet)
89
90     // Decode the packet as our custom layer
91     customLayer := packet.Layer(CustomLayerType)
92     if customLayer != nil {
93         fmt.Println("Packet was successfully decoded with custom layer decoder.")
94         customLayerContent, _ := customLayer.(*CustomLayer)
95         // Now we can access the elements of the custom struct
96         fmt.Println("Payload: ", customLayerContent.LayerPayload())
97         fmt.Println("SomeByte element:", customLayerContent.SomeByte)
98         fmt.Println("AnotherByte element:", customLayerContent.AnotherByte)
99     }
100 }

```

1. 更快地解码数据包

如果我们知道我们要预期的得到的层，我们可以使用现有的结构来存储分组信息，而不是为每个需要时间和内存的分组创建新的结构。使用DecodingLayerParser更快。就像编组和解组数据一样。

```
1  package main
2
3  import (
4      "fmt"
5      "github.com/google/gopacket"
6      "github.com/google/gopacket/layers"
7      "github.com/google/gopacket/pcap"
8      "log"
9      "time"
10 )
11
12 var (
13     device      string = "eth0"
14     snapshot_len int32  = 1024
15     promiscuous bool   = false
16     err          error
17     timeout      time.Duration = 30 * time.Second
18     handle       *pcap.Handle
19     // Will reuse these for each packet
20     ethLayer layers.Ethernet
21     ipLayer  layers.IPv4
22     tcpLayer layers.TCP
23 )
24
25 func main() {
26     // Open device
27     handle, err = pcap.OpenLive(device, snapshot_len, promiscuous, timeout)
28     if err != nil {
29         log.Fatal(err)
30     }
31     defer handle.Close()
32
33     packetSource := gopacket.NewPacketSource(handle, handle.LinkType())
34     for packet := range packetSource.Packets() {
35         parser := gopacket.NewDecodingLayerParser(
36             layers.LayerTypeEthernet,
37             &ethLayer,
38             &ipLayer,
39             &tcpLayer,
40         )
41         foundLayerTypes := []gopacket.LayerType{}
42
43         err := parser.DecodeLayers(packet.Data(), &foundLayerTypes)
44         if err != nil {
45             fmt.Println("Trouble decoding layers: ", err)
46         }
47
48         for _, layerType := range foundLayerTypes {
49             if layerType == layers.LayerTypeIPv4 {
50                 fmt.Println("IPv4: ", ipLayer.SrcIP, "->", ipLayer.DstIP)
51             }
52             if layerType == layers.LayerTypeTCP {
53                 fmt.Println("TCP Port: ", tcpLayer.SrcPort, "->", tcpLayer.DstPort)
54                 fmt.Println("TCP SYN:", tcpLayer.SYN, " | ACK:", tcpLayer.ACK)
55             }
56         }
57     }
58 }
```

```
}  
}
```

参考文档

- [1] <http://www.tcpdump.org/pcap.html>
- [2] <http://www.jianshu.com/p/ed6db49a3428>
- [3] <https://feisky.gitbooks.io/sdn/linux/tcpdump.html>