# Snort安装

使用系统CentOS Linux release 7.0.1406 (Core)

安装依赖的软件包

rpm -qa | grep 软件包名(例如dhcp) <== 列出所有被安装的rpm package

1. **flex (**2.5.35) 未安装

下载了flex-2.5.37-3.el7.x86\_64.rpm

flex-2.5.37-3.el7.x86\_64.rpm依赖m4

下载了m4-1.4.16-10.el7.x86\_64.rpm

1. **bison** (2.4.1)未安装

bison-2.7-4.el7.x86\_64.rpm

1. **libdnet** (1.11 or 1.12 including **libdnet-devel**)

libdnet-1.12-13.1.el7.x86\_64.rpm

libdnet-devel-1.12-13.1.el7.x86\_64.rpm

1. **zlib** (1.2.7 including **zlib-devel**)

**zlib-devel**未安装

zlib-devel-1.2.7-13.el7.x86\_64.rpm 对应于zlib-1.2.7-13.el7.x86\_64

1. **libpcap** (1.5.3 including **libpcap-devel**)

升级了libpcap libpcap-1.5.3-8.el7.x86\_64.rpm

安装 libpcap-devel-1.5.3-8.el7.x86\_64.rpm

1. **pcre** (8.32 including **pcre-devel**)

升级了pcre pcre-8.32-15.el7.x86\_64.rpm

安装了 pcre-devel-8.32-15.el7.x86\_64.rpm

安装指令 rpm –ivh ……..

升级：rpm -Uv libpcap-1.5.3-8.el7.x86\_64.rpm

为了修改preprocessor里的makefile还需安装automake

不修改的话可以不安装

cd /usr/local/src <enter>  
tar -zxvf <path to>libdnet-1.xx.tar.gz <enter>（这个不需要）  
tar -zxvf <path to>daq-2.0.x.tar.gz <enter>  
tar -zxvf <path to>snort-2.9.7.x.tar.gz <enter>

Snort解压的路径 /usr/local/src/snort-2.9.8.2 之后configure 和 make （带有参数，具体参看文档）

DAQ解压的路径 /usr/local/src/daq-2.0.6

cd /usr/local/src/daq-2.0.x <enter>  
./configure <enter>  
make <enter>  
make install <enter>

cd /usr/local/src/snort-2.9.7.x <enter>  
./configure --enable-sourcefire <enter>  
make <enter>  
make install <enter>

/usr/local/src/snort-2.9.8.2/etc 路径下内容cp至 /etc/snort/

snortrules-snapshot的文件（规则文件）解压到/etc/snort/路径下

/etc/snort/etc内文件cp至/etc/snort/

cd /etc <enter>  
mkdir -p snort <enter>  
cd snort <enter>  
cp /usr/local/src/snort-2.9.7.x/etc/\* . <enter>  
tar -zvxf <path to>snortrules-snapshot-<nnnn>.tar.gz <enter>  
cp ./etc/\* . <enter>  
touch /etc/snort/rules/white\_list.rules /etc/snort/rules/black\_list.rules <enter>

------------------------------------阿里云上的snort就安装到了这里------------------------

**由于bridge网卡配置的是dhcp，所以在snort.conf里需要经常更换保护的主机IP地址**

Locate and modify the following variables in your snort.conf file

(in directory /etc/snort) as follows (usually between lines 40 and 120):

This assumes the network you are going to monitor is 192.168.1.0/24

var RULE\_PATH /etc/snort/rules

ipvar HOME\_NET 192.168.1.0/24

ipvar EXTERNAL\_NET !$HOME\_NET

var SO\_RULE\_PATH /etc/snort/so\_rules

var PREPROC\_RULE\_PATH /etc/snort/preproc\_rules

var WHITE\_LIST\_PATH /etc/snort/rules

var BLACK\_LIST\_PATH /etc/snort/rules

暂时用的是 10.108.113.153/22 22代表子网掩码255.255.252.0



这步骤暂时没做

The file below should be named 'snort' and placed into the /etc/sysconfig directory on

your CentOS 6.x/7.x system:

----- CUT HERE -----

# /etc/sysconfig/snort

# $Id: snort.sysconfig,v 1.8 2003/09/19 05:18:12 dwittenb Exp $

#### General Configuration

INTERFACE=eth0

CONF=/etc/snort/snort.conf

USER=snort

GROUP=snort

PASS\_FIRST=0

#### Logging & Alerting

LOGDIR=/var/log/snort

ALERTMODE=fast

DUMP\_APP=1

BINARY\_LOG=1

NO\_PACKET\_LOG=0

PRINT\_INTERFACE=0

----- CUT HERE -----

之后执行了

chown -R snort:snort snort

chown -R 700 snort

所有配置完成后试运行，报错

ERROR: snort.conf(253) Could not stat dynamic module path "/usr/local/lib/snort\_dynamicrules": No such file or directory. Fatal Error, Quitting.

在常见错误里，第一条即有解决方式

Solution below:  
mkdir -p /usr/local/lib/snort\_dynamicrules <enter>  
chown -R snort:snort /usr/local/lib/snort\_dynamicrules <enter>  
chmod -R 700 /usr/local/lib/snort\_dynamicrules <enter> Copy any dynamic rulesets you have or are using to the above directory. Another method would be to comment out that line in snort.conf if you have no dynamic rules in use

# Snort学习

snort有三种工作模式：嗅探器、数据包记录器、网络入侵检测系统。

嗅探器模式仅仅是从网络上读取数据包并作为连续不断的流显示在终端上。

数据包记录器 模式把数据包记录到硬盘上。

网路入侵检测模式是最复杂的，而且是可配置的。

## Snort Overview

### 嗅探器

snort -v

只把IP TCP/UDP/ICMP包头信息打印在屏幕上。

snort -vd

显示传输的应用数据信息

snort -vde

显示链路层的包头

如图 这是一个HTTP的数据包

snort -v则只显示红框中的内容

snort -vd显示下图内容



snort -vde显示下图内容

可见绿框中的内容为新添的内容，为数据链路层的包头



### 数据包记录器

snort -dev -l ./log

如果要把所有的包记录到硬盘上，你需要指定一个日志目录，snort就会自动记录数据包

snort -dev -l ./log/ -h 10.0.7.118/8

这个命令告诉snort把进入网络10.的所有包的数据链路、TCP/IP以及应用层的数据记录到目录./log中

snort -l ./log -b

如果你的网络速度很快，或者你想使日志更加紧凑以便以后的分析，那么应该使用二进制的日志文件格式。所谓的二进制日志文件格式就是tcpdump程序使用的格式。使用下面的命令可以把所有的包记录到一个单一的二进制文件中：

注意此处的命令行和上面的有很大的不同。我们勿需指定本地网络，因为所有的东西都被记录到一个单一的文件。你也不必冗余模式或者使用-d、-e功能选项，因为数据包中的所有内容都会被记录到日志文件中。

snort -dv -r packet.log

使用-r功能开关，也能使snort读出包的数据。snort在所有运行模式下都能够处理tcpdump格式的文件。例如：如果你想在嗅探器模式下把一个tcpdump格式的二进制文件中的包打印到屏幕上，可以输入以上的命令

snort -dvr packet.log icmp

在日志包和入侵检测模式下，通过BPF(BSD Packet Filter)接口，可以使用许多方式维护日志文件中的数据。上面的命令行只读取icmp包

### 网络入侵检测系统

snort -dev -l ./log/ -h 10.0.7.118/8 -c /etc/snort/snort.conf

-h 10.0.7.118/8 指定主机 会覆盖conf文件中对主机的配置

但是目前总是得不到日志文件，不知是否是因为没有报警信息

#### 网络入侵检测模式下的输出选项

在默认情况下，snort以ASCII格式记录日志，使用full报警机制。如果使用full报警机制，snort会在包头之后打印报警消息。

snort有7种报警机制：full, fast,socket, syslog, console, cmg, and none。其中有6个可以在命令行状态下使用-A选项设置。这6个是：



用-s选项可以使snort把报警消息发送到syslog，默认的设备是LOG\_AUTHPRIV和LOG\_ALERT。可以修改snort.conf文件修改其配置。例如：

snort -c snort.conf -l ./log -h 192.168.1.0/24 -s

#### 理解报警输出

报警信息形如：

[\*\*] [116:56:1] (snort\_decoder): T/TCP Detected [\*\*]

116 是Generator ID，代表Snort产生报警的部件。etc/generators 可查看GIDs的列表

In this case, we know that this event came from the “decode”(116) component of Snort.

56是SnortID，For a list of preprocessor SIDs, please see etc/gen-msg.map。Rule-based SIDs are written directly into the rules with the sid option. In this case, 56 represents a T/TCP event.

1是版本ID

#### 高性能配置

如果想使Snort做够快(1000Mbps)，需要使用unified2 logging，unified2 logging的读取器是barnyard。这允许Snort使用二进制格式记录alert，这个比写到数据库中要快。

如果想写到一个text文件中，而且还需要尽可能地快，尝试使用binary logging和”fast”输出选项。

./snort -b -A fast -c snort.conf

这个命令将用tcpdump格式记录数据包并产生最少的报警

#### 改变报警顺序

### 包捕获

Snort使用DAQ Data Acquisition library，去捕获IO信号，而不是直接调用libpcap，这样可以保证跨平台时不用修改Snort。

当想使用Snort pcap readback或者inline模式时，DAQ可以选择模式



ethtool -k eth0查看当前状态

ethtool -K eth0 gro off 关闭



#### 配置

./snort \  
[--daq <type>] \  
[--daq-mode <mode>] \  
[--daq-dir <dir>] \  
[--daq-var <var>]  
config daq: <type>  
config daq\_dir: <dir>  
config daq\_var: <var>  
config daq\_mode: <mode>  
<type> ::= pcap | afpacket | dump | nfq | ipq | ipfw  
<mode> ::= read-file | passive | inline  
<var> ::= arbitrary <name>=<value> passed to DAQ  
<dir> ::= path where to look for DAQ module so’s

<type> ::= pcap | afpacket | dump | nfq | ipq | ipfw

例如snort -r select\_tcp.pcap --daq dump -c /etc/snort/snort.conf

使用dump格式，默认是pcap，暂时没看出来区别

pcap是默认模式，Snort可以运行在与tcpdump模式类似的sniffer模式，他可以记录包。

在特别繁忙的网络环境下sniffer模式不是很实用，因为把输出打到console屏幕上很快，不易阅读。

解决办法是使用filter，只看需要的包。

假设你连接在SPAN switch port or a network tap

使用snort –v –i eth0 host 172.16.100.53 而不是使用snort -v -i eth0

假设你担心感染上了网络蠕虫，想观察有哪些别的主机正在连接

snort –v –i eth0 src 172.16.100.53 and ‘tcp[13] & 2!=0 to display only SYN packets

短时间内，单一主机向多个主机发送大量SYN包预示着可能有病毒

snort在2.9版本加入了daq，同时引入了afpacket。afpacket是linux2.6开始引入的报文获取接口。其最大的特点是打破以往复制报文传递给用户的方式，而是采用了共享内存的交互方式。并且利用环形缓冲区使用户可以无锁处理。

分析

snort使用afpacket可以实现inline模式， 即IPS，不同于IDS的被动防御模式, IPS可以主动阻断。

snort首先会将配置的接口两两配对,这里以ETH0和ETH1为例.

然后使用afpacket的相关接口（具体可在内核中查找关键字）为每个接口分别建立两个环形缓冲区,

RX为输入缓冲区， TX为输出缓冲区。缓冲区是一块内核和用户共享的内存空间。

缓冲区的处理原理是循环队列，以输入缓冲区为例; 当内核发现RX有空位时就将报文放入空位中并打上可用标志，用户发现有标注可用的单元时就可以对其处理，使用完成后为该单元标注空闲标志内核又可继续利用。因为单消费者和生产者情况所以该队列能够无锁处理。

从ETH0进入的报文如红线所示，内核将报文放入RX中，snort获取该报文并进行匹配后返回通过和丢弃两种结果。如果是通过，则将该数据拷贝进ETH1的发送缓冲区TX并将在ETH0的RX中的该单元标志位空闲，否则不进行拷贝直接将ETH0的RX中的该单元标志为空闲。



### Reading Pcap file



### Basic Output

### Tunneling Protocol Support

### 杂项

#### 2.1.8.5 Snort模式

三种

1. Inline

IPS ，允许触发drop规则 ，命令行下使用-Q代表Inline模式

conf文件中 config policy\_mode:inline

1. Passive

IDS，不触发drop规则（-treat-drop-as-alert除外）

conf文件中 config policy\_mode:tap

1. Inline-Test

与Inline模式相似，允许在不影响数据流的情况下评估Inline的性能，Drop规则触发的流量会被触发为Wdrop (Would Drop) alert

snort --enable-inline-test  
config policy\_mode:inline\_test

## Configuring Snort

### Include

可以在snort.conf文件中使用include包含别的规则文件

#### Variables

规则变量名可以用多种方法修改。可以在"$"操作符之后定义变量。"?" 和 "-"可用于变量修改操作符。

$var - 定义变量。

$(var) - 用变量"var"的值替换。

$(var:-default) - 用变量"var"的值替换，如果"var"没有定义用"default"替换。

$(var:?message) - 用变量"var"的值替换或打印出错误消息"message"然后退出。

#### Config

太多了

### Preprocessor

预处理程序从Snort版本1.5开始引入，使得Snort的功能可以很容易地扩展，用户和程序员能够将模块化的插件方便地融入Snort之中。预处理程序代码在探测引擎被调用之前运行，但在数据包译码之后。通过这个机制，数据包可以通过额外的方法被修改或分析。使用preprocessor关键字加载和配置预处理程序。在Snort规则文件中的preprocessor指令格式如下：

preprocessor <name>: <options> 例子：

preprocessor minfrag: 128

### Decoder and Preprocessor Rule

Decoder和preprocessor rules允许基于规则启用和禁用decoder和preprocessor事件。

Decoder config选项会决定是否产生decoder事件。例如：如果snort.conf中有disable\_decode\_alerts，无论是否有匹配事件的规则，都不会产生decoder事件（alert之类）。

config\_enable\_decode\_drops优先于其他事件的规则。

#### Configuring

decoder和preprocessor规则位于preproc rules/

想启用的话去掉conf文件中关于这部分的注释



规则文件中的alert也可更换为别的，例如log pass drop sdrop reject

#### Reverting to original behavior

config autogenerate\_preprocessor\_decoder\_rules 回复到旧的行为

还需要conf文件中decoder和preprocessor的rules和相关的规则

### Event Processing

1. Detection Filter

用来配置阈值，只有匹配rule超过了这个次数才会产生事件。详见4.7.10

1. Rate Filter

使用rate filter改变一个规则的行为，当时间的次数或是频率意味着可能发生了攻击

1. Event Filter

减少noisy rules带来的logged events的数量。这个能明显降低误报率

1. Event Suppression

完全禁止不感兴趣事件的logging

#### Rate Filtering

在rule规则文件之外单独配置

#### Event Filtering

#### Enent Supression

#### Event Logging

#### Event Trace

### Performance Profiling性能剖析

Snort可以为rule和preprocessor的性能提供统计，只需要在snort.conf中config，snort就会在退出的时候统计。可指定文件名、可指定创建新文件或者append。

### Output Modules

在1.6版本后更新。允许snort在formatting和presentation上更灵活。Output modules在探测引擎和预处理器之后，alert或logging规则触发时调用。

例如：

Output modules are loaded at runtime by specifying the output keyword in the config file:  
output <name>: <options>  
output alert\_syslog: log\_auth log\_alert

#### alert\_syslog

#### alert\_fast

#### Log\_tcpdump

等等等等。。。。

### Host Attribute Table主机属性表

### Dynamic Modules

snort2.6之后加进去的，可在conf文件和命令行中配置

### Reloading a Snort Configuration

### Multiple Configurations

snort允许一个Snort实例对VLAN或IP子网进行多重配置，而不需要为了多种配置而运行多个snort。

### Active Response

snort2.9为inline模式下提供了更好的响应模式

# Snort源码分析

### 主函数（SnortMain函数）流程分析

1. 调用初始化函数SnortInit();
2. 获得数据包，确定链路协议类型，调用GetPacketSource 🡪 返回的是eth0或者是数据包名称
3. 注册控制处理器的句柄，调用ControlSocketRegisterHandler
4. 根据所选择的工作模式，初始化进程文件和权限操作

snort –r读取pcap模式下，运行



直接获取流量的情况下，运行



InlineFailOpen输出0 即为INLINE\_FAIL\_OPEN\_NOT\_USED

1. 调用解码函数SetPkPresessor



目前-r 和 直接捕获dlt值都为1 即为DLT\_EN10MB



实时抓包的话还会输出：



SnortUnprivilegedInit();会输出



直到Commencing packet processing (pid=20512)

下面接着的就是抓到的数据包显示

1. 循环抓包 PacketLoop()



在DAQ\_Acquire()中实现的循环抓包。

例如snort –r读取了一个有六个报文的pcap包，输出信息为



DAQ\_Acquire调用回调函数PacketCallback

PacketCallback中调用ProcessPacket对每一个数据包进行处理

### 每一个数据包

先走ProcessPacket()函数。snort.c中1877行左右。

在ProcessPacket()函数中第四行(\*grinder) (p, pkthdr, pkt); 调用解码引擎，对于以太网，即调用DecodeEthPkt()

再走DecodeEthPkt()函数（因为是以太网的数据包）。decode.c中876行左右。



感叹号为ProcessPacket中打的输出

Begin DecodeEthPkt为DecodeEthPkt输出

### ProcessPacket函数

1. 在读取文件snort -r 和实时监测时，无论是否运行于snort -c入侵检测模式，都不会触发



1. p->proto\_bits
2. NTP (UDP)输出8
3. ICMP输出10
4. TCP、HTTP输出4
5. Profinet实时数据输出0 之后变为0x8000.
6. 0变8000的会走if ( !p->proto\_bits )方法，别的不走if ( !p->proto\_bits )也不走else if
7. 只要不是被标记的忽略的端口的数据，都进入



检测函数Preprocess、包括预处理与入侵检测

调用输出函数

1. 下面的Active\_SessionWasDropped()内都没有走

### DecodeEthPkt函数

* uint32\_t cap\_len = pkthdr->caplen; 嗅探包长 ------ 字节
* uint32\_t rem\_len = cap\_len;
* uint8\_t linklen = ETHERNET\_HEADER\_LEN; 14字节，14\*8 = 112bit ，28个十六进制数

Destination 12个十六进制数 48bit uint8\_t \* 6

Source 12个十六进制数 48bit uint8\_t \*6

Type 4个十六进制数 16bit uint16\_t \* 1

* if(rem\_len < ETHERNET\_HEADER\_LEN) 判断不合法长度包
* p->eh = (EtherHdr \*) pkt; 拆包
* if(ntohs(p->eh->ether\_type) == ETHERNET\_TYPE\_FPATH) 判断是否是FabricPath包，需要特别处理
* PushLayer(PROTO\_ETH, p, pkt, sizeof(\*p->eh));

PROTO\_ETH enum枚举类型 表示当前层的名称

P结构体 pkt包内数据

以太网层 sizeof(\*p->eh)均为14

* DecodeEthTypes(p, p->pkt, ntohs(p->eh->ether\_type), cap\_len, linklen);

判断下一层的类型

P存放解码结果

p->pkt数据包

ntohs(p->eh->ether\_type) 当前层的类型 以太网第一层IPv4 统一为0x0800

* 结束

### 检测函数Preprocess

在detect.c中

DispatchPreprocessors( p, policy\_id, policy );把数据包分发给预处理器

DispatchPreprocessors中

TCP UDP dp、sp是实际值

ICMP是0

pps\_enabled\_foo 都是9 无论TCP UDP还是实时信号

do while循环中

按照预处理插件的顺序依次进行这五个if语句的判断

目前的顺序（就启动了三个预处理器）

1. hello\_snort
2. Frag3
3. Bo

如果进入这个预处理器的话 第二个if语句的判断会返回1

这里需要注意一下这个

#define PROTO\_BIT\_\_NONE 0x0000

#define PROTO\_BIT\_\_IP 0x0001

#define PROTO\_BIT\_\_ARP 0x0002

#define PROTO\_BIT\_\_TCP 0x0004

#define PROTO\_BIT\_\_UDP 0x0008

#define PROTO\_BIT\_\_ICMP 0x0010

#define PROTO\_BIT\_\_TEREDO 0x0020

#define PROTO\_BIT\_\_GTP 0x0040

#define PROTO\_BIT\_\_OTHER 0x8000

#define PROTO\_BIT\_\_ALL 0xffff

在预处理器注册的时候，例如Bo

AddFuncToPreprocList(sc, BoFind, PRIORITY\_APPLICATION, PP\_BO, PROTO\_BIT\_\_UDP);

AddFuncToPreprocList的定义如下：

PreprocEvalFuncNode \* AddFuncToPreprocList(SnortConfig \*sc, PreprocEvalFunc pp\_eval\_func, uint16\_t priority, uint32\_t preproc\_id, uint32\_t proto\_mask)

第二个if判断，决定是否使用这个预处理器

if ( preprocHandlesProto( p, ppn ) && IsPreprocessorEnabled( p, ppn->preproc\_bit ) )

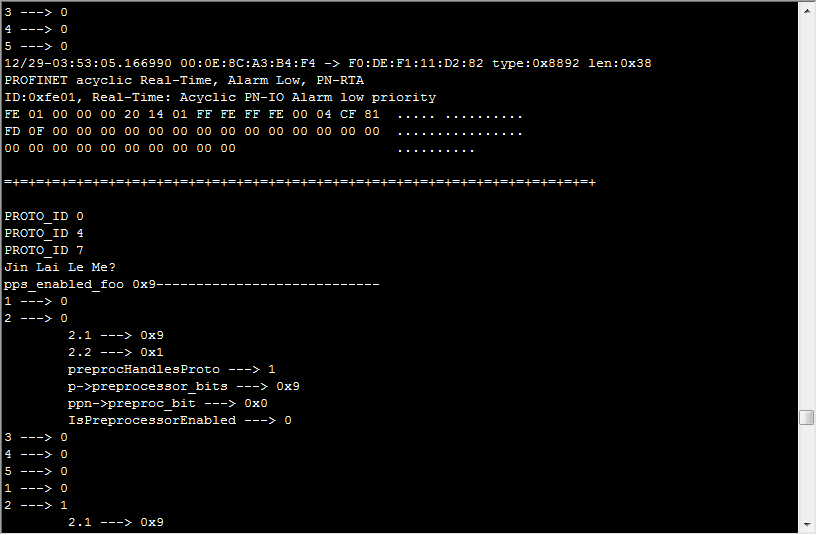
ppn->func( p, ppn->context );

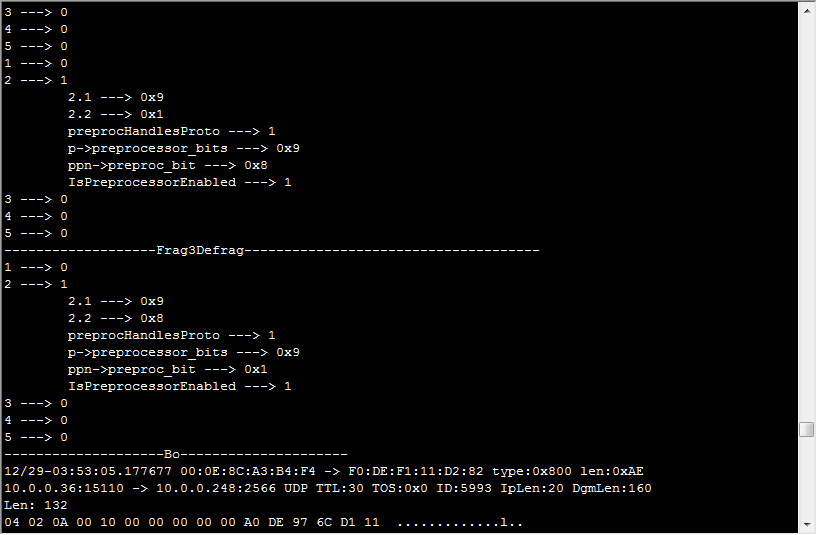
1. preprocHandlesProto( p, ppn )函数：

* ( p->proto\_bits & ppn->proto\_mask ) || ( ppn->proto\_mask == PROTO\_BIT\_\_ALL )
  + p->proto\_bits是解码时候赋值的
  + ppn->proto\_mask是注册预处理器时赋值的（如上）

1. IsPreprocessorEnabled( p, ppn->preproc\_bit )函数

* ( p->preprocessor\_bits & preproc\_bit ) != 0
  + p->preprocessor\_bits来源？？
  + ppn->preproc\_bit来源？？





### PushLayer函数

static inline void PushLayer(PROTO\_ID type, Packet\* p, const uint8\_t\* hdr, uint32\_t len)

{

if ( p->next\_layer < LAYER\_MAX )

{

Layer\* lyr = p->layers + p->next\_layer++;

lyr->proto = type;

lyr->start = (uint8\_t\*)hdr;

lyr->length = (uint16\_t)len;

}

else

{

LogMessage("(snort\_decoder) WARNING: decoder got too many layers;"

" next proto is %u.\n", type);

}

}

typedef struct {

PROTO\_ID proto;

uint16\_t length;

uint8\_t\* start;

} Layer;

### SnortInit函数（功能为初始化Snort各项配置）流程分析

① 解析命令行，调用ParseCmdLine函数

② 根据选择的工作模式，输出相应信息

③ 注册输出模块、预处理模块、规则选项模块

④ 打印所有网卡、接口

⑤ 加载动态插件

## 输出函数Util.c

### DropStats

输出的是最后的统计信息

1120行左右

## 日志函数log.c

### PrintPacket（snort.c中）

函数检测最底层是IP Arp Eapol Wifi四种情况。

Ethernet属于IP这一种

调用PrintIPPkt()

### PrintIPPkt

ScOutputDataLink()返回值

命令行参数带-d时返回非0，不带返回0

ScVerboseByteDump() –dv –dev –v -ev都返回0

-d代表显示应用层

## 预处理、检测函数detect.c

policy\_id ICMP Profinet实时数据都为0

进入DispatchPreprocessors

## 预处理器相关

Plugbase.h中

typedef struct \_PreprocConfigFuncNode

{

char \*keyword; //插件名称

union { //插件的初始化函数

PreprocConfigFunc fptr;

void \*void\_fptr;

} cfptr;

#ifdef SNORT\_RELOAD //这个一般都是def了的。。。。。。

/\* Tells whether we call the config func or reload func \*/

int initialized;

PreprocReloadFunc reload\_func;

PreprocReloadVerifyFunc reload\_verify\_func;

PreprocReloadSwapFunc reload\_swap\_func;

PreprocReloadSwapFreeFunc reload\_swap\_free\_func;

#endif

struct \_PreprocConfigFuncNode \*next; //指向下一个插件

} PreprocConfigFuncNode;

预处理插件在RegisterPreprocessors()中初始化，

RegisterPreprocessors 调用插件的Setup方法

在Setup方法，例如SetupFrag3中调用 RegisterPreprocessor()注册插件

# Snort实验

## 入门级

编辑/etc/snort/rules/local.rules

添加

alert tcp any any -> 10.0.6.59/8 8000 (msg: "Sample alert";classtype:misc-attack;sid: 2002973; rev:1;)

alert icmp any any -> 10.0.6.59/8 any (msg:"Got an ICMP Packet"; classtype:not-suspicious;sid:2000001; rev:1;)

这个ip是公司电脑的。

第一个alert所有连接到10.0.6.59:8000 的tcp包（在10.0.6.59上起了一个web2py）

第二个是检测icmp的

编辑/etc/snort/snort.conf

去掉了Step #7: Customize your rule set 中除local.rules外其他所有的rules

snort -dv -c /etc/snort/snort.conf -l /root/study/Snort/log/ 启动snort

ping主机 或者访问web2py网站即可看到报警信息

/root/study/Snort/log路径下 tail -f alert 可实时观察alert文件的信息变化

产生的报警形如



# Snort修改

### Decode.h

67行添加 #define ETHERNET\_TYPE\_PROFINET 0x8892 这是添加协议识别

1919行void DecodeProfinet(const uint8\_t \*, uint32\_t, Packet \*);

1033行 添加了简单的Profinet包头信息

1692行 添加 const PROFINETHdr \*proh; 在Packet结构体中保存profinet头信息（暂时去掉了）

### Decode.c

610行 添加DecodeProfinet函数

void DecodeProfinet(const uint8\_t \* pkt, uint32\_t len, Packet \* p)

{

pc.profinet++;

//p->proh = (PROFINETHdr \*) pkt;暂时注掉

p->data = pkt; //这里也许有问题 应该去掉包头FrameID

p->dsize = (uint16\_t)len;

//PushLayer(PROTO\_PROFINET, p, pkt, sizeof(\*p->proh));暂时注掉

//p->proto\_bits |= PROTO\_BIT\_\_ARP;

}

860行添加

case ETHERNET\_TYPE\_PROFINET:

DecodeProfinet(pkt + linklen,

cap\_len - linklen, p);

return;

1572行

case ETHERNET\_TYPE\_PROFINET:

DecodeProfinet(pkt + sizeof(VlanTagHdr),

len - sizeof(VlanTagHdr), p);

return;

针对在Vlan上的Profinet协议

612行添加

void DecodeProfinet(const uint8\_t \* pkt, uint32\_t len, Packet \* p)

{

……

p->proto\_bits这个暂时没处理、接下来视情况而定

}

### Snort.h

991行添加 uint64\_t profinet; 这是添加协议计数信息

### Util.c

1175行添加 LogStat("Profinet", pc.profinet, total); 输出profinet的统计信息

### Sf\_protocols.h

31行添加 PROTO\_PROFINET, /\* DecodeProfinet \*/ 应用于Profinet协议的PROTO\_ID

注意：只修改这里 不修改程序别的地方 修改将不会响应

### Log.c

1124行左右添加 PrintProfinetHeader()

打印Profinet包类型

448行 PrintProfinetPkt

打印Profinet数据包

添加.h中对应的函数声明

46 void PrintProfinetPkt(FILE \*, Packet \*);

60 void PrintProfinetHeader(FILE \*, Packet \*);

### preprocids.h

#define PP\_HELLO\_SNORT 34 // 自己添加的 预处理器标识

#define PP\_MAX 35

添加了PP\_HELLO\_SNORT PP\_MAX 34改成35

# 备忘

uint\_8 代表一个字节

以太网头部

typedef struct \_EtherHdr

{

uint8\_t ether\_dst[6];

uint8\_t ether\_src[6];

uint16\_t ether\_type;

} EtherHdr;

const DAQ\_PktHdr\_t \*pkth; // packet meta data 捕获包的包头

const uint8\_t \*pkt; // raw packet data 指向捕获到的数据包



C语言输出 不足补零 %04x 代表一共4位的十六进制数

1. 协议名 枚举类型 输出时会输出对应的序号从0开始

typedef enum {

**0** PROTO\_ETH, /\* DecodeEthPkt \*/

**1** PROTO\_PROFINET, /\* DecodeProfinet \*/ 自己添加的

**2** PROTO\_FPATH, /\* FabricPath - handled by DecodeEthPkt \*/

PROTO\_CISCO\_META, /\* Cisco Metadata - handled by DecodeEthPkt \*/

PROTO\_IP4, /\* DecodeIP \*/

/\* DecodeIPOptions - handled with IP4 \*/

PROTO\_ICMP4, /\* DecodeICMP \*/

PROTO\_ICMP\_IP4, /\* DecodeICMPEmbeddedIP \*/

PROTO\_UDP, /\* DecodeUDP \*/

PROTO\_TCP, /\* DecodeTCP \*/

/\* DecodeTCPOptions - handled with TCP \*/

PROTO\_IP6, /\* DecodeIPV6 \*/

/\* DecodeIPV6Extensions - nothing to do here, calls below \*/

PROTO\_IP6\_HOP\_OPTS, /\* DecodeIPV6Options - ip6 hop, dst, rte, and frag exts \*/

PROTO\_IP6\_DST\_OPTS,

**12** PROTO\_ICMP6, /\* DecodeICMP6 \*/

**13**  PROTO\_ICMP\_IP6, /\* DecodeICMPEmbeddedIP6 \*/

**14**  PROTO\_VLAN, /\* DecodeVlan \*/

#ifdef GRE

PROTO\_GRE, /\* DecodeGRE \*/

/\* DecodeTransBridging - basically same as DecodeEthPkt \*/

PROTO\_ERSPAN, /\* DecodeERSPANType2 and DecodeERSPANType3 \*/

#endif

PROTO\_PPPOE, /\* DecodePPPoEPkt \*/

PROTO\_PPP\_ENCAP, /\* DecodePppPktEncapsulated \*/

PROTO\_MPLS, /\* DecodeMPLS - decoder changes pkth len/caplen! \*/

/\* DecodeEthOverMPLS - basically same as straight eth \*/

PROTO\_ARP, /\* DecodeARP \*/

PROTO\_GTP, /\* DecodeGTP \*/

PROTO\_AH, /\* DecodeAH - Authentication Header (IPSec stuff) \*/

#ifndef NO\_NON\_ETHER\_DECODER

PROTO\_TR, /\* DecodeTRPkt \*/

PROTO\_FDDI, /\* DecodeFDDIPkt \*/

PROTO\_LSLL, /\* DecodeLinuxSLLPkt sockaddr\_ll for "any" device and \*/

/\* certain misbehaving link layer encapsulations \*/

PROTO\_80211, /\* DecodeIEEE80211Pkt \*/

PROTO\_SLIP, /\* DecodeSlipPkt - actually, based on header size, this \*/

/\* must be CSLIP (TCP/IP header compression) but all it \*/

/\* does is skip over the presumed header w/o expanding \*/

/\* and then jumps into IP4 decoding only; also, the actual \*/

/\* esc/end sequences must already have been removed because \*/

/\* there is no attempt to do that. \*/

PROTO\_L2I4, /\* DecodeI4LRawIPPkt - always skips 2 bytes and then does \*/

/\* IP4 decoding only \*/

PROTO\_L2I4C, /\* DecodeI4LCiscoIPPkt -always skips 4 bytes and then does \*/

/\* IP4 decoding only \*/

PROTO\_CHDLC, /\* DecodeChdlcPkt - skips 4 bytes and decodes IP4 only. \*/

PROTO\_PFLOG, /\* DecodePflog \*/

PROTO\_OLD\_PFLOG, /\* DecodeOldPflog \*/

PROTO\_PPP, /\* DecodePppPkt - weird - optionally skips addr and cntl \*/

/\* bytes; what about flag and protocol? \*/

/\* calls only DecodePppPktEncapsulated. \*/

PROTO\_PPP\_SERIAL, /\* DecodePppSerialPkt - also weird - requires addr, cntl, \*/

/\* and proto (no flag) but optionally skips only 2 bytes \*/

/\* (presumably the trailer w/chksum is already stripped) \*/

/\* Calls either DecodePppPktEncapsulated or DecodeChdlcPkt. \*/

PROTO\_ENC, /\* DecodeEncPkt - skips 12 bytes and decodes IP4 only. \*/

/\* (add family + "spi" + "flags" - don't know what this is) \*/

PROTO\_EAP, /\* DecodeEAP \*/

PROTO\_EAPOL, /\* DecodeEapol - leaf decoder \*/

PROTO\_EAPOL\_KEY, /\* DecodeEapolKey - leaf decoder \*/

#endif /\* NO\_NON\_ETHER\_DECODER \*/

PROTO\_MAX

} PROTO\_ID;

# Errors

~~在添加完最初始的修改后 不知为何读取以前的数据包会多出~~

~~Eth Loop: 6 (100.000%)~~

~~没找到原因，也没有进这个方法~~

~~void DecodeEthLoopback(const uint8\_t \*pkt, uint32\_t len, Packet \*p)~~

~~{~~

~~DEBUG\_WRAP(DebugMessage(DEBUG\_DECODE, "EthLoopback is not supported.\n"););~~

~~printf("-------------------caonima\n");~~

~~pc.ethloopback++;~~

~~#ifdef GRE~~

~~if (p->greh != NULL)~~

~~pc.gre\_loopback++;~~

~~#endif~~

~~return;~~

~~}~~

Make clean后重新configure后没这个问题了

1. 入侵检测模式时 必会Segmentation fault (core dumped)

出错程序确定到 detect.c

DispatchPreprocessors()方法中

do while循环 中

if ( preprocHandlesProto( p, ppn ) && IsPreprocessorEnabled( p, ppn->preproc\_bit ) )

**ppn->func( p, ppn->context );**

此时的preproc\_id 13 ,preproc\_bit 0x2000, proto\_mask 0xc

插件的加载顺序

preproc\_id 20 ,preproc\_bit 0x100000, proto\_mask 0xffff

preprocHandlesProto 1 IsPreprocessorEnabled 1

preproc\_id 3 ,preproc\_bit 0x8, proto\_mask 0x1

preprocHandlesProto 1 IsPreprocessorEnabled 1

**preproc\_id 13 ,preproc\_bit 0x2000, proto\_mask 0xc**

**preprocHandlesProto 1 IsPreprocessorEnabled 1**

preproc\_id 5 ,preproc\_bit 0x20, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 1

preproc\_id 7 ,preproc\_bit 0x80, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 0 ,preproc\_bit 0x1, proto\_mask 0x8

preprocHandlesProto 0 IsPreprocessorEnabled 1

preproc\_id 10 ,preproc\_bit 0x400, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 11 ,preproc\_bit 0x800, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 16 ,preproc\_bit 0x10000, proto\_mask 0xc

preprocHandlesProto 1 IsPreprocessorEnabled 1

preproc\_id 2 ,preproc\_bit 0x4, proto\_mask 0xc

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 12 ,preproc\_bit 0x1000, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 23 ,preproc\_bit 0x800000, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 22 ,preproc\_bit 0x400000, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 28 ,preproc\_bit 0x10000000, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 29 ,preproc\_bit 0x20000000, proto\_mask 0xc

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 4 ,preproc\_bit 0x10, proto\_mask 0x4

preprocHandlesProto 1 IsPreprocessorEnabled 0

preproc\_id 21 ,preproc\_bit 0x200000, proto\_mask 0xc

preprocHandlesProto 1 IsPreprocessorEnabled 0

不一定会全部遍历完、但是profinet协议的话肯定会遍历完

对应的编号应该是下面的这个

#define PP\_BO 0

#define PP\_APP\_ID 1

#define PP\_DNS 2

#define PP\_FRAG3 3

#define PP\_FTPTELNET 4

#define PP\_HTTPINSPECT 5

#define PP\_PERFMONITOR 6

#define PP\_RPCDECODE 7

#define PP\_SHARED\_RULES 8

#define PP\_SFPORTSCAN 9

#define PP\_SMTP 10

#define PP\_SSH 11

#define PP\_SSL 12

#define PP\_STREAM 13

#define PP\_TELNET 14

#define PP\_ARPSPOOF 15

#define PP\_DCE2 16

#define PP\_SDF 17

#define PP\_NORMALIZE 18

#define PP\_ISAKMP 19 // used externally

#define PP\_SESSION 20

#define PP\_SIP 21

#define PP\_POP 22

#define PP\_IMAP 23

#define PP\_NETWORK\_DISCOVERY 24 // used externally

#define PP\_FW\_RULE\_ENGINE 25 // used externally

#define PP\_REPUTATION 26

#define PP\_GTP 27

#define PP\_MODBUS 28

#define PP\_DNP3 29

#define PP\_FILE 30

#define PP\_FILE\_INSPECT 31

#define PP\_NAP\_RULE\_ENGINE 32

#define PP\_PREFILTER\_RULE\_ENGINE 33 // used externally

#define PP\_MAX 34

在DecodeProfinet中把存到Packet结构体中的proh去掉、

就不报这个错了、、可是这个不应该影响啊，平时的流量也进不来这个方法（5/21）