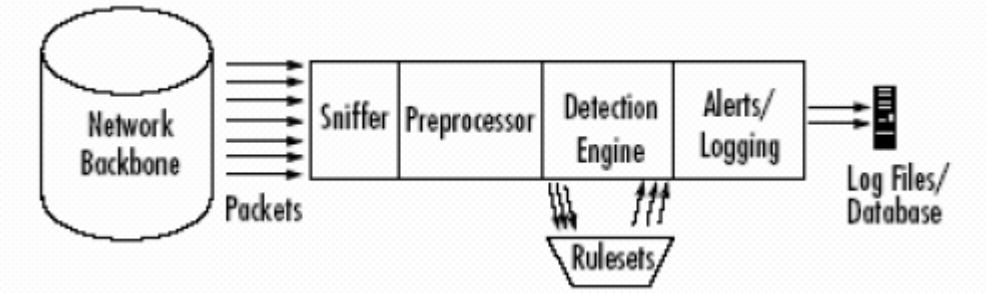
Snort 原代码分析

一：引言

Snort 系统是一个以开放源代码形式发行的网络入侵检测系统，由 Martin Roesch 编写，并由遍布世界各地的众多程序员共同维护和升级。Snort 运行在 Libpcap 库函数基础之上，并支持多种系统软硬件平台。它具有实时数据流量分析和记录 IP 网络数据包的能力，能够进行协议分析，对内容搜索/匹配。它能够检测各种不同的攻击方式，对攻击进行实时警报。本文中所使用的原代码为 Snort1.9.1(winpcap2.3)，调试环境为 windows2000 sp4 + VC6.0。

二：系统程序构架

Snort 主要由以下几大部分组成：数据包嗅探器，预处理器，检测引擎和报警输出模块。具体流程如下图：



Snort 从网卡取得数据包后先用预处理插件进行处理，然后经过检测引擎中的所有规则链，如果有符合规则链的数据包则会被检测出来，按照规则进行处理。下面是 Snort 主函数 main 的代码的主要执行流程图：

1. 初始化过程。包括 winsock 的初始化 init_winsock()，网络掩码数据 InitNetmasks()和协议名称数组 InitProtoNames()的初始化工作，以及各个 PV 结构参数初始化（Snort 命令行使用方式及 PV 数据结构请见附录）。

2. 命令行参数解析 ParseCmdLine(argc, argv)。将对应的命令行开关参数赋给相应的 PV 结构参数。

3. 打开 Libpcap（）包捕获接口。

4. 调用输出插件模块初始化函数 InitOutputPlugins()，注册所有的输出插件。

5. 调用 SetPktProcessor()根据所选用的数据链路层协议类型，设置对应的数

据包处理函数，该函数指针保存在全局变量 grinder 中。

6. 调用预处理器插件初始化函数 InitPreprocessors()，注册所有的预处理器插件。

7. 调用插件初始化函数 InitPlugins ()，注册所有的检测引擎插件。

8. 调用规则文件解析函数 ParseRulesFile，读入规则文件中的规则。

9. 调用 ProcessAlertCommandLine()和 ProcessLogCommandLine()设置指定的警告和日志模式。

10. 调用了建立和初始化规则优化和快速匹配数据结构的类型。其中 OtnXMatchInfoInitialize()主要是对数据类型为 OTNX_MATCH_DATA 的全局变量 omd 申请空间。OTNX_MATCH_DATA 数据结构中包括快速匹配时所需要的重要信息。而 fpCreateFastPacketDetection()是建立快速匹配引擎的主要接口函数，

11. 最后正式启动数据包捕获/读取和处理进程。这里主要调用 Libpcap 库函数来进行捕获网卡上的数据并且赋给 Packet 结构，然后调用 Preprocess(&p);对 Packet 结构的数据进行处理。

以上是 main 函数的简单介绍，由以上的过程可以看出 snort 原代码主要分为以下几大部分，一是利用 Libpcap 库函数来进行数据捕获机制，二是插件模块的管理，三是规则文件的读入以及链接预处理插件、检测插件以及输出插件的处理函数。四是对捕获的数据的进行处理。下面也将以这四个部分进行较为详细的介绍。

三：利用 Libpcap 库函数来进行数据捕获。

1，利用 OpenPcap 函数打开包捕获接口

1) 得到网卡设备列表，PCAP 提供了 pcap_findalldevs(errorbuf)这个函数来实现此功能，这个函数返回一个 pcap_if 结构的链表，链表的每项内容都含有全面的网卡信息，尤其是字段名字和含有名字的描述以及有关驱动器的易读信息。

2) 打开网卡并捕获数据流，打开网卡的功能是通过 pcap_open_live(d->name,snaplen, promisc,to_ms，errbuf)来实现的它。d 可以就是我们刚刚返回的网卡设备列表，snaplen 用于指定所捕获包的特定部分，在一些系统上（象 xBSD and Win32 等）驱动只给出所捕获数据包的一部分而不是全部，这样就减少了拷贝数据的数量从而提高了包捕获的效率。 promisc 指

明网卡处于混杂模式，在正常情况下网卡只接受发往该主机的数据包，而发往其其主机的数据包则被忽略，相反当网卡处于混杂模式时他将接收所有的流经它的数据包。 to_ms 参数指定读数据的超时时间，超时以毫秒计算。当在超时时间内网卡上没有数据到来时对网卡的读操作将返回。

3）利用函数 pcap_datalink(pd) 返回数据链路层协议类型存储在全局变量 datalink 中。

2，利用 SetPktProcessor()会根据不同的 datalink 值指定不同的包处理函数，并且将包处理函数赋给全局变量 grinder，譬如 Ethernet 类型设置包处理函数为 grinder=DecodeEthPkt。

3，一旦网卡被打开，就可以调用 pcap_loop(adhandle, 0, packet_handler, NULL)来进行数据的捕获，adhandle 是 pcap_open_live 的返回值，packet_handler 是回调函数。在 snort 代码中 grinder 就是我们的 Packet_handler 函数，譬如数据链路层协议类型为 Ethernet 类型就调用 DecodeEthPkt(Packet * p, struct pcap_pkthdr * pkthdr, u_int8_t * pkt) 对捕获的数据 pkt 进行处理。

4，对数据解码的实现在 decode.h 以及 decode.c 中，定义了一个非常重要的结构 Packet，解码的任务就是将捕获的数据按照协议类型填充到 Packet 中。这部分主要是对数据包的表头进行分析，数据包类型实在是太多，这里不一一列举，拿 TCP 包做个例子吧。

TCP 数据包格式如下：

包的格式

ethernet报头（14字节）	IP报头（20字节）	TCP报头（20字节）	应用数据（0-1460字节）

1. 进行 ethernet 报头解析，判断报头内的 ether_type 是否等于 ETHERNET_TYPE_IP，假如不等于，那肯定不是 TCP 包，假如等于那么我们移动 pkt 指针进入 IP 报头的分析，在 snort 中是调用了函数 DecodeIP。

2. 进行 IP 报头解析，判断报头内的 ip_proto 是否等于 IPPROTO_TCP，假如不等于，那么不是 TCP 包，采用别的方式，假如等于的话，那么我们下面就可以移动 pkt 指针进入 TCP 报头的分析，在 snort 中调用了函数 DecodeTCP。当然 DecodeIP 里面有很多对畸形包的判断，这里就不列举了。

3. 进行 TCP 报头的解析，将 TCP 的报头数据相应的赋给 Packet 结构。DecodeTCP 中还有畸形包处理以及 DecodeTCPOptions 函数就不分析了。到这里一个 TCP 数据包就分析完毕，捕获到的数据包中的参数已经被解码并且相应的存储到 Packet 结构里，下面就可以把这个 Packet 结构送入预处理插件。数据包有很多类型，snort 都一一对应的定义了相应的处理函数，但目的无非是填充 Packet 结构使其进行后续处理。

四：插件模块的管理分析

snort 一个非常成功的思想是利用了 plugin 机制，snort 先注册好所有的插件模块，然后从规则文件中读入规则，再把每个规则所需要的处理函数挂接到链表上。实际检测时，遍历这些链表，调用链表上相应的函数来分析。Snort 系统中的插件模块共分为三种类型：预处理器插件模块、输出插件模块以及处理规则选项关键字的插件模块，定义如下：（它们的数据结构都比较简单，都是一个描绘该节点状态的 entry 字段和指向下一节点的指针 next，PluginCleanExitList 和 PluginRestartList 同理）

KeywordXlateList *KeywordList;

PreprocessKeywordList *PreprocessKeywords;

PreprocessFuncNode *PreprocessList;

OutputKeywordList *OutputKeywords;

OutputFuncNode *OutputList;

OutputFuncNode *LogList;

由此组成了预处理器插件模块、输出插件模块以及处理规则选项关键字的插件模块的单向链表，下面具体以预处理器插件模块为例来简要分析插件模块的管理过程：

1. 主函数中调用预处理器插件模块初始化函数 InitPreprocessors()。

2. InitPreprocessors()函数中调用多个预处理器的 Setup*（）函数。

3. Setup*()函数中调用函数 RegisterPreprocessor 对各个预处理器进行注册，它将模块的关键字赋给 PreprocessKeywords->entry.keyword，并且将模块的处理函数赋给 PreprocessKeywords->entry.func。

4. 由此就形成了一个预处理器插件模块的链表：http_decode :

HttpDecodeInit -> http_decode_ingnore : HttpDecodeInitIgnore -> portscan : PortscanInit … -> … -> PerfMonitor : PerfMonitorInit。

5. 当规则文件中定义了某个预处理器后，将遍历预处理器插件模块的链表，找到与关键字相对应的初始化函数，从而将预处理器的处理函数挂接到函数列表中。例如假设在 snort.conf 中有这样一行规则：preprocessor frag2，经过 ParseRule（下面会详细讲解如何读取规则文件）后得出 rule_type=RULE_PREPROCESS（显而易见因为有 preprocessor 字段），那么将调用 ParsePreprocessor，在 ParsePreprocessor 函数中遍历刚刚建立的预处理器插件模块的链表，找到 PreprocessKeywords->entry.keyword 为 frag2 的 PreprocessKeywords，调用对应的 PreprocessKeywords->entry.func=Frag2Init 进行处理，在 Frag2Init 中设置 frag2 预处理器的一些参数，并且把 frag2 预处理器的处理函数 Frag2Defrag 利用 AddFuncToPreprocList 挂接到对预处理器处理函数列表 PreprocessList 中。

6. 到此为止，一个规则中需要的预处理模块加载完毕。规则中定义的预处理器模块处理函数已经导入 PreprocessList 中。

可以归纳该过程如下：

初始化预处理器形成一个 PreprocessKeywords ---> 读取规则文件并且遍历 PreprocessKeywords 找出与对应预处理器 ---> 调用对应的 PreprocessKeywords->entry.func- ----->将预处理的处理函数挂接到 PreprocessList 中。

由上面分析可知，预处理器模块是可以用插件扩展的，那么下面看看如何编写自己的预处理器插件模块模块：（建立一个 Mypreproc 模块）

1. 主要是修改 plugbase.c 文件，将插件的头文件 spp_mypreproc.h 包含到 plugbase.c 中，并且将插件的 Setupmypreproc（）函数插入 plugbase.c 的 InitPreprocessors()函数中。

2. 建立 spp_mypreproc.h 文件，即定义初始化函数原形 void Setupmypreproc ();

3. 建立以及 spp_mypreproc.c 文件，

编写函数 Setupmypreproc（）

{ RegisterPreprocessor("mypreproc", SetupmypreprocInit);}

之后编写函数 SetupmypreprocInit（）

{AddFuncToPreprocList(mypreproc);AddFuncToCleanExitList(mypreprocCleanExit, NULL); AddFuncToRestartList(mypreprocRestart, NULL);}

其中 mypreproc 函数就是实现自定义预处理功能的处理函数，mypreprocCleanExit 和 mypreprocRestart 是相应的清除和重启该预处理器的处理函数。

到这里我们分析了预处理器插件模块依照规则文件的加载过程以及如何编写自己定义的预处理器插件模块，相信大家对 snort 的 plugin 机制比较了解了。不过这里只是谈了谈预处理器插件模块的管理过程，并没有深入到某个预处理器模块中。真正要深入每个预处理器模块的具体函数中，比如要了解函数 Frag2Defrag 是怎么具体实现分片重组功能的，必须对这分片重组的原理做详细的了解，笔者功力不够，只能讲讲思路了，不能完成对每个预处理器中一些具体函数分析。

上面介绍了预处理器插件模块管理过程。输出插件模块以及处理规则选项关键字的插件模块的加载过程和编写和预处理器插件模块是类似的，都采用的是 plugin 机制，所以这里不再一一列举了。

五：规则文件的读取

在这部分里主要实现的功能是读取规则文件，并且把规则文件中的各项填充相应的数据结构中，完成这部分功能的函数为 ParseRulesFile，现在就让我们深入内部去了解它吧。读取规则文件中的一行，假如是有效行的话（例如不是#开头等等），调用函数 ParseRule。下面分别用 snort.conf 中几种典型的规则语句来说明规则文件的读取过程：

1. var HTTP_PORTS 80

判断规则类型属于 RULE_VAR 情形，进入 VarDefine 函数，在函数 VarDefine 内主要是填充 VarEntry 结构的并且挂接到全局变量 VarHead 链表中，VarEntry 结构定义如下：

```
struct VarEntry
{
```

```
    char *name;
    char *value;
    unsigned char flags;
    struct VarEntry *prev;
    struct VarEntry *next;
};
```

应用 var HTTP_PORTS 80 这句规则：先申请一个 VarEntry *p，使

```
p->name = strdup(name);
p->value = strdup(value);
```

然后将 p 插入到 VarHead 双向链表中，

```
p->prev = VarHead;
p->next = VarHead->next;
p->next->prev = p;
VarHead->next = p;
```

由此形成了存储 var 变量的双向链表，如：(再增加一句规则 var ORACLE_PORTS 1521)

VarHead->name:Varhead->value → HTTP_PORTS:80 → ORACLE_PORTS：1521—> ...当需要寻找某个 var 变量值时，遍历该双向链表即可。

2. preprocessor frag2，preprocessor 类型的加载过程上节已经讨论过了。

3. include *.rules，include 类型主要是还是调用 ParseRulesFile 函数来解析*.rules 文件，那么直接来看看*.rules 里的规则如何解析。

4，alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Jolt attack"; fragbits:M;dsize:408;reference:cve,CAN-1999-0345;classtype:attempted-dos; sid:268; rev:2;)对于这样的一条规则，主要是将规则中的各项填充到 RuleTreeNode 和 OptTreeNode 结构中，RuleTreeNode 是用于描绘二维规则链表结构中链表头的结构，其中包含了规则所及的数据包目的与源 IP 地址、端口号等信息。以及用于处理该链表头的函数列表指针等等，RuleTreeNode 数据结构如下：

```
typedef struct _RuleTreeNode
{
```

```
    RuleFpList *rule_func;     //规则列表头匹配处理的函数列表指针
    int head_node_number;    //链表头的编号值
    int type;                  //规则的类型，有三种：日志，忽略及警报类型
    IpAddrSet *sip;            //源 IP 地址
    IpAddrSet *dip;           //目的 IP 地址
    int not_sp_flag;          //源端口号的!（取反）标志
    u_short hsp;              //源端口的起始值
    u_short lsp;              //源端口的结束值
    int not_dp_flag;          //目标端口号的!（取反）标志
    u_short hdp;              //目标端口的开始值
    u_short ldp;              //目标端口的开始值
    u_int32_t flags;          //控制标志
    int active_flag;
    int activation_counter;
    int countdown;
    ActivateList *activate_list;
    struct _RuleTreeNode *right; //指向下一个规则链表头的指针
    OptTreeNode *down;         //指向与该链表头相联的规则链表选项的指针
    struct _ListHead *listhead;
} RuleTreeNode;
```

仍然是进入 ParseRule 函数，下面是填充 RuleTreeNode 数据结构的详细过程：

a) 调用 WhichProto 解析规则的第二个参数（ip）得到协议名称，那么这条规则协议名称为 IP

b) 调用 ProcessIP 解析规则的第三个参数（$EXTERNAL_NET）从而填充 IpAddrSet *sip，主要是填充发送源的地址，网络掩码，及是否为非（!）标志。因为本规则中源地址为$EXTERNAL_NET，并且 snort.conf 中定义 var EXTERNAL_NET any，故源地址为 any，由此填充 sip->ip_addr=0, sip->netmask=0, sip->addr_flags=0, 并且置控制为 flags|=ANY_SRC_IP;

c) 调用 ParsePort 解析规则的第四个参数（any）从而赋值 not_sp_flag，hsp ，

lsp。本规则中第四个参数为 any,所以起始值 lsp=0，结束值 hsp=0，not_sp_flag=1，并且置控制为 flags|=ANY_SRC_PORT;

d) 解析第五个参数（即流向标志），当第五个参数为<>时，置控制为 flags|=BIDIRECTIONAL;

e) 同理，下面解析第六个第七个参数，填充 dip，not_dp_flag，hdp，ldp。

f) 调用 ProcessHeadNode 将上面填充的 RuleTreeNode 挂入到 ListHead 结构下，在本规则中将 RuleTreeNode 挂接到 AlertList->IpList，形成一个 IP 协议包检测的链表,最终调用 SetupRTNFuncList 将需要使用的处理函数导入 rule_func 链表。在本例中由于源地址为 any，故没有导入 CheckSrcIP 函数，假设定义了源地址，那么会将 CheckSrcIP 导入到 rule_func 链表中。到此为止，规则头的解析完毕，RuleTreeNode 结构个部分均已填充。

下面的目标是解析规则体，主要是填充 OptTreeNode 结构。该数据结构是用于描绘二维规则链表结构中每个链表选项的结构，具体结构定义如下

```
typedef struct _OptTreeNode
{
    OptFpList *opt_func;     //指向插件和检测函数列表的指针
    RspFpList *rsp_func;     //response functions */
    void *ds_list[64];       //指向插件数据结构的指针数组
    int chain_node_number; //当前链表选项的编号
    int type;                //规则的类型
    int evalIndex;           // where this rule sits in the evaluation sets */
    int proto;               //协议类型，用于规则解析的一致性检查
    struct _RuleTreeNode *proto_node;     //指向规则头的指针
    int session_flag;        //指示记录会话数据的标识符
    char *logto;             //指向日志文件名称字符串的指针
    SigInfo sigInfo;         // metadata about signature
    u_int8_t stateless;      //this rule can fire regardless of session state
    u_int8_t established; // this rule can only fire if it has been marked as established
    Event event_data;
    TagData *tag;
```

```
    int active_flag;                // stuff for dynamic rules activation/deactivation
    int activation_counter;
    int countdown;
    int activates;
    int activated_by;
    u_int8_t    threshold_type; // type of threshold we're watching
    u_int32_t threshold;        // number of events between alerts
    u_int32_t window;           //number of seconds before threshold times out
    struct _OptTreeNode *OTN_activation_ptr;
    struct _RuleTreeNode *RTN_activation_ptr;
    struct _OptTreeNode *next;
    struct _RuleTreeNode *rtn;
} OptTreeNode;
```

进入 ParseRuleOptions 函数，下面是填充 OptTreeNode 数据结构的详细过程：

g) 规则体与规则头相连，otn_tmp = rtn_tmp->down; otn_tmp->proto_node = rtn_tmp;

h) 以“;”为分界符，读取“(”内内容，下面就是进入条件分支，分支中有两中类型，一种是利用上一节我们所说的处理规则选项关键字的插件模块来对规则体的内容进行处理，另一种是其它的一些规则选项，譬如 msg 选项，没有必要作成处理规则选项关键字的插件模块来进行处理，所以在这里就直接进入分支程序进行处理了。

i) 在本规则中第一条是 msg:"DOS Jolt attack"，那么进入函数 ParseMessage，将"DOS Jolt attack"赋给 otn_tmp->sigInfo.message 。

j) 第二条是 fragbits: M，这个在处理规则选项关键字的插件模块里定义里具体的处理函数，所以遍历 KeywordList（还记得吗？这个就是上一节注册的三个插件模块链表之一），找到 entry.keyword=fragbits 的那项，调用对应的 entry.func（也就是 FragBitsInit）对标志为 M 的 fragbits 规则进行配置，最后将 CheckFragBits 加入到 otn_tmp->opt_func 中。

k) 第三条是 dsize:408，仍然遍历 KeywordList，调用 DsizeCheckInit 以及

ParseDsize 对规则 dsize:408 进行处理，最后将 CheckDsizeEq 加入到 otn_tmp->opt_func 中。形成一个检测处理的函数链表。

l) 同理。。。。直至将整个规则体全部处理完成。

到此为止，已经处理了规则头和规则体，整个规则处理完毕。规则头匹配处理的函数列表 rtn_tmp-> opt_func 已经形成，指向插件和检测函数列表 otn_tmp->opt_func 也已经完成。只等 packet 结构数据通过这些函数的检测了。

五：对 Packet 结构的数据的处理

对 Packet 结构的数据的处理在函数 Preprocess 中，下面来详细了解一下该函数的过程：

1. 让捕获到的数据 p 经过由规则文件创建的预处理器处理函数列表，对数据 p 进行预处理。

2. 进入 Detect 函数，再进入 fpEvalPacket 函数，根据协议的不同进入不同的处理函数，譬如为 TCP 数据包时进入 fpEvalHeaderTcp

3. 在函数 fpEvalHeaderTcp 调用函数 prmFindRuleGroupTcp 判断该 TCP 包是否需要处理（按照规则文件中定义的原地址、原端口以及目的地址、目的端口来判断），分为不需要判断，原地址需要判断，目标地址需要判断，原地址和目标地址都需要判断，以及自定义的判断这 5 中情况，调用 fpEvalHeaderSW 函数来完成这些判断。

4. 接下来就是调用 otn_tmp->func 的函数处理列表对数据进行检测处理。具体就涉及到上面读取规则读入到 otn_tmp->func 链表中的具体函数，了解这些函数需要对该漏洞或者该种方式的攻击比较了解，才能为什么规则要这么定并且为什么要检测这些参数，这部分的函数实在是太多，不再一一列举，对该函数有兴趣的读者可以找到相关函数进行分析。譬如很多时候都想知道到底捕获到的数据如何和规则中的 content 项进行处理，那么我们可以知道真正对数据进行 content 项进行处理的函数为 CheckANDPatternMatch，那么我们就可以从 CheckANDPatternMatch 开始分析可以查得该函数到底是如何检测 content 项的。

以上是进行包检测的一些过程，具体函数都没涉及，只是分析了大概的过程，但是我相信根据前面几节分析的基础，对于一个具体的规则要找到具体的处理函数进行分析还是比较容易了。还有就是 snort 对捕获的数据进行检测时不会检测

到某一警告后就直接退出，它一定会通过所有的检测函数，检测出所有可能存在的警告。

六：结束语

在学习入侵检测的过程中，觉得有必要了解一下入侵检测的原代码，这样一来有利于了解入侵检测的实际过程，二是为将自己的算法思想应用到实践中，用实验来检验算法的优劣程度。所以对 Snort 原代码进行了一下分析，但说实话我也是刚刚接触 snort 时间不长，在文中仅仅分析了 snort 的一些实现过程以及各部分的大概思路，在最重要的插件模块管理里面也只简要分析了一下最简单的预处理器插件模块，另外两个更为重要的模块没有涉及，具体的处理函数几乎没有做过分析。但希望本文能对您了解 snort 的实现过程有些帮助，如有错误或不妥的地方也敬请原谅。

Sagely,在读研究生，主要做 PKI，入侵检测，防火墙等网络信息安全方面的研究，您可以通过 sagely_zou@163.com 与他联系。

附录：Snort 系统开关选项以及对应的 PV 数据结构

-A <alert> 设置<alert>的模式是 full,fast,还是 none;full 模式是记录 标准的 alert 模式到 alert 文件中；Fast 模式只写入时间戳，messages,IPs,ports 到文件中，None 模式关闭报警。

-a 显示 ARP 包。 对应 pv.showarp_flag

-b 把 LOG 的信息包记录为 TCPDUMP 格式，所有信息包都被记录成二进制形式，名字如 snort-0612@1385.log，这个选项对于 FAST 记录模式比较好，因为它不需要花费包的信息转化为文本的时间。Snort 在 100Mbps 网络中使用"-b" 比较好。对应 pv.logbin_flag

-c <cf> 使用配置文件<cf>,这个规则文件是告诉系统什么样的信息要 LOG，或者要报警，或者通过。

-C 在信息包信息使用 ASCII 码来显示，而不是 hexdump。对应 pv.char_data_flag

-d 解码应用层。 对应 pv.data_flag

-D 把 snort 以守护进程的方法来运行，默认情况下 ALERT 记录发送到 /var/log/snort.alert 文件中去。对应 pv.daemon_flag

-e 显示并记录 2 个信息包头的数据。对应 pv.show2hdr_flag

-F <bpf>从<bpf>文件中读 BPF 过滤器（filters），这里的 filters 是标准的 BPF 格式过滤器，你可以在 TCPDump 里看到，你可以查看 TCPDump 的 man 页怎样使用这个过滤器。

-h <hn>设置网络地址，如一个 C 类 IP 地址 192.168.0.1 或者其他的，使用这个选项，会使用箭头的方式数据进出的方向。

-I <if> 使用网络接口参数<if>

-l <ld> LOG 信息包记录到<ld>目录中去。

-M <wkstn> 发送 WinPopup 信息到包含<wkstn>文件中存在的工作站列表中去，这选项需要 Samba 的支持，wkstn 文件很简单，每一行只要添加包含在 SMB 中的主机名即可。（注意不需要\\两个斜杠）。对应 pv.smbmsg_flag

-n <num> 是指定在处理<num>个数据包后退出。

-N 关闭 LOG 记录，但 ALERT 功能仍旧正常。对应 pv.nolog_flag

-o 改变所采用的记录文件，如正常情况下采用 Alert->Pass->Log order，而采用此选项是这样的顺序：Pass->Alert->Log order，其中 Pass 是那些允许通过的规则而不记录和报警，ALERT 是不允许通过的规则， LOG 指 LOG 记录，因为有些人就喜欢奇奇怪怪，象 CASPER，QUACK 就喜欢反过来操作。对应 pv.rules_order_flag

-O 在对数据包以 ASCII 格式转储时，隐藏相应的 IP 地址 对应于 pv,.obfuscation_flag

-p 关闭杂乱模式嗅探方式，一般用来更安全的调试网络。对应 pv.promisc_flag

-q 设置安静模式，不显示任何状态信息或提示信息。对应 pv.quite_flag

-r <tf> 读取 tcpdump 方式产生的文件<tf>,这个方法用来处理如 得到一个 Shadow(Shadow IDS 产生)文件，因为这些文件不能 用一般的 EDIT 来编辑查看。对应 pv.readmode_flag

-s LOG 报警的记录到 syslog 中去，在 LINUX 机器上，这些警告信息会出现在 /var/log/secure, 在其他平台上将出现在 /var/log/message 中去。对应 pv.syslog_flag

-S <n=v>这个是设置变量值，这可以用来在命令行定义 Snort rules 文件中的变量，如你要在 Snort rules 文件中定义变量 HOME_NET,你可以在命令行中给它预定义值。

-v 使用为 verbose 模式，把信息包打印在 console 中，这个选项使用后会使速度很慢，这样结果在记录多的是时候会出现丢包现象。 对应 pv.verbose_flag,

-V 显示 SNORT 版本并退出；

-6 显示 Ipv6 协议数据包 对应 pv.showipv6_flag

-x 显示 IPX 协议数据包 对应 pv.showipx_flag

-X 转储从数据链路曾直到应用层协议的原始数据包字节（一般不使用）对应 pv.verbose_bytedump_flag;

-? 显示使用列表并退出；

参考文献：

《网络入侵检测系统的设计与实现》 唐正军 等编著 电子工业出版社出版

《Snort 2.0 入侵检测》 宋劲松等译 国防工业出版社出版

《自动排序入侵检测》 林仁杰 逢甲大学资讯工程学系硕士论文

[入侵检测]snort 源码分析

Snort 入侵检测系统