

# **Snort 入侵检测系统**

## **源码分析**

**作者：刘大林**

**时间：2005-02-28**

# Snort 源码分析

前言 .....	6
第 1 章 系统架构总概 .....	7
1. 1 程序总流程图 .....	7
1. 2 主函数代码完全注释 .....	10
1. 3 SnortServiceMain .....	22
1. 4 两个重要的全局变量 .....	26
1. 4. 1 Packet 数据结构 .....	26
1. 4. 2 PV 数据结构 .....	30
第 2 章 系统初始化 .....	33
2. 1 WinSock 初始化 .....	33
2. 2 检测引擎初始化 .....	34
2. 3 命令行解析 .....	35
第 3 章 打开数据截获接口 .....	37
3. 1 LibPcap() 主要库函数简介 .....	37
3. 2 OpenPcap() 函数 .....	37
第 4 章 插件初始化 .....	40
4. 1 输出插件注册与初始化 .....	41
4. 1. 1 重要的数据结构 .....	41
4. 1. 2 注册 .....	42
4. 1. 3 初始化函数 .....	45
4. 1. 4 调用 .....	50
4. 2 预处理插件初始化 .....	50
4. 2. 1 重要的数据结构 .....	50
4. 2. 2 注册 .....	51
4. 2. 3 初始化 .....	54
4. 2. 4 调用 .....	59
4. 3 规则选项关键字插件初始化 .....	60

第 5 章 检测规则初始化引擎.....	64
5. 1 重要的数据结构 .....	64
5. 2 规则初始化流程 .....	68
5. 3 CreateDefaultRules.....	69
5. 4 ParseRuleFile.....	71
5. 5 ParseRule.....	76
5. 5. 1 函数流程 .....	76
5. 5. 2 ProcessIP .....	87
5. 5. 3 ParsePort .....	91
5. 5. 4 mSplit 功能函数 .....	95
5. 6 ProcessHeadNode .....	102
5. 6. 1 函数流程 .....	102
5. 6. 2 规则链表头检测匹配函数列表的配置 .....	114
5. 7 ParseRuleOptions .....	117
5. 7. 1 函数流程分析.....	117
5. 7. 2 ParseMessage .....	132
第 6 章 构建规则快速配匹引擎.....	135
6. 1 概述 .....	135
6. 2 fpCreateFastPacketDetection .....	139
6. 3 prmAddRuleXX .....	143
6. 4 prmxAddPortRuleXX .....	145
6. 5 prmCompileGroups 和 BuildMultiPatternGroups .....	147
6. 6 小结 .....	151
第 7 章 数据包处理.....	153
7. 1 InterfaceThread.....	153
7. 2 ProcessPacket .....	153
7. 3 嗅探模式的终端输出.....	156
7. 4 日志记录模式的输出处理.....	159
7. 5 Preprocess.....	161

7. 6 小结.....	163
第 8 章 数据包解码引擎 .....	165
8. 1 DecodeEthPkt.....	166
8. 2 DecodeIP.....	167
8. 3 DecodeTCP .....	172
第 9 章 预处理插件的工作 .....	175
9. 1 Stream4 TCP 状态维护和会话重组 .....	175
9. 2 frag2 分片重组和攻击检测 .....	175
9. 2. 1 重要的数据结构.....	176
9. 2. 2 frag2 算法分析 .....	179
9. 2. 3 GetFragTracker .....	184
9. 2. 4 NewFragTracker 和 InsertFrag .....	187
9. 2. 5 FragIsComplete .....	191
9. 2. 5 RebuildFrag .....	194
9. 3 BO 后门 .....	194
9. 4 ARP 欺骗检测.....	194
第 10 章 数据包检测引擎 .....	199
10. 1 Detect .....	199
10. 2 fpEvalPacket.....	200
10. 3 fpEvalHeaderTcp .....	202
10. 3. 1 函数流程分析.....	202
10. 3. 2 prmFindRuleGroupTcp.....	203
10. 3. 3 InitMatchInfo .....	206
10. 4 fpEvalHeaderSW .....	208
10. 4. 1 函数流程 .....	208
10. 4. 2 mpseSearch .....	212
10. 4. 3 fpEvalOTN 和 fpEvalRTN .....	213
10. 5 检测事件的输出 .....	215
10. 5. 1 fpAddMatch .....	215

10. 5. 2 fpFinalSelectEvent .....	217
10. 5. 3 SnortEventqAdd.....	218
10. 4. 4 sfeventq_add .....	221
第 11 章    规则链表头的检测函数.....	224
11. 1 概述.....	224
11. 2 CheckBidirectional .....	224
11. 3 端口的检测 .....	229
11. 4 地址的检测 .....	230
第 12 章    规则选项关键字插件的实现.....	232
第 13 章    输出检测结果 .....	245
12. 1 报警日志处理流程 .....	245
12. 2 向 MYSQL 输出报警日志的实现 .....	249
第 14 章    附录.....	250
12. 1 WinPcap 使用入门 .....	250
12. 1. 1 获取网卡列表.....	250
12. 1. 2 获得已安装网络驱动器的高级信息.....	252
12. 1. 3 打开网卡捕获数据包 .....	255
12. 1. 4 不用 loopback 捕获数据报 .....	260
12. 1. 5 数据流的过滤.....	264
12. 1. 6 解析数据包.....	265
12. 1. 7 处理脱机的堆文件.....	273
12. 1. 8 发送数据包.....	281
12. 1. 9 收集并统计网络流量 .....	288
12. 2 在 Wind2000 上搭建 Snort 入侵检测平台.....	294
12. 3 Snort 版本升级历程 .....	294
12. 3 参考资料.....	294

# 前言

与 Snort 的接触很偶然，最初的目的仅仅是想抄袭——在写一个 Win32 平台下的 Sniffer 程序的时候，总感觉自己的拆包函数不够严谨，于是下载了 Snort 的源码，起了 Copy 念头。岂知，这一抄，竟然如此地不能自拔，也许，这也正是开源的魅力所在。

其实，最先面对如此庞然大物之时，我也有退却的时候，以前在面对稍微长一点的程序的时候，就是草草地将看完，再长一点，就没有了耐性。这一次能够坚持下来，最重要的还是兴趣。所以，阅读本书的第一个要求，就是“耐性+兴趣”！

Snort 最初为 Linux 而生，但是写这本书的时候，大多都出差在外，完成本书更多的地方是火车上、机场或者是长途汽车上。因为实在不习惯在 Linux 下编写文档，所以，本书分析的 Snort 源码 是 for Windows 的。Snort 的 Windows 环境的源码其实是 Linux 源码的超集，因此，对于那些想研究 Linux 下源码的朋友，不必担心，明白了 for Windows 的源码，也就理解 for Linux 的源码。基于此，本书调试平台是 Windows2000 Professional SP4 + VC 6.0 +Snort 2.2。

从开始分析 Snort 到本书的完成，前后竟达四个月之久，主要还是因为工作太过繁忙，只能抽一些零碎的时间来完成，所以，我要特别感谢我的女友，没有她的理解与支持，就没有本书的完成！

Snort 做为一个出色的网络入侵检测系统，包含了许多方面的知识，所以在阅读本书前，你应该了解以下前导知识：

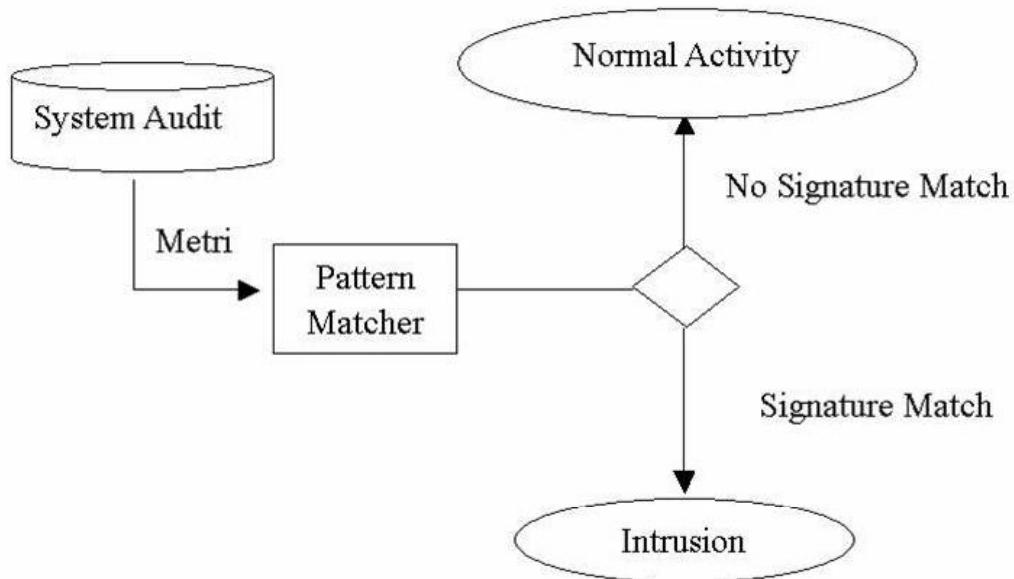
- C 语言基础知识
- 链表、二叉树、Boyer-Moore 等算法
- LibPcap 库函数的调用
- TCP/IP 协议
- Socket 网络编程
- 《Snort 用户手册》——较熟练地应用和配置 Snort

谨以此书献给所有开源的爱好者们！

# 第1章 系统架构总概

## 1. 1 程序总流程图

在我们对 Snort 的源代码进行分析之前，让我们来先了解一下它的大体的运行流程。Snort 是一个基于模式匹配的网络入侵检测系统。所谓模式匹配，是与状态分析相对应的，它是指系统预先定义一些入侵的特征码，然后将实际数据包与特征码相匹配，以判断检测数据包是否包含了一个入侵行为，这些特征码定义在安装目录的 rules 文件夹下。一个基于模式匹配的入侵检测工作流程如下图所示：



所以，一个基于模式匹配的网络入侵检测系统至少包含了以下四个步骤：

- 嗅探网络中的数据包
- 拆包
- 检测引擎，进行规则匹配
- 输出报警或日志信息

除了这四个步骤，Snort 在运行之前，要进行一些初始化配置，比如，读取用户的一些配置文件，分析命令行参数等等（系统初始化）。而系统在规则匹配

之前，需要首先将规则读入内存（初始化规则引擎），同时，为了提高检测效率，需要将规则按一定的方式进行二次分类（构建规则快速匹配引擎）。另外，有一些特殊的数据包，不能直接对数据包进行匹配，如分片包就要先对其进行重组，否则它将绕过 Snort 的检测。这些工作在 Snort 中叫做预处理，需要在检测引擎被调用前进行。

预处理和输出都是以插件的形式存在的，之所以采用插件，一方面方便了第三方开发者定制出适合自己的预处理系统和输出系统，另一方面，提供许多功能不同的预处理插件和输出插件，方便用户根据自己的需要进行有效地选择。既然是以插件的形式存在，那么在实际调用之前，就需要先进行初始化（判断用户使用了哪些功能插件，配置每个插件的一些参数）。

如上分析，系统可以共分为两个阶段：

- 初始化：完成检测数据包前的所有准备工作；
- 数据包处理：完成数据包从嗅探到报日志输出等主要的工作；

初始化又可以分为以下五个阶段：

- 前期准备工作 [第二章]
- 打开数据包截获接口 [第三章]
- 插件初始化 [第四章]
- 规则链表初始化 [第五章]
- 规则快速匹配引擎初始化 [第六章]

数据包处理可以分为以下六个阶段：

- 数据包截获 [第七章]
- 数据包解码 [第八章]
- 处理嗅探模式和日志记录模式 [第七章]
- 数据包预处理 [第九章]
- 数据包检测 [第十章]
- 日志及报警的输出 [第十一章]

注：*Snort* 的源码包，你可以在 *Snort* 的官方网站 [www.snort.org](http://www.snort.org) 上得到

程序文件 snort.c 中的 Main() 函数为整个系统入口，主要对 Win32 平台下一些参数进行判断，如输入参数、是否以 Windows 服务进程启动等。如果不是以服务进程方式启动，则使用 return SnortMain() 语句，将程序控制权交由主函数 SnortMain()。SnortMain() 函数构成了整个系统的主体，函数运行流程如下图所示：

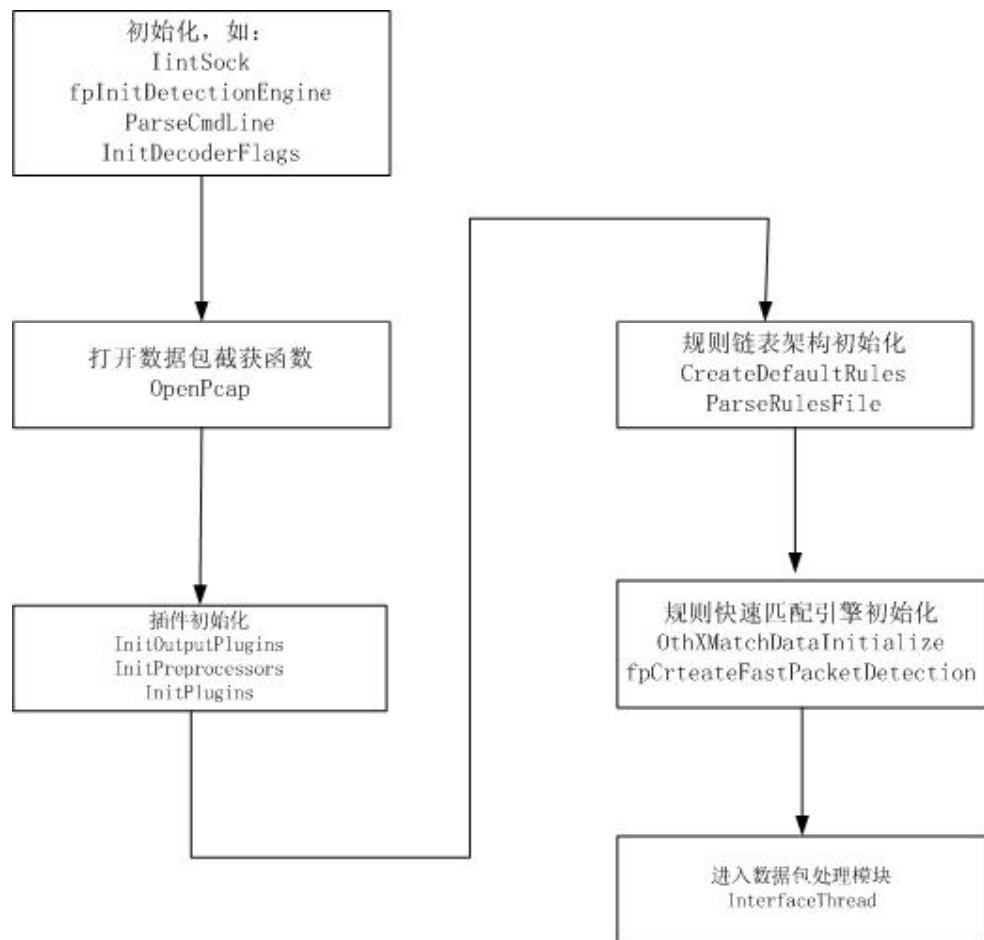


图 1.1 Snort 主流程图

图中的每一个函数完成了系统的每一个阶段的重要功能，以下各个章节将对这些函数进行一一分析和说明。另外需要交待的是 Snort 的数据包截获主要是采用了 LibPcap 库，如果对这一套函数库使用不太熟悉的朋友，可以参考附录第一节。要想了解 Snort 的平台搭建及使用，可以参考附录第二节，要想了解 Snort 的进一步管理及配置，可以参考附录第三节。

## 1. 2 主函数代码完全注释

如上节流程图 1.1 所示，SnortMain 是整个系统的骨架，完成了系统的初始化、包嗅探、抓包、分析及输出等所有工作。源码注释如下：

```
int SnortMain(int argc, char *argv[])
{
#ifndef WIN32
    #if defined(LINUX) || defined(FREEBSD) || defined(OPENBSD) ||
defined(SOLARIS)
        sigset_t set;

        sigemptyset(&set);
        sigprocmask(SIG_SETMASK, &set, NULL);

    #else
        sigsetmask(0);
    #endif
#endif /* !WIN32 */

/*对应于各种进程间信号，设置对应的处理函数.*/
    signal(SIGTERM, SigTermHandler);      if(errno!=0) errno=0;
    signal(SIGINT, SigIntHandler);        if(errno!=0) errno=0;
    signal(SIGQUIT, SigQuitHandler);     if(errno!=0) errno=0;
    signal(SIGHUP, SigHupHandler);       if(errno!=0) errno=0;
    signal(SIGUSR1, SigUsr1Handler);     if(errno!=0) errno=0;

    programe = argv[0];           /*设置两个全局指针指向 argv,argc*/
    progargs = argv;

#endif WIN32             /*如果是 Win32，初始化 Socket*/
```

```

if (!init_winsock())
    FatalError("Could not Initialize Winsock!\n");
#endif

/*清空全局变量 pv， 其存储了系统运行时需要的绝大部分标志信息*/
memset(&pv, 0, sizeof(PV));

/*初始化一个数组， 包含了 A-D 类所有可能被划分子网的掩码*/
InitNetmasks();

/*初始化协议名， 这样， 我们在后面要输出协议名称就不用使用 switch case
语句了 util.c(1005)*/

InitProtoNames();

/* 检测引擎初始化 fpInitDetectinEngine， 用于制定快速规则匹配模块的配置参数， 包
括模式搜索算法等(Snort 可使用的算法有 Aho-Corasick , Wu-Manber , Boyer-Moore 等算法,
缺省 Snort 使用 Byer-More 算法。并负责在协议解析过程中产生警报信息。 */
fpInitDetectionEngine();

/* pv(pkt_cnt 表示总共要捕获包的数数， 初始化为-1， 表示永循环， 也可以
由用户指定数目， 其值在函数 ParseCmdLine (解析命令行) 中设置*/
pv(pkt_cnt = -1;

/* 设置报警文件为空 */
pv.alert_filename = NULL;

/* 设置默认报警模式 */
pv.alert_mode = ALERT_FULL;

/* set the default assurance mode (used with stream 4) */
pv.assurance_mode = ASSURE_ALL;

```

```

pv.use_utc = 0;

pv.log_mode = 0;           /*日志记录模型*/

pv.quiet_flag = 0;         /*安静模式开关，=1 表示不向终端输出任何信息*/

InitDecoderFlags();        /* 打开默认的解码器报警,通常被用于 bug 报告 */

/*打开默认较验和开关*/
pv.checksums_mode = DO_IP_CHECKSUMS | DO_TCP_CHECKSUMS |
                    DO_UDP_CHECKSUMS | DO_ICMP_CHECKSUMS;

#if defined(WIN32) && defined(ENABLE_WIN32_SERVICE)
/*若是以 Win32 服务进程启动，初始化 Win32 服务控制标志*/
pv.terminate_service_flag = 0;
pv.pause_service_flag = 0;
#endif /* WIN32 && ENABLE_WIN32_SERVICE */

ParseCmdLine(argc, argv);      /* 命令行解析 */

/*如果是非 root 权限，则虚拟一个管理权限用户代替*/
if (userid != 0)
    signal(SIGHUP, SigCantHupHandler);

/* 判断 Snort 将运行于哪一种工作模式，即决定全局变量 runMode 的值 ，
pv.config_file 的值是根据 ParseCmdLine 函数从用户终端输入获取的*/
if(pv.config_file)           //如果 Snort 配置文件存在
{
    runMode = MODE_IDS;       //运行于 IDS 模式
    if(!pv.quiet_flag)

```

```

        LogMessage("Running in IDS mode\n");
    }

else if(pv.log_mode || pv.log_dir)      //如果为日志模式，且日志目录存在
{
    runMode = MODE_PACKET_LOG;          //运行于 LOG 模式
    if(!pv.quiet_flag)

        LogMessage("Running in packet logging mode\n");

    }

else if(pv.verbose_flag)              //如果为包记录模式， -向终端输出
{
    runMode = MODE_PACKET_DUMP;        //运行于嗅探模式
    if(!pv.quiet_flag)

        LogMessage("Running in packet dump mode\n");

    }

else if((pv.config_file = ConfigFileSearch()) ) //是否能在默认目录查到
Snort 配置文件?

{
    runMode = MODE_IDS;                //查找到了， 运行于 IDS 模式
    if(!pv.quiet_flag)

        LogMessage("Running in IDS mode with inferred config file: %s\n",
                    pv.config_file);

    }

else
{
    /*不能确定运行于何种模式下， 输出 Banner 信息， 退出*/
    DisplayBanner();
    ShowUsage(progname);
    PrintError("\n\nUh, you need to tell me to do something...\n\n");
    exit(1);
}

```

```

/* 如果 pv.log_dir 没有被设置，则设置默认的目录路径 */
if(!pv.log_dir)
{
    /* strdup() 返回指向被复制的字符串的指针，所需空间由 malloc()分配且
可以由 free()释放。 */
    if(!(pv.log_dir = strdup(DEFAULT_LOG_DIR)))
        FatalError("Out of memory setting default log dir\n");
}

if(!pv.quiet_flag)
{
    LogMessage("Log directory = %s\n", pv.log_dir); //向终端输出目录路径
}

/* 确认日志目录 */
if(runMode == MODE_IDS || runMode == MODE_PACKET_LOG)
{
    SanityChecks(); /* 日志目录存在且可写 */
}

/*如果运行于 LOG 模式，日志模式未设置，则设置日志模式*/
if(runMode == MODE_PACKET_LOG && !pv.log_mode)
{
    pv.log_mode = LOG_ASCII;
}

/* 判断是从网络上嗅探还是从文件中读取 */
if(!pv.readmode_flag)          //如果是从网络嗅探
{

```

```

DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Opening interface:
%s\n",
PRINT_INTERFACE(pv.interface)););

/* 初始化 libpcap, 如选择并打开网卡, 设置过滤规则等等 */
OpenPcap();

}

else //从 Tcpdump 文件读取
{

DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Opening file: %s\n",
pv.readfile));;

/* 从文件回读 */
OpenPcap();

}

/*从配置文件名中提取出配置目录*/
if(pv.config_file)
{
/* 分隔符为 '/' */
if(strrchr(pv.config_file,'/'))
{
char *tmp;
/* lazy way, we waste a few bytes of memory here */
if(!(pv.config_dir = strdup(pv.config_file)))
    FatalError("Out of memory extracting config dir\n");

tmp = strrchr(pv.config_dir,'/');
*(++tmp) = '\0';

}

```

```

else      /*Win32 下，'\\'也被认为是合法的*/
{
#endif WIN32

/* is there a directory seperator in the filename */
if(strrchr(pv.config_file,'\\'))
{
    char *tmp;

    /* lazy way, we waste a few bytes of memory here */
    if(!(pv.config_dir = strdup(pv.config_file)))
        FatalError("Out of memory extracting config dir\n");

    tmp = strrchr(pv.config_dir,'\\');
    *(++tmp) = '\0';
}

else
#endif

if(!(pv.config_dir = strdup("./")))
    FatalError("Out of memory extracting config dir\n");
}

DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Config file = %s, config
dir = "
"%s\n", pv.config_file, pv.config_dir););
}

/* XXX do this after reading the config file? */
if(pv.use_utc == 1)
{
    thiszone = 0;
}
else

```

```

{
    /* set the timezone (ripped from tcpdump) */
    thiszone = gmt2local(0);
}

if(!pv.quiet_flag)
{
    LogMessage("\n          --== Initializing Snort ==--\n");
}

/*运行于 IDS 模式，且规则顺序要改变*/
if(runMode == MODE_IDS && pv.rules_order_flag)
{
    if(!pv.quiet_flag)
    {
        LogMessage("Rule      application      order      changed      to
Pass->Alert->Log\n");
    }
}

/*
如果需要程序运行于守护进程*/
if(pv.daemon_flag)
{
    DEBUG_WRAP(DebugMessage(DEBUG_INIT,      "Entering      daemon
mode\n););
    GoDaemon();      /*处理系统为守护进程的情况*/
}

InitOutputPlugins();      /*初始化输出插件*/

```

```

/* 创建 PID 文件 */
if((runMode == MODE_IDS) || pv.log_mode || pv.daemon_flag
    || *pv.pidfile_suffix)
{
    /* ... then create a PID file if not reading from a file */
    if (!pv.readmode_flag && (pv.daemon_flag || *pv.pidfile_suffix))
    {
        #ifndef WIN32
            CreatePidFile(pv.interface);
        #else
            CreatePidFile("WIN32");
        #endif
    }
}

DEBUG_WRAP(DebugMessage(DEBUG_INIT,          "Setting      Packet
Processor\n));
```

/\* 关连解码函数 主要使用了一个函数指针，并使用 datalink 的值来判断\*/

**SetPktProcessor();**

/\* 系统运行于 IDS 模式下，需要用到检测规则，初始化其先…… \*/

```

if(runMode == MODE_IDS)
{
    SanityChecks();

    /* 初始化所有的插件模块 */
    InitPreprocessors(); /*初始化预处理插件*/
    InitPlugIns();      /*规则选项关键字插件模块的初始化*/
    InitTag();
}
```

```

#ifndef DEBUG
    DumpPreprocessors();
    DumpPlugIns();
    DumpOutputPlugins();
#endif

/* 建立规则链表，不过只是搭建框架，不填充内容，待进一步调用
parseRulesFile 来填充 */
CreateDefaultRules();

if(pv.rules_order_flag)
{
    OrderRuleLists("pass activation dynamic alert log");
}

if(!pv.quiet_flag)
    LogMessage("Parsing Rules file %s\n", pv.config_file);

/*读取配置文件及规则文件，填充规则链表、输出插件、输入插件等
等.....*/
ParseRulesFile(pv.config_file, 0);

OtnXMatchDataInitialize();           //初始化全局数据结构 omd

FlowBitsVerify();

asn1_init_mem(512);

/*

```

```

    ** Handles Fatal Errors itself.

    */

SnortEventqInit();

if(!pv.quiet_flag)
{
    print_thresholding();

    printRuleOrder();           //输出规则顺序信息
}

#endif WIN32

/* Drop the Chrooted Settings */
if(pv.chroot_dir)
    SetChroot(pv.chroot_dir, &pv.log_dir);

/* Drop privileges if requested, when initialization is done */
SetUidGid();

#endif /*WIN32*/

/*
*如果系统工作于 IDS 模式并且命令行中没有设置报警选项或者我们没有
*激活报警插件，则设置它们。
*/
if(runMode == MODE_IDS &&
    (pv.alert_cmd_override || !pv.alert_plugin_active))
{

```

```

    ProcessAlertCommandLine();

}

if((runMode == MODE_IDS || runMode == MODE_PACKET_LOG) &&
   (pv.log_cmd_override || !pv.log_plugin_active))
{
    ProcessLogCommandLine();
}

/* 规则快速匹配检测引擎, 主要是在规则链表 RTN 中建立一个快速包分级
数组。 */

fpCreateFastPacketDetection();

if(!pv.quiet_flag)
{
    mpsePrintSummary();
}

if(!pv.quiet_flag)
{
    LogMessage("\n          --== Initialization Complete ==-\n");
}

/* Tell 'em who wrote it, and what "it" is */

if(!pv.quiet_flag)
    DisplayBanner();

if(pv.test_mode_flag)
{
    LogMessage("\nSnort sucessfully loaded all rules and checked all rule "

```

```

        "chains!\n");
    CleanExit(0);
}

if(pv.daemon_flag)
{
    LogMessage("Snort initialization completed successfully\n");
}

DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Entering pcap loop\n"));

InterfaceThread(NULL); //完成嗅探->解码->检测->报警等工作

return 0;
}

```

这个主函数还是比较长的，读者可以对照图 1.1 了解其流程和组成的模块，这些模块都将在接下来的章节一一介绍。除去模块，大都是一些标志位的设置，这些重要的标志位大都包含在一个重要的全局变量当中。读者可以对照 1.4 节的内容，一一分析这些标志位的作用。

## 1. 3 SnortServiceMain

如果用户决定以 Win32 进程方式启动 Snort，则 main 函数将控制权交由 SnortServiceMain (Win32\_server.c)，如果你对 Snort 在 Win32 下细节不感兴趣，可以跳过本节。

SnortServiceMain 函数首先填充一个 SERVICE\_TABLE\_ENTRY 结构数组 steDispatchTable：

```
SERVICE_TABLE_ENTRY steDispatchTable[] =  
{  
    { g_lpszServiceName, SnortServiceStart },  
    { NULL, NULL }  
};
```

steDispatchTable[0][0]是服务的名字

steDispatchTable[0][1]一个指向服务主程序的函数指针

这个结构做为启动服务进程函数 StartServiceCtrlDispatcher 的参数使用。

然后分析命令行参数，判断是安装、卸载还是显示服务进程参数信息：

```
InstallSnortService(argc, argv); //安装服务  
UninstallSnortService(); //卸载服务  
ShowSnortServiceParams(); //显示参数信息
```

如果三者都不是，则启动服务：

```
if (!StartServiceCtrlDispatcher(steDispatchTable))
```

API 函数 StartServiceCtrlDispatcher 为每一个传递到它的数组中的非空元素产生一个新的线程，每一个进程开始执行由数组元素中的 steDispatchTable 指明名称和实际执行函数。

### **InstallSnortService**

InstallSnortService 函数的作用是安装一个服务，主要是使用 API 函数 CreateService 来实现。

在调用 CreateService 安装之前，先读写注册表，将命令行参数写入注册表。

然后调用 API 函数

```
schSCManager = OpenSCManager(NULL,  
                           NULL,  
                           SC_MANAGER_ALL_ACCESS);
```

来得到 CreateService 需要的 SCM 句柄 (schSCManager)。接着就开始安装服务：

```
schService = CreateService( schSCManager, /* SCM 句柄 */
```

```

        g_lpszServiceName,           /* 服务名*/
        g_lpszServiceDisplayName,   /* 服务显示的名称 */
        SERVICE_ALL_ACCESS,         /* 访问权限*/
        SERVICE_WIN32_OWN_PROCESS,  /* SERVICE 类
型: 独享或共享 */
        SERVICE_DEMAND_START,      /* 启动类型*/
        SERVICE_ERROR_NORMAL,       /* 出错时采取什么
动作*/
        lpszBinaryPathName,         /* Service 实际执行的
程序的路径名 */
        NULL,                      /* no load ordering group*/
        NULL,                      /* no tag identifier */
        NULL,                      /* no dependencies*/
        NULL,                      /* LocalSystem account*/
        NULL);                    /* no password */

```

如果 CreateService 函数返回非空, 则注册成功, 否则, 根据相应的返回信息, 进行相关处理。

最后, 调用

```

CloseServiceHandle(schService);
CloseServiceHandle(schSCManager);

```

关闭句柄, 完成安装。

### **UninstallSnortService**

服务的卸载主要是调用 API 函数 OpenService 来实现的, 其流程与安装相似。不再赘述。

### **ShowSnortServiceParams**

显示参数的实现则简单了许多, 它直接调用 ReadServiceCommandLineParams 函数从注册表中将参数信息读取出来, 然后输出即是。源码如下:

```

VOID ShowSnortServiceParams()
{
    int      argc;
    char ** argv;
    int i;

    ReadServiceCommandLineParams( &argc, &argv );           //读取命令行参数

    printf("\n"
           "Snort is currently configured to run as a Windows service using the
following\n"
           "command-line parameters:\n\n"
           "");

    for( i=1; i<=argc; i++ )
    {
        if( argv[i] != NULL )
        {
            printf(" %s", argv[i]);
            free( argv[i] );
            argv[i] = NULL;
        }
    }

    free( argv );
    argv = NULL;

    printf("\n");
}

```

注：关于更多 Win32 服务进程控制的相关内容，请参考 MSDN。

## 1. 4 两个重要的全局变量

Snort 中包含了许多重要的全局变量和数据结构。其中又以 Packet 数据结构和 PV 数据结构最为庞大和重要。

### 1. 4. 1 Packet 数据结构

Snort 系统中最重要的数据结构非 Packet 结构莫属，Packet 数据结构 (decode.h) 控制着整个系统正常工作的关键信息，定义如下：

```
typedef struct _Packet
{
    struct pcap_pkthdr *pkth; /* BPF data ,libpcap 捕获包的包头*/
    u_int8_t *pkt;           /* 指向捕获到的数据包*/

/*以下结构对应了 Snort 支持的相关协议*/
    Fddi_hdr *fddihdr;       /* FDDI support headers */
    Fddi_llc_saps *fddisaps;
    Fddi_llc_sna *fddisna;
    Fddi_llc_iparp *fddiiparp;
    Fddi_llc_other *fddiother;

    Trh_hdr *trh;           /* Token Ring support headers */
    Trh_llc *trhllc;
    Trh_mr *trhmr;

    SLLHdr *sllh;           /* Linux cooked sockets header */

    PflogHdr *pfh;           /* OpenBSD pflog interface header */

    EtherHdr *eh;             /* standard TCP/IP/Ethernet/ARP headers
```

```

*/



VlanTagHdr *vh;
EthLlc    *ehllc;
EthLlcOther *ehllcother;

WifiHdr *wifih;           /* wireless LAN header */

EtherARP *ah;

EtherEapol *eplh;         /* 802.1x EAPOL header */
EAPHdr *eaph;
u_int8_t *eaptype;
EapolKey *eapolk;

IPHdr *iph, *orig_iph;   /* and orig. headers for ICMP_*_UNREACH
family */

u_int32_t ip_options_len;
u_int8_t *ip_options_data;

TCPHdr *tcp, *orig_tcp;
u_int32_t tcp_options_len;
u_int8_t *tcp_options_data;

UDPHdr *udph, *orig_udph;
ICMPHdr *icmph, *orig_icmph;

echoext *ext;             /* ICMP echo extension struct */

u_int8_t *data;            /* packet payload pointer */
u_int16_t dsiz;            /* packet payload size */

```

```
u_int16_t alt_dsize; /* the dsize of a packet before munging  
                      (used for log)*/  
/*这一组这五个主要用于拆包的时候处理 IP 分片，我们讲 frag2 插件时  
会讲到*/
```

```
u_int8_t frag_flag; /* flag to indicate a fragmented packet */  
u_int16_t frag_offset; /* fragment offset number */  
u_int8_t mf; /* more fragments flag */  
u_int8_t df; /* don't fragment flag */  
u_int8_t rf; /* IP reserved bit */
```

```
u_int16_t sp; /* source port (TCP/UDP) */  
u_int16_t dp; /* dest port (TCP/UDP) */  
u_int16_t orig_sp; /* source port (TCP/UDP) of original datagram */  
u_int16_t orig_dp; /* dest port (TCP/UDP) of original datagram */  
u_int32_t caplen;
```

```
u_int8_t uri_count; /* number of URIs in this packet */
```

```
void *ssnptr; /* for tcp session tracking info... */  
void *flow; /* for flow info */  
void *streamptr; /* for tcp pkt dump */
```

```
Options ip_options[40]; /* ip options decode structure */  
u_int32_t ip_option_count; /* number of options in this packet */  
u_char ip_lastopt_bad; /* flag to indicate that option decoding was  
                           halted due to a bad option */
```

```
Options tcp_options[TCP_OPTLENMAX]; /* tcp options decode struct  
*/
```

```
u_int32_t tcp_option_count;  
u_char tcp_lastopt_bad; /* flag to indicate that option decoding was
```

```

        halted due to a bad option */

/*csum_flags 指示各种协议报文头的校验出错情况*/
u_int8_t csum_flags;

/*指示当前数据包所处的状态*/
u_int32_t packet_flags;

/*用于标识需要调用的预处理器的类型*/
int preprocessors;

} Packet;

```

可见，Packet 结构共分三类：

第一类：指示原始数据包截获信息的字段，包括\*pkth 和\*pkt 指针字段。关于它们的使用，你阅读完本书附录中关于 libpcap 的使用，就一目了然了；

第二类：用于存放当前数据包进行协议解析后所得信息的字段。这一类也是最为繁多的。但是它们都对应了相应的协议，只要了解了对应的协议的包的结构，要了解它们就非常容易了；

第三类：最后三个是标识字段。csum\_flags 的取值范围定义在 checksum.h 中：

```

/* define checksum error flags */

#define CSE_IP      0x01
#define CSE_TCP     0x02
#define CSE_UDP     0x04
#define CSE_ICMP    0x08
#define CSE_IGMP    0x10

```

packet\_flags 的取值范围定义在 decode.h 中：

```

#define PKT_REBUILT_FRAG      0x00000001 /* is a rebuilt fragment */
#define PKT_REBUILT_STREAM    0x00000002 /* is a rebuilt stream */
#define PKT_STREAM_UNEST_UNI 0x00000004
.....

```

preprocessors 的初始值是 PP\_ALL。可选的设置赋值定义在 plugbase.h 中：

```
#define PP_ALL
```

-1

```
#define PP_LOADBALANCING           1  
#define PP_PORTSCAN                 2  
#define PP_HTTP_DECODE              4  
.....
```

## 1. 4. 2 PV 数据结构

PV 数据结构的重要性主要体现在其存储了系统配置信息和命令行参数解析后的结果信息，从而对系统的众多控制标识符的设置起着重要的作用。

PV 数据结构定义在 snort.h 中，这里给出其中大部份成员的解释，另外一些，将在后文的分析当中提及：

```
typedef struct _progvars  
{  
    int stateful;  
    int line_buffer_flag;  
    int checksums_mode;  
    int assurance_mode;  
    int max_pattern;  
    int test_mode_flag;  
    int alert_interface_flag;  
    int verbose_bytedump_flag; /*转储详细各层协议原始数据包字节信息的标志符，对应于参数 -X*/  
    int obfuscation_flag; /*隐藏日志文件中 IP 地址的标志符，对应于参数 -O*/  
    int log_cmd_override; /*命令行日志选项的覆盖标志，表明此选项覆盖规则文件中的对应项设置*/  
    int alert_cmd_override; /*命令行警报选项的覆盖标志，表明此选项覆盖规则文件中的对应项设置*/  
    int char_data_flag; /*仅转储数据包载荷中 ASCII 字符的标志符，对应于命令行参数 -C*/
```

```

int data_flag;           /*转储应用层协议数据包的标志符，对应于命令行参数-d*/
int verbose_flag;        /*设置详细模式开关，对应命令参数 -v*/
int readmode_flag;       /*从文件中读取而不是从网络中嗅探数据包的标志符
*/
int show2hdr_flag;        /*显示第二层数据包报头信息的开关*/
int showwifimgmt_flag;

#ifndef WIN32

int syslog_remote_flag;
char syslog_server[STD_BUF];
int syslog_server_port;

#endif /* ENABLE_WIN32_SERVICE */ /*Win32 服务控制标志*/

int terminate_service_flag;
int pause_service_flag;

#endif /* ENABLE_WIN32_SERVICE */

#endif /* WIN32 */

int promisc_flag;         /*混杂模式开关， 默认为 1， 表示工作为混杂模式，若设置
参数 -p，则关闭*/
int rules_order_flag;     /*改变规则处理顺序的标志符*/
int track_flag;
int daemon_flag;          /*设置守护进程模式的标志符， 对应于命令行参数-D*/
int quiet_flag;           /*安静模式开关*/
int pkt_cnt;              /*所需处理的数据包的总数。=1， 表示进入无限循环，不限处
理数目*/
int pkt_snaplen;          /*截获数据包的长度， 默认值为 SNAPLEN*/
u_long homenet;           /*以二进制形式表示的本地网络地址*/
u_long netmask;           /*以二进制形式表示的本地掩码地址*/
u_int32_t obfuscation_net;
u_int32_t obfuscation_mask;
int alert_mode;            /*报警模式标志符*/
int log_plugin_active;

```

```

int alert_plugin_active;

u_int32_t log_bitmap;

char pid_filename[STD_BUF];

char *config_file;      /*配置文件*/
char *config_dir;       /*配置文件目录*/
char *log_dir;          /*日志目录*/

char readfile[STD_BUF];

char pid_path[STD_BUF];

char *interface;

char *pcap_cmd;

char *alert_filename;   /*报警文件名*/

char *binLogFile;

int use_utc;           /*使用 UTC 时间代替本地系统时间，对应于参数 -U*/
int include_year;

char *chroot_dir;

u_int8_t min_ttl;

u_int8_t log_mode;     /*日志记录模式，可以是 LOG_PCAP 等*/
int num_rule_types;

char pidfile_suffix[MAX_PIDFILE_SUFFIX+1]; /* room for a null */

DecoderFlags decoder_flags; /* if decode.c alerts are going to be enabled */

#endif NEW_DECODER

char *daq_method;

char *interface_list[MAX_IFS];

int interface_count;

char *pcap_filename;

char *daq_filter_string;

#endif // NEW_DECODER

} PV;

```

# 第2章 系统初始化

系统的初始化的工作有很多，本章先介绍一些 Snort 在执行 OpenPcap 函数打开用于截包的网卡之前的一些初始化工作，如完成 Sock 初始化、检测引擎初始化、命令行解析等工作。

## 2. 1 WinSock 初始化

**Init\_winsock 函数：**如果是 Windows 平台，在调用 Socket API 之前，必须调用 WSAStartup 函数初始化 WinSock：

```
int init_winsock(void)
{
    WORD wVersionRequested = MAKEWORD(1, 1);
    WSADATA wsaData;

    //调用 WSAStartup 函数进行初始化
    if (WSAStartup(wVersionRequested, &wsaData))
    {
        FatalError("(!) ERROR: Unable to find a usable Winsock.\n");
        return 0;
    }

    //Socket 版本判断，如果版本小于 1.1，则退出
    if (LOBYTE(wsaData.wVersion) < 1 || HIBYTE(wsaData.wVersion) < 1)
    {
        FatalError("(!) ERROR: Unable to find Winsock version 1.1 or greater. You have
version %d.%d.\n",
                  LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));
        WSACleanup();
        return 0;
    }
}
```

```
    return 1;  
}
```

注：关于更多 WinSock 编程的内容，请参考《Windows 网络编程》或其它相关资料

## 2. 2 检测引擎初始化

**fpInitDetectionEngine** 函数用于配置检测引擎的一些默认参数，也就是初始化 FPDETECT 结构。

结构 FPDETECT 用来记录检测引擎的配置选项定义如下 (fpcreate.h)：

```
/*  
** This structure holds configuration options for the  
** detection engine.  
*/  
  
typedef struct _FPDETECT {
```

```
    int inspect_stream_insert;  
    int search_method;  
    int debug;  
    int max_queue_events;
```

```
} FPDETECT;
```

函数执行流程如下：

- 1、对 fpDetect (static FPDETECT fpDetect, 定义于 fpCreate.c) 赋初值；
- 2、调用 fpSetDetectionOptions 函数，将值传递给定义于 fpDetect.c 中的 fpDetect (static FPDETECT \*fpDetect)。

源代码分析：

```
int fpInitDetectionEngine()  
{
```

```

memset(&fpDetect, 0x00, sizeof(fpDetect));      //初始化 fpDetect 结构

//设置检测引擎的初始化配置选项
fpDetect.inspect_stream_insert = 1;
fpDetect.search_method = MPSE_MWM;
fpDetect.debug = 0;
fpDetect.max_queue_events = 5;

fpSetDetectionOptions(&fpDetect);

return 0;
}

```

函数 **fpSetDetectionOptions** 接受初始化引擎函数传递过来的值，赋给 **fpDetect** 指针，用于实际检测时调用：

```

int fpSetDetectionOptions(FPDETECT *detect_options)
{
    fpDetect = detect_options;
    return 0;
}

```

## 2. 3 命令行解析

**ParseCmdLine** 函数负责命令行解析，在分析该函数之前，可以先参考附录《Snort 用户手册》，了解 Snort 的各个参数的含义，以方便理解该函数。

```

ParseCmdLine 算法比较简单，主要利用
While()
{
    Switch()

```

Case .....

}

语句，利用函数 getopt() 循环分析读取命令行参数，交与 ch，进行匹配。以此设置 pv 各标志变量的值，读者可以对照 Snort 用户手册，分析出 pv 的该部份的参数的含义。

# 第3章 打开数据截获接口

## 3. 1 LibPcap() 主要库函数简介

LibPcap 是一套最初用于 Linux 的数据包用户级截获驱动，其 Windows 版本为 WinPcap。

注：在 Windows 下使用 WinPcap，可能存在一些与 Snort 的兼容性问题，Snort 的安装目录 doc 中，README.WIN32 文件对此已有比较详细的说明。

Snort 使用了这套驱动库做为数据包的截获引擎，这里介绍 Snort 使用了的 LibPcap 的函数。

pcap\_lookupdev( )：查找第一个可以使用的网络适配器；

pcap\_open\_live( )：获取适配器的描述信息；

pcap\_open\_live( )：打开网卡；

pcap\_open\_offline( )：从文件读取数据包；

pcap\_snapshot( )：获取数据链路层的类型；

pcap\_compile( )，pcap\_setfilter( )：设置过滤规则；

pcap\_loop( )：捕获数据包；

关于更多 LibPcap 的信息，请参考本书附录的相关章节；

## 3. 2 OpenPcap() 函数

OpenPcap 函数的主要作用是调用上节所介绍的函数，完成了包截获引擎在截获数据包前的所有工作，如查找网卡、打开网卡，设置过滤规则等等。其需要注意的有两点：

1、Snort 的数据包的获取，有两种来源：从网络中获取或是从文件中获取。主函数中，ParsaCmdLine 根据用户的输入设定了 pv.readmode\_flag 的值，在这里，再

通过 pv.readmode\_flag 来判断：

```
if(!pv.readmode_flag)           /*先根据标志变量判断*/
{
    /*开始进行数据包捕捉*/
    OpenPcap();
}
else
{
    /* open the packet file for readback */
    OpenPcap();
}
```

OpenPcap 函数中，同样地判断：

```
/* if we're not reading packets from a file */
if(pv.interface == NULL)           /*还没有定义接口*/
{
    if (!pv.readmode_flag)         /*根据标志位判断数据获取模式*/
    {
        /*打开首个可用网卡 */
        pv.interface = pcap_lookupdev(errorbuf);

        if(pv.interface == NULL)      /*如果没有可用网卡*/
        {
            FatalError("OpenPcap() interface lookup: \n\t%s\n",
                         errorbuf);
        }
    }
    else                          /*如果是从文件读取*/
    {
        pv.interface = "[reading from a file]";
    }
}
```

```
    }  
}
```

2. 第二个需要留意的地方是 pcap\_snapshot 函数的调用：

```
    snaplen = pcap_snapshot(pd);
```

函数的作用是获取数据链路层的协议类型，置于全局变量 snaplen 中。在后面，主函数会调用 SetPktProcessor 函数来根据 snaplen 的值判断调用何种对应的拆包函数。关于 SetPktProcessor 函数的分析，请参考第八章。

这里仅仅是打开数据截获接口，即执行寻找网卡、使网卡工作于混杂模式、设置数据包过滤规则等等功能，并没有真正地执行抓包。要等到系统初始化完成插件、载入规则等完成后，再调用 pcap\_loop 函数抓包。相关内容，请参考第七章。

## 第4章 插件初始化

大量使用插件来完成工作，是 Snort 的一个主要特点，也是 Snort 模块化结构设计的标志。这样做呢，主要有两个好处：

- 用户可以灵活地选择使用哪些功能；
- 开发人员很容易就可以开发出第三方插件，扩展 Snort 的功能。

Snort 涉及的插件主要有三类：

- 日志输出插件
- 预处理插件
- 检测插件

理解 Snort 中插件的实现，是了解整个系统的一个重点。Snort 的插件系统，在算法上，全部都采用了链表来实现。

插件的工作，主要都可以分为三步：注册->初始化->调用；

**注册：**将系统支持的所有插件建立成一个链表，将插件名同对应的初始化函数进行简单地封装，这种封装机制与 Windows 的消息机制比较类似；

**初始化：**读取 Snort.conf 关于插件的配置，对用户打开的插件设用注册时封装的初始化函数进行初始化，并进一步分析、填充一些数据结构，并将它们的实际功能函数构建成一个链表；

**调用：**遍历插件链表，调用所有的将被使用的插件；

对应这三步，每一类插件也有三类函数：注册函数、初始化函数和实际执行的功能函数。

我们以输出插件为例，再来重复这三个步骤：假设 Snort 支持十种输出插件，系统首先调用注册函数将支持的这十种插件的名称和它们对应的初始化函数封装起来，构建成一个链表。然后读取 Snort.conf 中的配置，判断用户打开了哪些插件，比如用户使用了两个输出插件，则将这两个插件的名称与之前建立的链表

的十个节点中的插件名称进行匹配，找到这两个插件对应的链表节点，以确定用户打开的插件是系统所支持的，并同时也找到了对应的初始化函数。然后，调用之前封装的初始化函数，对使用的这两个插件配置参数进行读取、分析，存储入对应的结构变量中。并将它们的实际的功能执行函数构建成一个链表，以供后面系统来调用。

## 4. 1 输出插件注册与初始化

输出插件的注册的思路主要是首先把系统支持的所有的插件名和与之对应的初始化函数封装，构建成一个链表，以留待系统的进一步调用。要了解注册的机制，只需了解链表的节点的数据结构即可。

每一类输出插件的实现你可以在文件夹 Output Plugins 下找到。

注：按照系统的运行流程，*Snort* 运行到 *InitOutputPlugins()* 函数时，仅仅是注册，而初始化要到后面进行规则解析时才“顺便”来完成，这里将初始化提上来，统一分析输出插件，其它插件也是如此。

### 4. 1. 1 重要的数据结构

#### **OutputKeywordList**

插件链表的节点，封装了关键字节点（即每一类输出插件的相关信息，定义为 **OutputKeywordNode**）：

```
typedef struct _OutputKeywordList
{
    OutputKeywordNode entry;           //关键字节点
    struct _OutputKeywordList *next;   //指向下一个链表节点
} OutputKeywordList;
```

#### **OutputKeywordNode**

关键字节点，包含了插件名称、类型和指向实际执行初始化的函数的指针，可以看到，所谓的封装，也就是在这个结构中实现的：

```

typedef struct _OutputKeywordNode
{
    char keyword;           /*插件名*/
    char node_type;         /*节点类型*/
    void (*func)(char *);   /*该插件对应的初始化函数*/
} OutputKeywordNode;

```

## 4. 1. 2 注册

主函数通过调用 InitOutputPlugins() 函数 (plugbase.c) 来完成输出插件的注册工作。如上文所述，注册是将 Snort 支持的所有输出插件建成一个输出插件链表的过程，链表的每一个节点主要包含了插件名、插件类型和插件对应的初始化函数。其主要通过调用每一种插件对应的 XXX Setup() 函数来实现注册的：

```

void InitOutputPlugins()
{
    if(!pv.quiet_flag)
    {
        LogMessage("Initializing Output Plugins!\n");
    }

    AlertSyslogSetup();      //记录到 syslog
    LogTcpdumpSetup();       //以 TCPDump 格式输出日志文件
    DatabaseSetup();         //输出到数据库，如 Mysql、MS SQL 等等
    AlertFastSetup();        //快速格式
    AlertFullSetup();        //完整报警模式

#ifndef WIN32
/* Win32 doesn't support AF_UNIX sockets */

    AlertUnixSockSetup();  //套接字
#endif /* !WIN32 */

    AlertCSVSetup();         //CSV 格式，允许报警数据以一种易于导入到数据库或电子
                            //表格中的格式输出

```

```

LogNullSetup();           //允许通过报警工具报警，但不产生任何日志文件

UnifiedSetup();          //二进制格式记录

LogAsciiSetup();

#ifndef LINUX

/* This uses linux only capabilities */

AlertSFSocket_Setup();

#endif

}

```

注：对照 *Snort* 的用户手册，我们可以很容易地了解这些 *XXXSetup* 函数所对应的输出插件。

注册函数 *XXX Setup* 又通过调用 *RegisterOutputPlugin* 来注册实现，例如：

```
RegisterOutputPlugin("alert_syslog", NT_OUTPUT_ALERT, AlertSyslogInit);
```

*RegisterOutputPlugin* 的参数对应了数据结构 *sturct \_OutputKeywordNode*:

- Char \*keyword : 该插件所对应的关键字标识；
- Char node\_type : 输出插件的类型、例如报警、日志记录等；
- Void (\*func)(char \*) : 指向初始化函数的指针；

**RegisterOutputPlugin** 负责了链表的搭建，其算法思想为，判断链表是否为空，为空则新建链表和头节点，否则，在链表尾添加一个新节点。源码分析如下：

```

void RegisterOutputPlugin(char *keyword, int type, void (*func) (u_char *))

{
    OutputKeywordList *idx;

    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,"Registering keyword:output =>
% s:% p\n",
                           keyword, func););

```

```

idx = OutputKeywords; /*指向输出插件链表*/

if(idx == NULL)           /*首节点为空*/
{
    /* 为首节点分配空间 */
    OutputKeywords = (OutputKeywordList *) calloc(sizeof(OutputKeywordList),
                                                   sizeof(char));

    idx = OutputKeywords;      /*idx 指向分配了空间的首节点*/
}

else
{
    /* 如果不为空，则循环到末尾添加之 */
    while(idx->next != NULL)
    {
        if(!strcasecmp(idx->entry.keyword, keyword))          /*判断插件被重复注
册*/
        {
            FatalError("%s(%d) => Duplicate output keyword!\n",
                        file_name, file_line);
        }

        idx = idx->next;
    }
}

/*已到末尾，为新节点分配空间，添加之*/
idx->next = (OutputKeywordList *) calloc(sizeof(OutputKeywordList),
                                           sizeof(char));

idx = idx->next;

```

```

}

/* 为下层关键字节点分配空间 */
idx->entry.keyword = (char *) calloc(strlen(keyword) + 1, sizeof(char));

/* 注册关键字 */
strncpy(idx->entry.keyword, keyword, strlen(keyword)+1);

/*type 表明，这是一个 Alert 还是一个 Log*/
idx->entry.node_type = (char) type;

/* 注册插件初始化函数，由规则解析模块负责调用该初始化函数来解析插件函数
*/
idx->entry.func = (void *) func;
}

```

## 4. 1. 3 初始化函数

### 4. 1. 3. 1 概述

注册完成后，下一步就是初始化了。为什么插件需要初始化？这是因为：

- 1、虽然 Snort 支持了很多插件，但是并不是每一种输出插件都会被用到了，要判断用户在 snort.conf 中究竟打开了哪些插件；
- 2、每一种插件都有对应的各种配置参数，需要对这些参数进行分析；

初始化的工作，首先是读取用户在 snort.conf 中的配置，看用户开启哪些插件，这样，就遍历注册时构建的链表，找到这些将被开启的插件所对应的节点，因为插件名和该插件的初始化函数已经被封装，所以，很容易就可以调用此插件的初始化函数。初始化函数主要完成两件工作：

- 1、读取相关插件的配置参数；
- 2、将插件名和实际处理函数建立到一个链表中，以供系统后面调用。事

实际上，输出插件有两个链表：LOG 和 ALERT；

#### 4. 1. 3. 2 规则字符串查找分析

输出插件的定义都在 snort.conf 文件中，主函数通过调用规则拆分函数 ParseRule 来读取 snort.conf 相关配置语句，实现输出插件初始化工作（关于 ParseRule 函数的更多详细分析，请参考规则解析相关章节）。

在函数 ParseRule (parser.c) 中：

```
....  
switch(rule_type)  
{  
    //如果关键字是 output 输出插件，则调用 ParseOutputPlugin  
    case RULE_OUTPUT:  
        ParseOutputPlugin(rule);  
    }  
....
```

调用 ParseOutputPlugin 函数来进行初始化。

#### 4. 1. 3. 3 规则配置及字符串拆分

ParseOutputPlugin 函数 (parser.c) 是完成输出插件初始化的主要函数。函数的主要流程如下：

```
void ParseOutputPlugin(char *rule)  
{  
    char **toks;  
    char **pp_head;  
    char *plugin_name = NULL;  
    char *pp_args = NULL;  
    int num_arg_toks;  
    int num_head_toks;  
    OutputKeywordNode *plugin;  
  
    /*拆分输出插件的规则配置，toks 指向拆分后的子串，这里的拆分分隔符是串号，  
    字符串被拆分成两部份：插件名和插件配置参数，分别对应 toks[0] 和 toks[1] */  
    toks = mSplit(rule, ":", 2, &num_arg_toks, '\\');
```

```

if(num_arg_toks > 1)
{
    pp_args = toks[1];      /*pp_args 指向规则配置参数*/
}

/*再拆分一下，比如 output database 就被拆成两部份了：插件类型和插件名*/
pp_head = mSplit(toks[0], " ", 2, &num_head_toks, '\\');
plugin_name = pp_head[1];      /*得到插件名*/

if(plugin_name == NULL)
{
    FatalError("%s (%d): Output directive missing output plugin name!\n",
               file_name, file_line);
}

plugin = GetOutputPlugin(plugin_name);      /*得到此关键字对应的 Entry 节点，
即 OutputKeywordNodes 结构*/
}

if( plugin != NULL )
{
    switch(plugin->node_type)           /*判断插件类型，然后调用相应的
plugin->func(pp_args) 调用相应的初始化函数*/
    {
        case NT_OUTPUT_SPECIAL:
            if(pv.alert_cmd_override)
                ErrorMessage("command line overrides rules file alert "
                               "plugin!\n");
            if(pv.log_cmd_override)
                ErrorMessage("command line overrides rules file login "
                               "plugin!\n");
            plugin->func(pp_args);
            break;
        case NT_OUTPUT_ALERT:           //报警
            if(!pv.alert_cmd_override)
            {
                /* call the configuration function for the plugin */
                plugin->func(pp_args);
            }
            else

```

```

{
    ErrorMessage("command line overrides rules file alert "
        "plugin!\n");
}

break;

case NT_OUTPUT_LOG:           //记录
if(!pv.log_cmd_override)
{
    /* call the configuration function for the plugin */
    plugin->func(pp_args);
}
else
{
    ErrorMessage("command line overrides rules file logging "
        "plugin!\n");
}

break;
}
}

```

```

mSplitFree(&toks, num_arg_toks);
mSplitFree(&pp_head, num_head_toks);
}

```

该函数还有两个重要的功能性函数:mSplit 和 GetOutputPlugin。前者用于字符串的拆分，因为在多处都将使用到，我们将在 5.5.4 节中详细介绍，而 GetOutputPlugin 函数用于从输出插件注册链表中获取与当前插件名对应的节点，其算法就是一个遍历链表的过程：

```

OutputKeywordNode *GetOutputPlugin(char *plugin_name)
{
    OutputKeywordList *list_node;

    if(!plugin_name)
        return NULL;

    list_node = OutputKeywords;      /*指向链表首部*/

```

```

        while(list_node)           /*查找与插件名对应的节点*/
    {
        if(strcasecmp(plugin_name, list_node->entry.keyword) == 0)
            return &(list_node->entry);      /*若查找到，则返回相应节点*/
        list_node = list_node->next;
    }
    FatalError("unknown output plugin: '%s",
               plugin_name);

    return NULL;
}

```

#### 4. 1. 3. 4 初始化函数

每一类输出插件的初始化函数的工作流程都类似，这里以 Syslog 插件为例来进行分析：

```

void AlertSyslogInit(u_char *args)
{
    SyslogData *data;           //存放参数解析信息的数据结构
    DEBUG_WRAP(DebugMessage(DEBUG_INIT, "Output: Alert-Syslog Initialized\n"));

    pv.alert_plugin_active = 1;   //激活 pv 变量中的对应标志

    /*解析插件的配置参数，并设定相关数据结构*/
    data = ParseSyslogArgs(args);
    openlog("snort", data->options, data->facility); //打开所需的文件设备

    DEBUG_WRAP(DebugMessage(DEBUG_INIT,"Linking syslog alert function to call
list...\n"));

    /* 将输出插件的功能函数 AlertSyslog 加入到对应的输出函数表中*/
    AddFuncToOutputList(AlertSyslog, NT_OUTPUT_ALERT, data);
    /*将相应的清除函数加入清除链表*/
    AddFuncToCleanExitList(AlertSyslogCleanExit, data);
    /*加入重起函数到重起链表*/
    AddFuncToRestartList(AlertSyslogRestart, data);
}

```

这里需要注意的是，注册化工作完成后，对应的两个指向不同输出函数链表头的全局指针变量：`LogList` 和 `AlertList`。这两个全局指针变量将在后面的时候被引用。

其它插件初始化函数与此类似，读者可以自行分析之。

#### 4. 1. 4 调用

当 Snort 检测到某符合预定义的事件后，将遍历输出插件功能函数链表，即前文建立的 `LogList` 和 `AlertList`，调用输出插件，输出相关信息。关于输出插件的调用，请参考第九章“输出检测结果”。

### 4. 2 预处理插件初始化

数据包的预处理，简单地说，就是在数据包进入检测引擎之前，对其进行前期的一些处理，这是因为：

- 分片的数据包需要重组
- HTTP 请求的 URL 字符串需要统一格式化
- 需要识别扫描探测等行为
- 其它

而单是依靠单纯的规则匹配，是很完成这些工作的。有鉴于此，Snort 引入了分片重组、BO 后门等大量的插件，在数据包进入检测引擎之前对其先期处理。其注册的思想、初始化流程与输出插件是完全相同的，下文将具体分析。

同样地，所有输出插件的实现，也都可以在 `Preprocessors` 文件夹下找到。

#### 4. 2. 1 重要的数据结构

##### PreprocessKeywordlist

预处理插件链表的节点，封装了关键字节点（即每一类预处理插件的相关信息，定义为 `PreprocessNode`），定义如下：

```

typedef struct_preprocessKeywordList
{
    PreprocessNode entry;      //关键字节点，插件名和相应的初始化函数指针
    Struct_PreprocessKeywordList *next;
}PreprocessKeywordlist;

```

同输出插件的注册相同，预处理插件的注册，其实就是针对每一种预处理插件，建立一个 PreprocessKeywordlist 节点，然后将这些节点用 next 链接起来即可。留待初始化的时候，调用 PreprocessNode 中的相应初始化函数，对插件进行初始化。

**PreprocessNode** 封装了预处理插件的名称和对应的初始化函数指针，定义如下：

```

typedef struct_preprocessKeywordNode
{
    char keyword;          /*插件名称*/
    void(*func)(u_char*); /*插件的初始化函数*/
}PreprocessKeywordNode;

```

## 4. 2. 2 注册

主函数调用 InitPreprocessors (plugbase.c) 函数进行预处理插件的初始化，InitPreprocessors 函数源码如下：

```

void InitPreprocessors()
{
    if(!pv.quiet_flag)      /*输出初始化预处理插件的信息*/
    {
        LogMessage("Initializing Preprocessors!\n");
    }
    SetupPortscan();           //端口扫描
    SetupPortscanIgnoreHosts();
    SetupRpcDecode();          //Rpc 解码
}

```

```

    SetupBo();           //Bo 后门
    SetupTelNeg();      //Telnet 会话协商
    SetupStream4();     //Stream4 会话重组
    SetupFrag2();       //Frag2 分片重组和攻击检测
    SetupARPspoofer(); //ARP 欺骗 [实验阶段]
    SetupConv();        //Conversation 对 postscan2 提供支持, 主要
                        负责对 tcp/udp/icmp 协议的状态维护 [实验阶段]
    SetupScan2();       //postscan2 基于状态的端口扫描检测系统
    [实验阶段]
    SetupHttpInspect(); //http 检查 代替了 2.0 版本的 http_decode
    SetupPerfMonitor(); //在控制台或日志文件中打印性能统计数据。
    SetupFlow();
}

```

函数依次调用 `SetupXXX` 函数来实现每一个对应的预处理插件的初始化。注册的基本思想是将关键字名相应的处理函数进行绑定，再构建链表。

以 `SetupPortscan` 为例：

```

void SetupPortscan(void)
{
    RegisterPreprocessor("portscan", PortscanInit);
}

```

可见，每一个 `SetupXXX` 函数都是通过调用了 `RegisterPreprocessor` 函数来实现注册的，`RegisterPreprocessor` 函数的两个参数分别表示插件名和对应的处理函数。

`RegisterPreprocessor` 函数的实现仍然是一个简单的链表操作，源码如下：

```

void RegisterPreprocessor(char *keyword, void (*func) (u_char *))
{
    PreprocessKeywordList *idx;

    DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN,"Registering keyword:preproc =>
% s:% p\n", keyword, func););
    idx = PreprocessKeywords;           //指向链表首节点

    if(idx == NULL)                  //如果链表为空

```

```

{
    /* 为首先节点分配空间 */
    PreprocessKeywords = (PreprocessKeywordList *)
        calloc(sizeof(PreprocessKeywordList), sizeof(char));

    /* 为关键字节点分配空间 */
    PreprocessKeywords->entry.keyword = (char *) calloc(strlen(keyword) + 1,
        sizeof(char));

    /* 将插件名拷贝进结构 */
    strncpy(PreprocessKeywords->entry.keyword, keyword, strlen(keyword)+1);

    /* 封装与插件对应的函数 */
    PreprocessKeywords->entry.func = (void *) func;
}

else
{
    while(idx->next != NULL)           /*循环至链表尾部 */
    {
        /*如果链表已经注册了*/
        if(!strcasecmp(idx->entry.keyword, keyword))
        {
            FatalError("%s(%d) => Duplicate preprocessor keyword!\n",
                file_name, file_line);
        }
        idx = idx->next;
    }

    /*分配空间，相其加入到链表尾部*/
    idx->next = (PreprocessKeywordList *)
        calloc(sizeof(PreprocessKeywordList), sizeof(char));

    idx = idx->next;

    /* alloc space for the keyword */
    idx->entry.keyword = (char *) calloc(strlen(keyword) + 1, sizeof(char));

    /* copy the keyword into the struct */
    strncpy(idx->entry.keyword, keyword, strlen(keyword)+1);
}

```

```
    /* set the function pointer to the keyword handler function */
    idx->entry.func = (void *) func;
}
}
```

## 4. 2. 3 初始化

因为 Snort 允许用户在 snort.conf 中对预处理插件进行配置，所以，系统必须从配置文件中把用户配置的信息读取出来，分析参数，以待进一步对数据包进行预处理时使用。这就是预处理插件的初始化。算法类似于输出插件的初始化。

初始化流程算法流程如下：

1、主函数调用 ParseRuleFile ，从 snort.conf 配置文件中一行一行地读取配置参数，交由 ParseRule 函数进一步拆分；

2、ParseRule 函数分析每一行的配置，判断配置的类型，如果是预处理插件的配置（case RULE\_PREPROCESS:），则调用 ParsePreprocessor 进一步分析预处理插件的配置。

3、ParsePreprocessor (parser.c) 函数将取出配置文件中使用的预处理插件的插件名关键字：

```
toks = mSplit(rule, ":", 2, &num_arg_toks, '\\');
```

然后遍历我们注册时建立的链表。找到与此配置的节点后，将配置参数交由对应的预处理插件的初始化函数来完成。

例如：我们在 snort.conf 中配置了预处理插件 stream4:

```
preprocessor stream4: disable_evasion_alerts
```

ParseRule 函数提取出关键字'preprocessor'，发现是预处理插件的配置，则将这个配置字符串 stream4: disable\_evasion\_alerts 交由 ParsePreprocessor 函数，ParsePreprocessor 进一步拆分分析后，提取出预处理插件的名称 stream4。然后遍历预处理插件初始化时建立的链表（PreprocessKeywordList \*PreprocessKeywords），找到与 stream4 相配置的节点，因为在注册插件之时，

节点中已经封装了相应的初始化函数，那么直接调用

```
pl_idx->entry.func(pp_args);
```

将配置参数 disable\_evasion\_alerts 传递给对应的初始化函数，就可以进行 stream4 插件的初始化了。

关于 ParseRule 等函数的详细分析，请参考检测规则初始化相关章节，这里给出 ParsePreprocessor 函数的源码分析：

```
/*参数 char *rule: 等待分析的预处理插件的配置字符串*/
void ParsePreprocessor(char *rule)
{
    char **toks;           /* pointer to the tokenized array parsed from
                           * the rules list */
    char **pp_head;        /* parsed keyword list, with preprocessor
                           * keyword being the 2nd element */
    char *funcname;        /* the ptr to the actual preprocessor keyword */
    char *pp_args = NULL;   /* parsed list of arguments to the
                           * preprocessor */

    int num_arg_toks;      /* number of argument tokens returned by the mSplit function */
    int num_head_toks;     /* number of head tokens returned by the mSplit function */
    int found = 0;          /* flag var */
    PreprocessKeywordList *pl_idx; /* index into the preprocessor
                           * keyword/func list */

    /*把配置字符串以冒号为分隔符一段一段地拆分出来，置于二维数组中 */
    toks = mSplit(rule, ":", 2, &num_arg_toks, '\\');

    if(num_arg_toks > 1)
    {
        /*
         DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"toks[1] = %s\n",
        toks[1]));
        */
        /* the args are everything after the ":" */
        pp_args = toks[1]; /*提取出预处理插件的参数*/
    }
}
```

```

/* toks[0]中包含了配置字符串的前半部分，即配置参数类型和插件名，这里再将它
们分开 */
pp_head = mSplit(toks[0], " ", 2, &num_head_toks, '\\');

/* 获得插件的名称*/
funcname = pp_head[1];

/* set the index to the head of the keyword list */
pl_idx = PreprocessKeywords;

/* 遍历我们在注册的时候创建的链表 */
while(pl_idx != NULL)
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
        "comparing: \"%s\" => \"%s\"\n",
        funcname, pl_idx->entry.keyword));

    /* compare the keyword against the current list element's keyword */
    if(!strcasecmp(funcname, pl_idx->entry.keyword))
    {
        /*如果找到了，就调用相应的初始化函数*/
        pl_idx->entry.func(pp_args);
        found = 1;
    }
    if(!found)
    {
        pl_idx = pl_idx->next;
    }
    else
        break;
}

mSplitFree(&toks, num_arg_toks);
mSplitFree(&pp_head, num_head_toks);

if(!found)
{
    FatalError(" unknown preprocessor \"%s\"\n",

```

```
        funcname);  
    }  
}
```

这样，所有在配置文件中打开的插件将被完成初始化。各种预处理插件的初始化都很相似，都遵循以下步骤：

- 1、拆解分析配置参数，填充相应的数据结构；
- 2、将该预处理器插件的实际功能函数，填充至实际执行的预处理插件链表；
- 3、将该预处理器插件的退出清除函数，填充至实际执行的预处理插件链表；
- 4、将该预处理器插件的重置函数，填充至实际执行的预处理插件链表；

关于插件的初始化函数，以下将以 Flow 插件为例进行分析，其它插件，读者可以对应分析。

```
/*参数 args：等分析的配置选项字符串*/  
static void FlowInit(u_char *args)  
{  
    static int init_once = 0; /*插件只初始化一次，故需要一个静态标志*/  
    int ret;  
    static SPPFLOW_CONFIG *config = &s_config; /*结构用来存储命令行  
参数*/  
  
    if(init_once)           /*已经初始化过了.....*/  
        FatalError("%s(%d)  Unable  to  reinitialize  flow!\n",  file_name,  
file_line);  
    else  
        init_once = 1;  
  
    /* 先填充一些默认参数先 */  
    config->stats_interval = DEFAULT_STAT_INTERVAL;  
    config->memcap = DEFAULT_MEMCAP;
```

```

config->rows      = DEFAULT_ROWS;
config->hashid = HASH2; /* use the quickest hash by default */

FlowParseArgs(config, args);      /*在这里进行拆分了*/

/*初始化缓存，进行一些分配内存的工作*/
if((ret = flowcache_init(&s_fcache, config->rows, config->memcap,
                         giFlowbitSize,      config->hashid)) !=

FLOW_SUCCESS)
{
    FatalError("Unable to initialize the flow cache!"
               "-- try more memory (current memcap is %d)\n",
               config->memcap);
}

/*显示配置参数*/
DisplayFlowConfig();

s_flow_running = 1;

/*将执行、退出清除、重置三个函数分别添加进三个对应的链表，实际
执行函数 FlowPreprocessor 的工作原理，请参考第九章*/
AddFuncToPreprocList(FlowPreprocessor);
AddFuncToCleanExitList(FlowCleanExit, NULL);
AddFuncToRestartList(FlowRestart, NULL);
}

```

流跟踪模块主要用于建立和维护一个连接状态表（协议、来源地址、目的地址、来源端口、目的端口），主要涉及到的参数有：表的大小、表的行数、把信

息输出到终端的间隔时间和查表的算法。存储这些参数对应的结构 SPPFLOW\_CONFIG 定义如下 (spp\_flow.c):

```
typedef struct _SPPFLOW_CONFIG
{
    int stats_interval;
    int memcap;
    int rows;
    FLOWHASHID hashid;
} SPPFLOW_CONFIG;
```

注：关于 *Flow* 插件的更多内容，请参考 *Snort* 安装目录/doc/README.flow

这样，所有的链表被建立好后，待由截获到数据包后对数据包进行逐一地调用，对数据包进行预处理。

#### 4. 2. 4 调用

在数据包入检测引擎之前，将逐一调用所有初始化过的插件，对数据包进行预处理。

在函数 Preprocess 中：

```
int Preprocess(Packet * p)
{
    PreprocessFuncNode *idx;
    .....
    idx = PreprocessList; /*指向预处理插件的顶层链表*/
    p->processors = PP_ALL; /*打开所有预处理器*/
    while(idx != NULL)      /*遍历预处理器插件，对数据包进行预处理*/
    {
        assert(idx->func != NULL);
        idx->func(p);
    }
}
```

```
    idx = idx->next;
}
.....
}
```

注：关于每一类预处理插件的实现，请参考第九章

## 4. 3 规则选项关键字插件初始化

InitPlugins 为处理规则选项关键字插件模块的初始化调用函数。所谓规则选项关键字插件，也叫做检测插件。这是因为我们在定义 Snort 的规则的时候，许多的关键字，不能单纯地依靠字符串的匹配来解决，比如我们定义了如下规则：

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 较大的 ICMP 包"; dsiz...>800;.....
```

规则中的负载长度关键字 `dsiz>800`，表示如果捕获包的负载长度超过了 800 字节，则我们认为它是一个非法的包。显然，这种规则的匹配需要有专门的函数来处理 `dsiz` 大于、等于、小于等各种情况。这样，把 `dsiz` 关键字同专门的处理函数对应起来，就是规则选项关键字插件了。这些插件都定义在源码目录的 `Detection Plugins` 目录下。我们可以参考 Snort 的用户手册，以了解这些源文件与规则选项关键字的对应关系。

与上两类插件初始化相同，首先在主函数中调用 `InitPlugIns` 函数注册插件。在 `InitPlugIns` 函数中，分别调用每一类检测插件的 `SetupXXX` 函数进行注册。而 `SetupXXX` 函数又通过调用了 `RegisterPlugin`（插件名、插件初始化函数）来建立检测插件的初始化链表。以实现关键字名与检测插件初始化函数的封装。

检测插件的初始化是在检测规则初始化构建 OTN 节点的函数 `ParseRuleOptions` 中实现的（请参考 5.7 节相关内容），相关源码如下：

```
kw_idx = KeywordList;      /*指向插件链表首部*/
found = 0;

while(kw_idx != NULL)      /*遍历链表*/
{

```

```

/*进行关键字匹配，调用在注册时封装的初始化函数*/
if(!strcasecmp(option_name, kw_idx->entry.keyword))
{
    if(num_opts == 2)
    {
        kw_idx->entry.func(option_args, otn_tmp, protocol);
    }
    else
    {
        kw_idx->entry.func(NULL, otn_tmp, protocol);
    }
    found = 1;
    break;
}
kw_idx = kw_idx->next;
}

```

这里，指向函数的指针实际指向的是对应检测插件的初始化函数。在这些初始化函数中，又会调用 AddOptFuncToList 函数（plugbase.c）来实现实际功能函数的封装。函数源码如下：

```

OptFpList *AddOptFuncToList(int (*func) (Packet *, struct _OptTreeNode *,
                                         struct _OptFpList *), OptTreeNode * otn)
{
    OptFpList *idx;      /* index pointer */

    idx = otn->opt_func;

    if(idx == NULL)
    {
        otn->opt_func = (OptFpList *) calloc(sizeof(OptFpList), sizeof(char));

        if(otn->opt_func == NULL)
        {
            FatalError("new node calloc failed: %s\n",
                       strerror(errno));
        }

        /* set the head function */
    }
}

```

```

otn->opt_func->OptTestFunc = func;

    idx = otn->opt_func;
}

else
{
    /* walk to the end of the list */
    while(idx->next != NULL)
    {
        idx = idx->next;
    }

    /* allocate a new node on the end of the list */
    idx->next = (OptFpList *) calloc(sizeof(OptFpList), sizeof(char));

    if(idx->next == NULL)
    {
        FatalError("AddOptFuncToList new node calloc failed: %s\n",
                  strerror(errno));
    }

    /* move up to the new node */
    idx = idx->next;

    /* link the function to the new node */
    idx->OptTestFunc = func;
}

return idx;
}

```

初始化函数将实际功能执行函数的指针 `func` 传递过来，以实现将检测插件函数链表封装进 OTN。

检测插件函数的调用，是在进入检测引擎后，进行规则匹配的时候，来进行调用的，请参考 10.4 节相关内容。

关于每一类插件的具体实现，因为 Snort 的检测插件众多，在 2.0 版本中，有 22 个，到了 2.2 版本，已有 28 个。我们将在第 11 章中，选择其中一

些具有代表性的插件，进行详细的分析。

# 第5章 检测规则初始化引擎

Snort 是一个以特征检测为基础的入侵检测系统，主要是通过截获网络中的每一个数据包，然后将数据包同预定义的入侵特征的规则相匹配，以发现可疑的入侵行为。Snort 规则定义在 rules 目录下一系列 \*.rules 文件中，Snort 的检测规则初始化引擎的工作，就是把这些文本文件的内容，合理地组织分类，读取到内存当中，以供检测引擎匹配使用。所以，如何高效地组织规则的，是提高 Snort 运行效率、降低误报漏报的关键。

要更好地理解 Snort 如何组织规则，首先应该了解 Snort 的规则是怎么样的、如何配置规则，所以，在阅读本章之前，读者应该在能够使用 Snort 的基础上，了解 Snort 的规则结构，能够自定义规则。关于这些内容，读者可以参考 doc 目录下的《snort\_manual.pdf》第二章和第三章的相关内容或者其它 Snort 的相关参考资料。

Snort 规则的组织采用的链表这一种数据结构，在以前的版本中，采用的是二维链表，从 2.0 版本开始，引入了三维链表来组织规则，Snort 的检测效率由此大大提高。

三维链表？立体？岂不是很玄？那就继续往下看吧……

## 5. 1 重要的数据结构

既然我们说主要算法是链表，那么其实检测规则的初始化，就是如何定义一种结构，然后将规则从文件中一条一条地读取出来赋给结构的相应成员变量，然后把他们组织成链表。检测引擎进行配则匹配的时候，就是一个遍历链表，查找匹配节点的过程。当然，最笨的办法是从头到尾串成一串做成一条一维的链表，这样实现起来虽然简单，但是却是不可行的，因为 Snort 目前有两个多条规则库，以后还会继续扩展，谁也不希望每一个数据包被匹配几千次才有结果吧！所以，合理地进行规则的分类，这是本章以及下一章的所有设计的思想基础

Snort 的规则主要分成两块：规则头和规则选项。如下面这条关于木马后门

的规则：

```
alert tcp $EXTERNAL_NET any <> $HOME_NET 0 (msg:"BAD-TRAFFIC tcp port  
0 traffic"; flow:stateless; classtype:misc-activity; sid:524; rev:8;)
```

括号之前部份就是规则头，它主要包含了 IP 地址、端口、协议、数据流的方向以及对此规则采取的动作。

Snort 的动作共分为五类：alert、log、pass、activate 和 dynamic。Snort 规则的分类其实就是按动作来划分的，所以，首层链表结构定义如下：

```
typedef struct _RuleListNode  
{  
    ListHead *RuleList;           /* 二层链表数据结构 */  
    int mode;                   /* 规则模式 */  
    int rval;                   /* 0 表示不检测, 1 表示检测事件 */  
    int evalIndex;              /* 规则类型的索引值 */  
    char *name;                 /* 规则链表节点的名称 */  
    struct _RuleListNode *next;  
} RuleListNode;
```

于是，首层链表自然就出来了：定义五个 RuleListNode 结构的节点：alert、log、pass、activate 和 dynamic。然后把它们五个连起来做成一维链表就 OK 了（至于如何连起来的，请参考 5.3 节）。

Snort 按照规则的动作来分类，每种动作的规则再按照协议来划分，可能是 TCP/UDP/ICMP/IP 协议中的一种，而且，如果数据包与规则匹配，需要输出的话，还需要调用对应的输出插件，于是，第二层数据结构定义如下：

```
typedef struct _ListHead  
{  
    RuleTreeNode *IpList;          // 协议的分类  
    RuleTreeNode *TcpList;  
    RuleTreeNode *UdpList;  
    RuleTreeNode *IcmpList;  
    struct _OutputFuncNode *LogList; // 输出插件（日志）  
    struct _OutputFuncNode *AlertList; // 输出插件（报警）  
    struct _RuleListNode *ruleListNode; // 三层链表节点结构  
} ListHead;
```

规则头除了包含动作、协议外，还有在此协议下的地址、端口、数据流方向

等等，于是加上一些控制标志，第三层链表节点结构定义如下（RTN）：

```
typedef struct _RuleTreeNode
{
    RuleFpList *rule_func; /* match functions.. (Bidirectional etc.. ) */

    int head_node_number;

    int type;           /*规则类型*/

    IpAddrSet *sip;      /*来源地址*/
    IpAddrSet *dip;      /*目的地址*/

    int not_sp_flag;     /* 源端口取反 */
    /*来源端口可能是一个范围，lsp 表示开始的端口，hsp 表示高端口*/
    u_short hsp;         /* hi src port */
    u_short lsp;         /* lo src port */

    int not_dp_flag;     /*目标端口取反*/
    u_short hdp;         /* hi dest port */
    u_short ldp;         /* lo dest port */

    u_int32_t flags;     /* 控制标志，用来表示地址取反、任意地址、数据流是单向
                           还是双向、端口存在与否等等相关的信息 */

    /* stuff for dynamic rules activation/deactivation */
    int active_flag;
    int activation_counter;
    int countdown;
    ActivateList *activate_list;

    struct _RuleTreeNode *right; /* ptr to the next RTN in the list */

    OptTreeNode *down; /* list of rule options to associate with this
                       rule node */

    struct _ListHead *listhead;

} RuleTreeNode;
```

RTN 成员并没有 Next , 它用了 Right 来代替, 用来组织这一层所有的 RTN 节点。另一个成员 down, 指向第四层的数据节点: 规则节点选项。

规则选项组成了 snort 入侵检测引擎的核心, 既强大又灵活。共包含了 40 多个关键字(请参考《snort\_manual》), 这也决定了规则选项的复杂程度是相当高的, 所以, 这也要求我们, 要理解 Snort 的规则选项的结构定义, 首先应当了解如何使用 Snort 规则配置使用的相关语法。

这里给出 Snort 的规则选项的结构定义 RTN, 并给出其中关键的成员的注释。

```
typedef struct _OptTreeNode
{
    /* plugin/detection functions go here */
    OptFpList *opt_func;
    RspFpList *rsp_func; /* response functions */
    OutputFuncNode *outputFuncs; /* per sid enabled output functions */

    /* the ds_list is absolutely essential for the plugin system to work,
       it allows the plugin authors to associate "dynamic" data structures
       with the rule system, letting them link anything they can come up
       with to the rules list */
    void *ds_list[64]; /* list of plugin data struct pointers */

    int chain_node_number;

    int type;           /* what do we do when we match this rule */
    int evalIndex;      /* where this rule sits in the evaluation sets */

    int proto;          /* protocol, added for integrity checks
                           during rule parsing */
    struct _RuleTreeNode *proto_node; /* ptr to head part... */
    int session_flag;   /* record session data */

    char *logto;        /* log file in which to write packets which
                           match this rule*/
    /* metadata about signature */
    SigInfo sigInfo;
```

```

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;

/* stuff for dynamic rules activation/deactivation */
int active_flag;
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;

<skynet>
```

## 5. 2 规则初始化流程

在分析了以上的四个结构的定义之后，我们来简单谈谈系统规则初始化的总体流程，让大家先有一个模糊的总体性的了解，以为后续分析做个铺垫。

主函数首先调用 `CreateDefaultRules` 函数，来完成第一层链表的搭建。接着，主函数中调用 `ParseRuleFile`，来读取 `snort.conf` 等配置文件，将配置文件一行一行地送往 `ParseRule` 解析，读到：

```
include $RULE_PATH/local.rules
```

```
include $RULE_PATH/bad-traffic.rules  
include $RULE_PATH/exploit.rules  
.....
```

ParseRule 把字符串解析后，把这些规则的文件名提取出来，再交回来给 ParseRuleFile，这样，ParseRuleFile 就读取了规则文件中一条条的规则，交由 ParseRule 进行解析，这样，这两个函数反复回调，直至完成规则的初始化过程。

当然，其它的诸如变量配置（var）、预处理插件配置、输出插件配置等等动作也同时进行的（前文在讲插件的时候已有所涉及），它们读取、分析流程与之相似，读者可以根据规则的初始化流程对照分析。

## 5. 3 CreateDefaultRules

CreateDefaultRules 搭建最上层链表主要是通过五次调用 CreateRuleType 函数来实现的：

```
void CreateDefaultRules()  
{  
    CreateRuleType("activation", RULE_ACTIVATE, 1, &Activation);  
    CreateRuleType("dynamic", RULE_DYNAMIC, 1, &Dynamic);  
    CreateRuleType("alert", RULE_ALERT, 1, &Alert);  
    CreateRuleType("pass", RULE_PASS, 0, &Pass);  
    CreateRuleType("log", RULE_LOG, 1, &Log);  
}
```

五次调用完成，以动作来划分的规则首层链表就搭建完成了。

CreateRuleType 函数负责链表的具体实现，源码分析如下：

```
ListHead *CreateRuleType(char *name, int mode, int rval, ListHead *head)  
{  
    RuleListNode *node;  
    int evalIndex = 0;  
  
    if(!RuleLists)           /*如果链表头为空，则新建分配空间*/  
    {  
        RuleLists = (RuleListNode *)calloc(1, sizeof(RuleListNode));  
        node = RuleLists;
```

```

    }

else /* 否则， 循环至链表末尾*/
{
    node = RuleLists;
    while(1)
    {
        evalIndex++; /*索引累加*/
        if(!strcmp(node->name, name))
            return NULL;
        if(!node->next)
            break;
        node = node->next;
    }
    /*为末尾节点分配空间*/
    node->next = (RuleListNode *) calloc(1, sizeof(RuleListNode));
    node = node->next;
}

/*如果该规则头对应的第二层链表 ListHead 为空，则分配空间并初始化*/
if(!head)
{
    node->RuleList = (ListHead *)calloc(1, sizeof(ListHead));
    node->RuleList->IpList = NULL;
    node->RuleList->TcpList = NULL;
    node->RuleList->UdpList = NULL;
    node->RuleList->IcmpList = NULL;
    node->RuleList->LogList = NULL;
    node->RuleList->AlertList = NULL;
}
else
{
    node->RuleList = head;
}

//给规则头节点赋值
node->RuleList->ruleListNode = node; //回指针
node->mode = mode;
node->rval = rval;

```

```

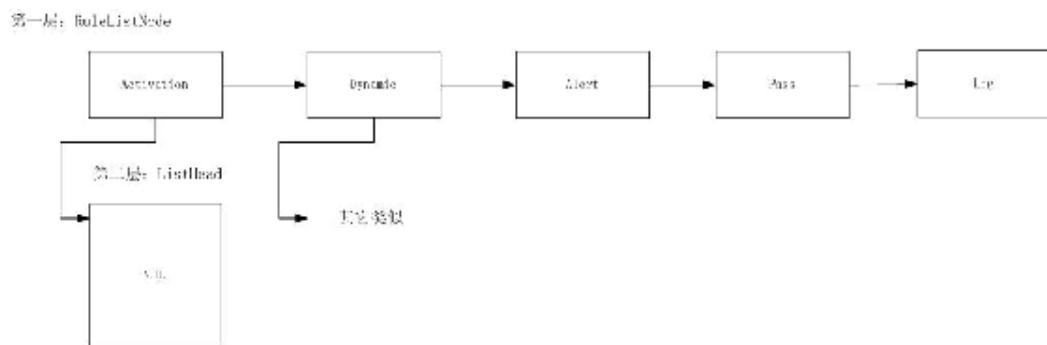
node->name = strdup(name);
node->evalIndex = evalIndex;
node->next = NULL;

pv.num_rule_types++; /*规则类型计数器累加*/
return node->RuleList;
}

}

```

这样，五次调用 CreateRuleType 完成后，建成的规则的第一层链表（对应的全局变量 RuleListNode \*RuleLists）大致如下图所示：



## 5. 4 ParseRuleFile

我们在前一章介绍各种插件的时候，就反复地提到了 ParseRuleFile，函数的主要作用是就打开参数 `char *file` 指向的文件，然后一行一行地读取文件，再交由后续函数得处理。在本节，我们将对其进行详细地分析。

主函数调用 ParseRuleFile 函数完成关键的检测规则解析任务，实质上该函数不仅是解析检测规则的集合，还包括所有的系统配置规则的解析，包括预处理器、输出插件、配置命令等。

ParseRuleFile 函数在读取了一行行地读取了相应的文件后，将其送往实际的规则解析模块 ParseRule()进行最后解析。第一次，解析的文件是 `sornt.conf` 配置文件，再送往下层函数 ParseRule，再由 ParseRule 进行继续解析，判断执行指令（每行的行首），若果是 `include xxxfile`，再回调 ParseRuleFile(xxxfile) 如此循环，直至将配置文件、插件文件、规则文件等全部解析完毕。

函数 ParseRuleFile 代码分析如下（有删节）：

```
void ParseRulesFile(char *file, int inclevel)
{
    FILE *thefp;          /* file pointer for the rules file */
    char buf[STD_BUF];    /* file read buffer */
    char *index;          /* buffer indexing pointer */
    char *stored_file_name = file_name;
    int stored_file_line = file_line;
    char *saved_line = NULL;
    int continuation = 0;
    char *new_line = NULL;
    struct stat file_stat; /* for include path testing */

    if(inclevel == 0)           /*是在进行规则解析吗？ */
    {
        if(!pv.quiet_flag)
        {

LogMessage("\n+++++++++++++++++++++++++++++++++++++\n");
            LogMessage("Initializing rule chains...\n");
        }
    }

    stored_file_line = file_line;
    stored_file_name = file_name;
    file_line = 0;

    file_name = strdup(file);      /*提取文件名*/
    if(file_name == NULL)
    {
        FatalError("ParseRulesFile strdup failed: %s\n",
                   strerror(errno));
    }

/*stat 函数返回一个与此文件有关的信息结构 buf*/
    if(stat(file_name, &file_stat) < 0)
    {
```

```

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"ParseRulesFile: stat
""on %s failed - going to config_dir\n", file_name););

free(file_name);

/*分配空间*/
file_name = calloc(strlen(file) + strlen(pv.config_dir) + 1,
                   sizeof(char));

if(file_name == NULL)      /*分配空间失败*/
{
    FatalError("ParseRulesFile calloc failed: %s\n",
               strerror(errno));
}

/*将文件名拷贝过*/
strncpy(file_name, pv.config_dir, strlen(file) +
        strlen(pv.config_dir) + 1);

strlcat(file_name, file, strlen(file) + strlen(pv.config_dir) + 1);

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"ParseRulesFile:
Opening ""and parsing %s\n", file_name););
}

/* 打开规则文件 */
if((thefp = fopen(file_name, "r")) == NULL)
{
    FatalError("Unable to open rules file: %s or %s\n", file,
               file_name);
}

/* 清空缓存，buf 用于存储一行规则的信息 */
bzero((char *) buf, STD_BUF);

/* 循环读取，每一次读取 STD_BUF(1024)个字节 */
while((fgets(buf, STD_BUF, thefp)) != NULL)
{
    file_line++;           /*行计数器累加*/
}

```

```

index = buf;           /*指针指向缓存内容，再进行相应判断*/

#ifndef DEBUG2
    LogMessage("Got line %s (%d): %s", file_name, file_line, buf);
#endif

/*判断行首是否有特殊字符，有则跳过*/
while(*index == ' ' || *index == '\t')
    index++;

/* 判断当前行是注释'#'或者是回车，若如不是，则进行进一步分解 */
if((*index != '#') && (*index != 0x0a) && (*index != ';') &&
   (index != NULL))
{
    if(continuation == 1)    /*如果是续行，是的话，把旧行连过来*/
    {
        new_line = (char *) calloc(strlen(saved_line) + strlen(index)
                                   + 1, sizeof(char));
        strncat(new_line, saved_line, strlen(saved_line));
        strncat(new_line, index, strlen(index));
        free(saved_line);
        saved_line = NULL;
        index = new_line;

        if(strlen(index) > PARSERULE_SIZE)
        {
            FatalError("Please don't try to overflow the parser, "
                       "that's not very nice of you... (%d-byte "
                       "limit on rule size)\n", PARSERULE_SIZE);
        }
    }

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "concat
rule: %s\n",
                           new_line););
}
}

/* 判断当前行还有没有续行，如果没有，就调用调用 ParseRule 进** 行进一步分解，
否则的话，就设置标志变量 continuation ,继续提取*/

```

```

if(ContinuationCheck(index) == 0)
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
                           "[*] Processing rule: %s\n", index));

    /* 调用 ParseRule 函数，解析所有的系统配置规则，包括插件配置，
     * 检测规则配置，变量定义，类定义等等
     */
    ParseRule(thefp, index, inclevel);

    if(new_line != NULL)      /*重置所有变量，以待下一行*/
    {
        free(new_line);
        new_line = NULL;
        continuation = 0;
    }
    else                      /*如果是续行*/
    {
        /* 则保存当前行 */
        saved_line = strdup(index);

        /* 设置续行标志 */
        continuation = 1;
    }
}

bzero((char *) buf, STD_BUF); /*分析完一行，清空缓存*/
}

if(file_name)
    free(file_name);

file_name = stored_file_name;
file_line = stored_file_line;

if(inclevel == 0 && !pv.quiet_flag)
{

```

```

LogMessage("%d Snort rules read...\n", rule_count);
LogMessage("%d Option Chains linked into %d Chain Headers\n", opt_count,
           head_count);
LogMessage("%d Dynamic rules\n", dynamic_rules_present);
LogMessage("+++++++++++++++++++++\n"
\n");
}

fclose(thefp);

/* plug all the dynamic rules together */
if(dynamic_rules_present)
{
    LinkDynamicRules();
}

if(inclevel == 0)
{
#ifdef DEBUG
    DumpRuleChains();
#endif

    IntegrityCheckRules();
/*FindMaxSegSize();*/
}

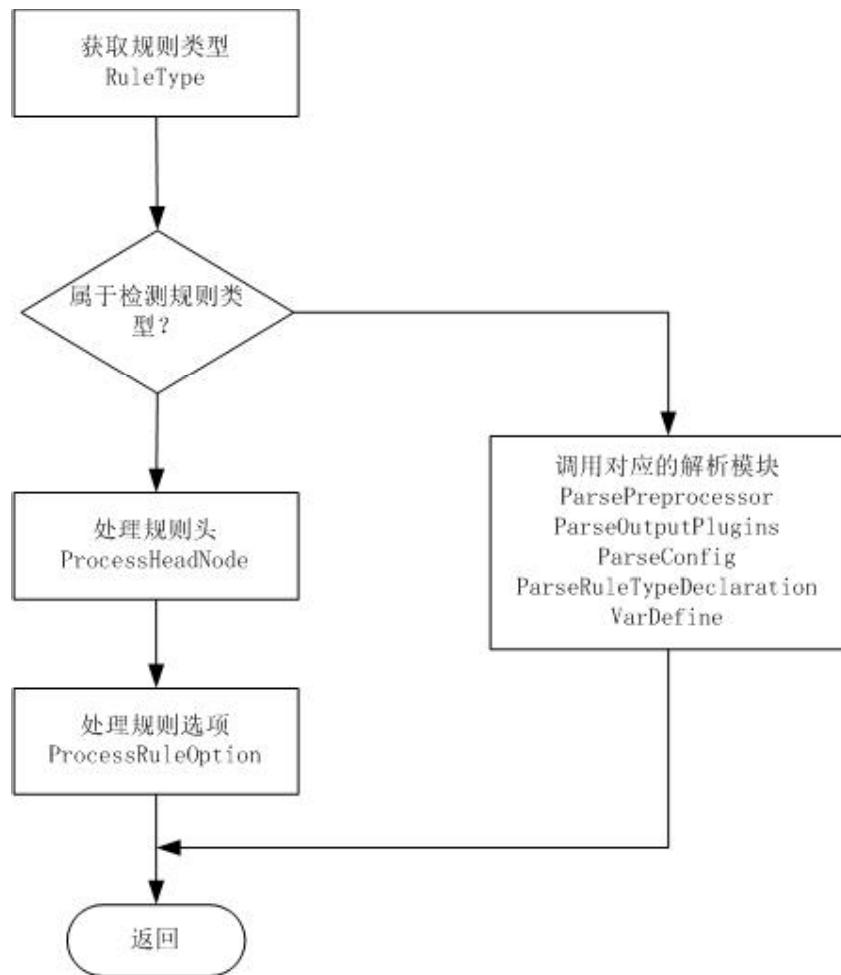
return;
}

```

## 5. 5 ParseRule

### 5. 5. 1 函数流程

ParseRule 函数负责将上层函数 ParseRuleFile 传递过来的一行字符串进行再进一步分解，提取出指令类型，再进行进一步判断，调用相应的处理函数，算法流程如下图所示：



函数源代码分析如下：

```

void ParseRule(FILE *rule_file, char *prule, int inclevel)
{
    char **toks; /* 按某个分隔符拆一个字符串，结果返回到一个二维字符指针中 */
    int num_toks; /* 拆分后的子串数 */
    int rule_type; /* 规则类型列举变量 */
    char rule[PARSERULE_SIZE];
    int protocol = 0;
    char *tmp;
    RuleTreeNode proto_node;
  
```

```

RuleListNode *node = RuleLists;           // 规则链表头

/* 从字符串中去掉回车换行符 */
strip(prule);

/* 清空 rule[] 数组 */
bzero((void *)rule, sizeof(rule));

strncpy(rule, ExpandVars(prule), PARSERULE_SIZE-1);      /* 把待分拆的字符串拷到
rule 当中 */

/* 按空格符拆分 rule 串，结果返回至 toks 中 */
toks = mSplit(rule, " ", 10, &num_toks, 0);

/* 清空 RTN */
bzero((char *) &proto_node, sizeof(RuleTreeNode));

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"[*] Rule start\n"));

/* toks[0] 中包含了拆分出来的规则类型，如 var、output、rule……，用于进行指令类型
判断 */
rule_type = RuleType(toks[0]);

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Rule type: "));

/* 判断指令类型，如果是五种规则类型之一，则跳过，如果是文件包含、变量定义、预
处理插件、输出插件等等，调用相应的处理函数 */

switch(rule_type)
{
    case RULE_PASS:

```

```

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Pass\n"));

    break;

case RULE_LOG:

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Log\n"));

    break;

case RULE_ALERT:

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Alert\n"));

    break;

case RULE_INCLUDE:      /*如果是文件包含指令，则读出相应的包含的文件
名写入 tmp，再回调 ParseRulesFile 函数，进一步拆分这个被包含的文件*/
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Include\n"));

    if(*toks[1] == '$')

    {

        if((tmp = VarGet(toks[1]+1)) == NULL)

        {

            FatalError("%s(%d) => Undefined variable %s\n",
                      file_name, file_line, toks[1]);

        }

    }

    else

    {

        tmp = toks[1];

    }

ParseRulesFile(tmp, inclevel + 1);

mSplitFree(&toks, num_toks);

return;

```

```
case RULE_VAR:           /*如果是变量声明*/
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Variable\n"));
    VarDefine(toks[1], toks[2]);           /*存储该变量*/
    mSplitFree(&toks, num_toks);          /*清空*/
    return;

case RULE_PREPROCESS:    /*预处理插件*/
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Preprocessor\n"));
    ParsePreprocessor(rule);
    mSplitFree(&toks, num_toks);
    return;

case RULE_OUTPUT:         /*如果是配置输出插件*/
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Output Plugin\n"));
    ParseOutputPlugin(rule);
    mSplitFree(&toks, num_toks);
    return;

case RULE_ACTIVATE:
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Activation
rule\n"));
    break;

case RULE_DYNAMIC:
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Dynamic rule\n"));
    break;

case RULE_CONFIG:        /*如果是 Config 配置指令*/
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Rule      file
"));
```

```

config\n");}

ParseConfig(rule);

mSplitFree(&toks, num_toks);

return;

case RULE_DECLARE:

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Rule           type

declaration\n"));

ParseRuleTypeDeclaration(rule_file, rule);

mSplitFree(&toks, num_toks);

return;

case RULE_THRESHOLD:

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Threshold\n"));

ParseSFThreshold(rule_file, rule);

mSplitFree(&toks, num_toks);

return;

case RULE_SUPPRESS:

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Suppress\n"));

ParseSFSuppress(rule_file, rule);

mSplitFree(&toks, num_toks);

return;

case RULE_UNKNOWN:

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Unknown rule type,
might be declared\n"));

/* find out if this ruletype has been declared */

while(node != NULL)

```

```

    {
        if(!strcasecmp(node->name, toks[0]))
            break;
        node = node->next;
    }

    if(node == NULL)
    {
        FatalError("%s(%d) => Unknown rule type: %s\n",
                  file_name, file_line, toks[0]);
    }

    break;

default:
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "Invalid input: %s\n",
                           prule););
    mSplitFree(&toks, num_toks);
    return;
}

/*规则最少也要包含规则头，规则头最少也得有 7 个子串，包含动作、地址、端口等*/
if(num_toks < 7)
{
    FatalError("%s(%d): Bad rule in rules file\n", file_name, file_line);
}

if(!CheckRule(prule))      /*对规则进行简单的完整性校验，如最后一位是否是 ')'，倒数第二位是否为 ';' 等*/
{

```

```

        FatalError("Unterminated rule in file %s, line %d\n"
                    "      (Snort rules must be contained on a single line or\n"
                    "      on multiple lines with a '\\\' continuation character\n"
                    "      at the end of the line,  make sure there are no\n"
                    "      carriage returns before the end of this line)\n",
                    file_name, file_line);

        return;
    }

    if (rule_type == RULE_UNKNOWN)
        proto_node.type = node->mode;
    else
        proto_node.type = rule_type;      /*填充 RTN 的规则类型*/

    protocol = WhichProto(toks[1]);      /*取得规则的协议值*/

    ProcessIP(toks[2], &proto_node, SRC);      /*填充 RTN 的来源地址 IP 地址和掩码进
RTN*/
/* 检测是不是规则中没有包含端口信息，直接就是数据流方向标志了 */
if(!strcasecmp(toks[3], "->") || !strcasecmp(toks[3], "<>"))
{
    FatalError("%s:%d => Port value missing in rule!\n",
              file_name, file_line);
}

/* 填充 RTN 的来源端口*/
if(ParsePort(toks[3], (u_short *) & proto_node.hsp,
             (u_short *) & proto_node.lsp, toks[1],
             (int *) &proto_node.not_sp_flag))

```

```

{
    proto_node.flags |= ANY_SRC_PORT;           /*函数返回真表示为任意端口,
置标志位*/
}

if(proto_node.not_sp_flag)                  /*源端口取反(!), 置标志位*/
    proto_node.flags |= EXCEPT_SRC_PORT;
if(!strncmp("<>", toks[4], 2))          /*数据流如果是双向的话, 置标志位*/
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Bidirectional rule!\n"));
    proto_node.flags |= BIDIRECTIONAL;
}

/*进行数据流方向的合法性较验*/
if(strcmp("->", toks[4]) && strcmp("<>", toks[4])) {
    FatalError("%s(%d): Illegal direction specifier: %s\n", file_name,
               file_line, toks[4]);
}

ProcessIP(toks[5], &proto_node, DST);        /*填充 RTN 的目的地址*/

if(ParsePort(toks[6], (u_short *) &proto_node.hdp,      /*填充 RTN 目的端口*/
             (u_short *) &proto_node.ldp, toks[1],
             (int *) &proto_node.not_dp_flag))
{
    proto_node.flags |= ANY_DST_PORT; /*如果是任意端口, 置标志位*/
}

/*如果规则中包含了规则选项, 但是选项却又不是以括号开头的话..... */
if (num_toks > 7 && toks[7][0] != '(')

```

```

{
    FatalError("%s(%d): The rule option section (starting with a '(') must "
              "follow immediately after the destination port.  "
              "This means port lists are not supported.\n",
              file_name, file_line);
}

if(proto_node.not_dp_flag)          /*如果目的地址取反，设置标志位*/
    proto_node.flags |= EXCEPT_DST_PORT;

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"proto_node.flags = 0x%X\n",
proto_node.flags););
DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Processing Head
Node...\n"));

switch(rule_type) /*调用 ProcessHeadNode 处理 RTN，搭建第三层链表*/
{
    case RULE_ALERT:
        ProcessHeadNode(&proto_node, &Alert, protocol);
        break;

    case RULE_LOG:
        ProcessHeadNode(&proto_node, &Log, protocol);
        break;

    case RULE_PASS:
        ProcessHeadNode(&proto_node, &Pass, protocol);
        break;

    case RULE_ACTIVATE:

```

```

        ProcessHeadNode(&proto_node, &Activation, protocol);

        break;

    case RULE_DYNAMIC:

        ProcessHeadNode(&proto_node, &Dynamic, protocol);

        break;

    case RULE_UNKNOWN:

        ProcessHeadNode(&proto_node, node->RuleList, protocol);

        break;

    default:

        FatalError("Unable to determine rule type (%s) for processing, exiting!\n",
                  toks[0]);

    }

    rule_count++;

}

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Parsing Rule Options...\n"));

if (rule_type == RULE_UNKNOWN)      /*处理规则选项*/
{
    ParseRuleOptions(rule, node->mode, protocol);
}
else
{
    ParseRuleOptions(rule, rule_type, protocol);
}

mSplitFree(&toks, num_toks);

return;
}

```

函数中又涉及到了一些重要的功能函数，本节将继续分析 ProcessIP,ProcessPort 两个函数，而 ProcessHeadNode 和 ParseRuleOptions 将会再后两节做详细分析。

## 5. 5. 2 ProcessIP

函数 ParseRule 在进行规则头的 IP 地址处理的时候，调用了 ProcessIP 来向 RTN 填充源/目的 IP 地址及掩码的信息，按照《Snort 用户手册》的说明，规则头中 IP 地址信息的相关字符串，可以是：

- 1、192.168.0.1 ->单地址
- 2、192.168.0.0/24 ->含了掩码
- 3、!192.168.0.1 ->不包含此地址
- 4、\$EXTERNAL\_NET ->地址是一个变量
- 5、any ->任意地址
- 6、[[192.168.1.0/24,10.1.1.0/24]] ->一个地址列表

函数将就这几种情况分别进行一一处理，源代码分析如下：

```
/* addr: 规则头中的 IP 地址； mode: 表示是目的地址还是来源地址*/
int ProcessIP(char *addr, RuleTreeNode *rtn, int mode)
{
    char **toks = NULL;
    int num_toks;
    int i;
    IpAddrSet *tmp_addr;
    char *tmp;
    char *enbracket;

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Got address string: %s\n",
                           addr));
```

```

if(*addr == '!')
    /*如果地址是取反的，设置标志位*/
{
    switch(mode)
    {
        case SRC:
            rtn->flags |= EXCEPT_SRC_IP;
            break;

        case DST:
            rtn->flags |= EXCEPT_DST_IP;
            break;
    }

    addr++;           /*跳过取反标志*/
}
}

if(*addr == '$')           /*如果还是一个变量，将其取出来置于 tmp，否则直接置于
tmp*/
{
    if((tmp = VarGet(addr + 1)) == NULL)
    {
        FatalError("%s(%d) => Undefined variable %s\n",
file_name,
file_line, addr);
    }
    else
    {
        tmp = addr;
    }
}

```

```

/* 如果是包含在方括号内的括号，则表示这个是一个地址列表*/
if(*tmp == '[')
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Found IP list!\n"));

    /* *(tmp+strlen(tmp)) = '';*/
    enbracket = strrchr(tmp, (int)')'); /* 检验最后一位是不是反括号*/
    if(enbracket)
        *enbracket = '\x0';
    else
        FatalError("%s(%d) => Unterminated IP List\n", file_name, file_line);

/*把地址列表中的地址提取出来*/
toks = mSplit(tmp+1, ",", 128, &num_toks, 0);

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"mSplit      got      %d
tokens...\n",
                        num_toks));;

for(i=0; i< num_toks; i++) /*把地址逐个提取出来后，递归调用 ParseIP*/
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"adding %s to IP "
                           "address list\n", toks[i]));

    tmp = toks[i];
    while (isspace((int)*tmp)||*tmp=='[') tmp++;
    enbracket = strrchr(tmp, (int)'>'); /* null out the en-bracket */
    if(enbracket)
        *enbracket = '\x0';

    if (strlen(tmp) == 0)

```

```

        continue;

tmp_addr = AllocAddrNode(rtn, mode);

ParseIP(tmp, tmp_addr);

if(tmp_addr->ip_addr == 0 && tmp_addr->netmask == 0)

{

    switch(mode)

    {

        case SRC:

            rtn->flags |= ANY_SRC_IP;

            break;

        case DST:

            rtn->flags |= ANY_DST_IP;

            break;

    }

}

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "Freeing %d tokens...\n",

num_toks));;

mSplitFree(&toks, num_toks);

}

else /*如果不是一个列表的话*/

{

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,

"regular IP address, processing...\n"));

tmp_addr = AllocAddrNode(rtn, mode); /*为 RTN 的来源或目的地址分配地

址空间*/

```

```

ParseIP(tmp, tmp_addr);           /*将 IP 字符串 tmp 拆分后提取出来
源或目的地址，填充 RTN*/
if(tmp_addr->ip_addr == 0 && tmp_addr->netmask == 0)      /*如果为任意地址，设
标志变量*/
{
    switch(mode)
    {
        case SRC:
            rtn->flags |= ANY_SRC_IP;
            break;

        case DST:
            rtn->flags |= ANY_DST_IP;
            break;
    }
}

return 0;
}

```

## 5. 5. 3 ParsePort

端口号可以用几种方法表示，包括"any"端口、静态端口定义、范围、以及通过否定操作符。

"any"端口是一个通配符，表示任何端口。静态端口定义表示一个单个端口号，例如 23 表示 telnet。端口范围用范围操作符": "表示。端口否定操作符用"!"表示。

函数 ParsePort 用于拆分规则头中的端口子串，将分析后的结果填充入 RTN 结构的相应成员变量。源码分析如下：

```

/* prule_port: 指向待分析的端口字符串的指针
** hi_port/lo_port: 高/低端口（因为端口值有可能是一个范围）
** proto: 协议
** not_flag: 端口是否取反（即前面加了一个'!?'）的标志
** 如果是任意端口，则返回 1，否则返回 0
*/
int ParsePort(char *prule_port, u_short *hi_port, u_short *lo_port, char *proto, int *not_flag)
{
    char **toks;          /* token dbl buffer */
    int num_toks;         /* number of tokens found by mSplit() */
    char *rule_port;      /* port string */

    *not_flag = 0;

    /* 检查是否是变量，是的话，调用 VarGet 函数取出*/
    if(!strncmp(prule_port, "$", 1))
    {
        if((rule_port = VarGet(prule_port + 1)) == NULL)
        {
            FatalError("%s(%d) => Undefined variable %s\n", file_name, file_line,
prule_port);
        }
    }
    else
        rule_port = prule_port;

    if(rule_port[0] == '(')           /*不能用括号来表示一个范围*/
    {
        /* user forgot to put a port number in for this rule */

```

```

FatalError("%s(%d) => Bad port number: \"%s\"\n",
           file_name, file_line, rule_port);
}

/* 如果为任意端口，则低/高端口置 0，返回 */
if(!strcasecmp(rule_port, "any"))
{
    *hi_port = 0;
    *lo_port = 0;
    return 1;
}

if(rule_port[0] == '!')          /*如果表示取反，置标志位*/
{
    *not_flag = 1;
    rule_port++;                 /*跳过'!'*/
}

if(rule_port[0] == ':')          /*直接就是 ':' 开始而没有设置低端口，则低端口默认设 0*/
{
    *lo_port = 0;
}

toks = mSplit(rule_port, ":", 2, &num_toks);           /*分析完以上各种情况后，就开始按
':'来拆分了*/

switch(num_toks)
{
    case 1:                  /*如果只提取到了一个子串，那么可能是没有配置低端口或
*/
}

```

```

高端口*/
*hi_port = ConvPort(toks[0], proto);

if(rule_port[0] == ':')           /*如果没有配置低端口，其实刚才已经做过
这一步了*/
{
    *lo_port = 0;
}

else                           /*没有配置高端口， 默认最高为 65535*/
{
    *lo_port = *hi_port;

    if(index(rule_port, ':') != NULL)
    {
        *hi_port = 65535;
    }
}

break;

case 2:             /*返回是两串，则别填充之.....*/
*lo_port = ConvPort(toks[0], proto);

if(toks[1][0] == 0)           /*如果不小心把高端口配置为 0 了，还是默认
最大为 65535*/
{
    *hi_port = 65535;
}
else
{
    *hi_port = ConvPort(toks[1], proto);
}

break;

```

```

    default:

        FatalError("%s(%d) => port conversion failed on \'%s\'\n",
                   file_name, file_line, rule_port);

    }

    mSplitFree(&toks, num_toks);

    return 0;
}

```

## 5. 5. 4 mSplit 功能函数

mSplit 函数是规则分析等许多模块中的一个重要函数。它的主要功能是将指定的字符串，使用指定的分隔标识进行分隔后，把结果存储在二维字符串指数组中返加。源代码分析如下：

```

char **mSplit(char *str, char *sep, int max_strs, int *toks, char meta)

{
    char **retstr;          /* 2D array which is returned to caller */

    char *idx;              /* index pointer into str */

    char *end;               /* ptr to end of str */

    char *sep_end;           /* ptr to end of separator string */

    char *sep_idx;           /* index ptr into separator string */

    int len = 0;             /* length of current token string */

    int curr_str = 0;         /* current index into the 2D return array */

    char last_char = (char) 0xFF;

    /* 对 toks、str 做点合法性检查*/
    if(!toks) return NULL;

```

```

*toks = 0;

if (!str) return NULL;

DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
    "[*] Splitting string: %s\n", str);
    DebugMessage(DEBUG_PATTERN_MATCH, "curr_str = %d\n", curr_str););

/*
 * find the ends of the respective passed strings so our while() loops
 * know where to stop
 */
sep_end = sep + strlen(sep);      /*设置分隔符的结尾位置*/
end = str + strlen(str);         /*设置待分隔字符串的结尾位置*/

/* remove trailing whitespace */
while(isspace((int)*(end - 1)) && ((end - 1) >= str))           /*去除字符串末尾空格*/
    *(--end) = '\0';      /* -1 because of NULL */

/* set our indexing pointers */
sep_idx = sep;                /*设置分隔符字符串索引*/
idx = str;                     /*设置字符串索引指针*/

/*
 * alloc space for the return string, this is where the pointers to the
 * tokens will be stored
 */
if((retstr = (char **) malloc(sizeof(char **) * max_strs))) == NULL) /*为返回的二维字符
串指针数组分配内存空间*/

```

```

FatalPrintError("malloc");

max_strs--;

DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
    "max_strs = %d  curr_str = %d\n",
    max_strs, curr_str));;

/* loop thru each letter in the string being tokenized */

while(idx < end)      /*遍历字符串中的每一个字符*/
{
    /* loop thru each separator string char */

    while(sep_idx < sep_end)      /*当分隔符字符串指针未到该字符串结尾时循环
*/
    {
        /*
         * if the current string-indexed char matches the current
         * separator char...
         */
        if((*idx == *sep_idx) && (last_char != meta))      /*当前字符串中的字符匹配当
前分隔符*/
        {
            /* if there's something to store... */

            if(len > 0)          /*如果非分隔符的字符数目大于 0*/
            {
                DEBUG_WRAP(
                    DebugMessage(DEBUG_PATTERN_MATCH,
                        "Allocating %d bytes for token ", len + 1););
                if(curr_str <= max_strs)      /*如果当前经过分隔处理后所得的字符
*/
            }
        }
    }
}

```

串数目不大于指定的最大返回数目，实际上是最大返回数-1\*/

```
{  
    /* allocate space for the new token */  
    if((retstr[curr_str] = (char *)           /*为当前二维字符串指针  
数组元素分配空间*/  
        malloc(sizeof(char) * len) + 1)) == NULL)  
    {  
        FatalPrintError("malloc");  
    }  
  
    /* copy the token into the return string array */  
    memcpy(retstr[curr_str], (idx - len), len);           /*将对应长度的  
子字符串拷贝到返回的二维字符串数组中*/  
    retstr[curr_str][len] = 0;  
    DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,  
                           "tok[%d]: %s\n", curr_str,  
                           retstr[curr_str]));  
  
    /* twiddle the necessary pointers and vars */  
    len = 0;          /*重新设置相应的变量值和指针。返回的标识字符  
串数目加 1*/  
    curr_str++;  
    DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,  
                           "curr_str = %d\n", curr_str));  
    DebugMessage(DEBUG_PATTERN_MATCH,  
                 "max_strs = %d  curr_str = %d\n",  
                 max_strs, curr_str));  
  
    last_char = *idx;  
    idx++;
```

```

}

DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
    "Checking if curr_str (%d) >= max_strs (%d)\n",
    curr_str, max_strs));;

/*
 * if we've gotten all the tokens requested, return the
 * list
 */
if(curr_str >= max_strs) /*如果当前经过分隔处理后所得的字符串数
目不小于指定的最大返回数目，即我们已经获得了所需的所有标识字符串*/
{
    while(isspace((int) *idx)) /*当前字符串指针跳过空格符
*/
        idx++;

len = end - idx; /*设置拷贝长度值*/
DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
    "Finishing up...\n"));
DebugMessage(DEBUG_PATTERN_MATCH,
    "Allocating %d bytes "
    "for last token ", len + 1););
fflush(stdout);

if((retstr[curr_str] = (char *) /*为返回二维字符串
指针数组的当前元素分配对应的内存空间*/
    malloc(sizeof(char) * len + 1)) == NULL)
    FatalPrintError("malloc");

```

```

        memcpy(retstr[curr_str], idx, len);      /*拷贝对应长度的子字符串
串到返回的二维字符串数组中*/
        retstr[curr_str][len] = 0;

        DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
                                "tok[%d]: %s\n", curr_str,
                                retstr[curr_str])););

*toks = curr_str + 1;                  /*设置返回的标识字符串数目
值*/
        DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
                                "max_strs = %d  curr_str = %d\n",
                                max_strs, curr_str);
        DebugMessage(DEBUG_PATTERN_MATCH,
                                "mSplit got %d tokens!\n", *toks)););

return retstr;                      /*返回二维字符串数组指针*/
}

}

else
/*
 * otherwise, the previous char was a separator as well,
 * and we should just continue
 */
{
    last_char = *idx;
    idx++;
    /* make sure to reset this so we test all the sep. chars */
    sep_idx = sep;
    len = 0;

```

```

        }

    }

else

{

/* go to the next seperator */

sep_idx++;

}

}

sep_idx = sep;           /*重新设置分隔符字符串指针的指向的起始位置*/

len++;                  /*递增非分隔符的字符数目*/

last_char = *idx;

idx++;                  /*递增字符串指针*/



}

/* put the last string into the list */



if(len > 0)

{

DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,

"Allocating %d bytes for last token ", len + 1));



if((retstr[curr_str] = (char *)           /*分配空间*/

malloc(sizeof(char) * len) + 1)) == NULL)

FatalPrintError("malloc");



memcpy(retstr[curr_str], (idx - len), len);          /*拷贝*/

retstr[curr_str][len] = 0;



DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,"tok[%d]:      %s\n",

```

```

curr_str,
retstr[curr_str]););

*toks = curr_str + 1;           /*设置返回的标识字符串数目值*/
}

DEBUG_WRAP(DebugMessage(DEBUG_PATTERN_MATCH,
"mSplit got %d tokens!\n", *toks));;

/* return the token list */

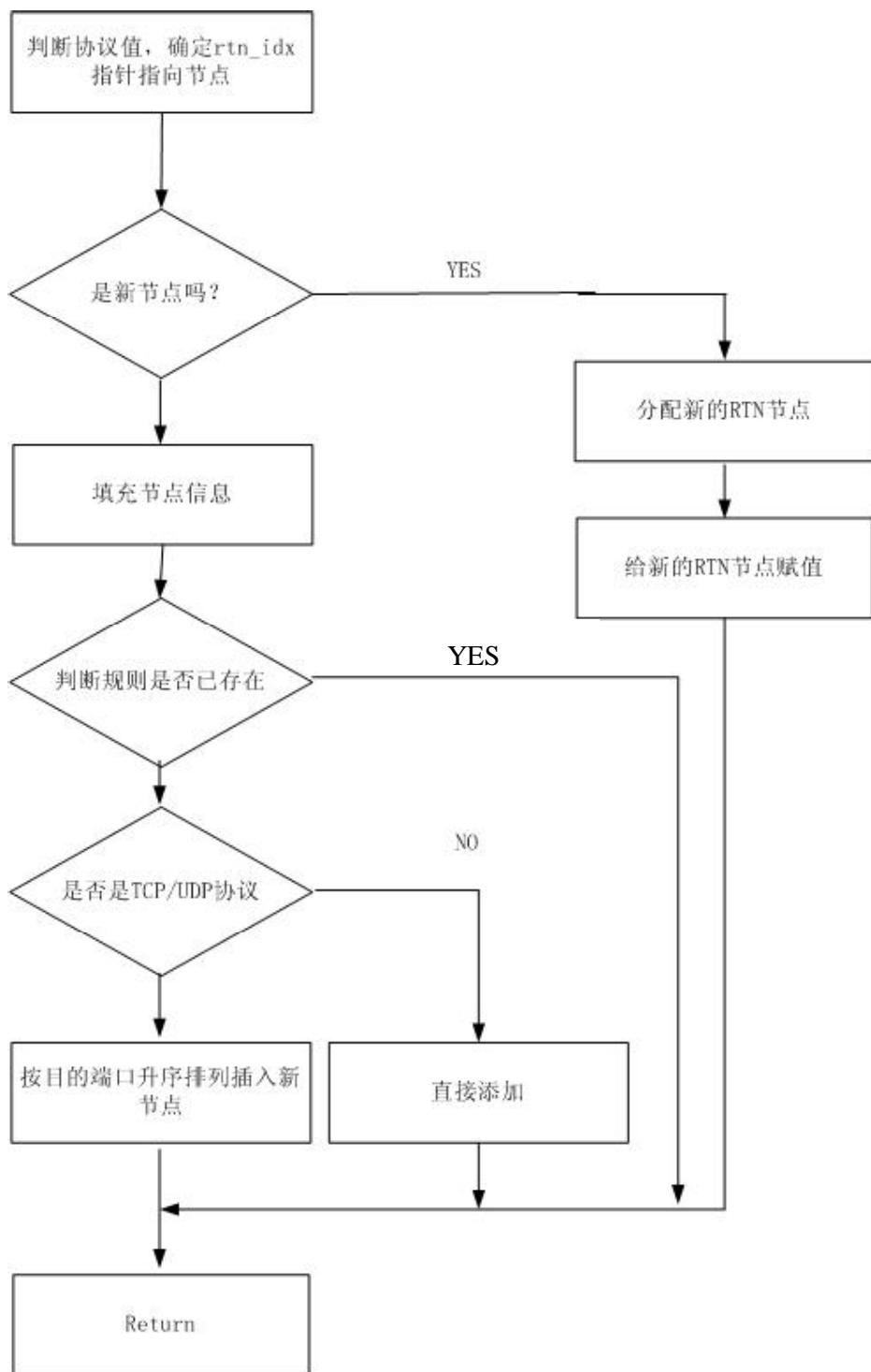
return retstr;
}

```

## 5. 6 ProcessHeadNode

### 5. 6. 1 函数流程

ProcessHeadNode 的功能是用来搭建第三层 RTN 的链表，首先它需要判断第二次节点对应的相应协议的 RTN 节点是否为空，如果是空，则表示一个新节点，为其分配空间，并填充相应成员变量的值。如果不为空，则按照目的端口的升序来将节点插入到已存在链表当中去（当然，如果是 TCP/UDP 协议才有端口）。整个算法流程如下图所示：



源码分析如下：

```
/* test_node: 上层函数中已经填充了地址、掩码、端口等信息的 RTN
```

```

    ** list: 第二节点
    ** protocol: 协议
    */

void ProcessHeadNode(RuleTreeNode * test_node, ListHead * list, int protocol)
{
    int match = 0;
    RuleTreeNode *rtn_idx;
    RuleTreeNode *rtn_prev;
    RuleTreeNode *rtn_head_ptr;
    int count = 0;
    int insert_complete = 0;

#ifndef DEBUG
    int i;
#endif

/* 判断协议，指针 rtn_idx 指向第二层链表 ListHead 相应协议的 RTN */
switch(protocol)
{
    case IPPROTO_TCP:
        rtn_idx = list->TcpList;
        break;

    case IPPROTO_UDP:
        rtn_idx = list->UdpList;
        break;

    case IPPROTO_ICMP:
        rtn_idx = list->IcmpList;
        break;
}

```

```

case ETHERNET_TYPE_IP:

    rtn_idx = list->IpList;

    break;

default:

    rtn_idx = NULL;

    break;

}

rtn_head_ptr = rtn_idx;

if(rtn_idx == NULL)           /*第三层 RTN 为空，则分配空间，填充之*/
{

    head_count++;

switch(protocol)           /*判断相应的协议，为其分配新的空间*/

{

    case IPPROTO_TCP:

        list->TcpList = (RuleTreeNode *) calloc(sizeof(RuleTreeNode),
                                                   sizeof(char));

        rtn_tmp = list->TcpList;

        break;

    case IPPROTO_UDP:

        list->UdpList = (RuleTreeNode *) calloc(sizeof(RuleTreeNode),
                                                   sizeof(char));

        rtn_tmp = list->UdpList;

        break;

    case IPPROTO_ICMP:

```

```

list->IcmpList = (RuleTreeNode *) calloc(sizeof(RuleTreeNode),
                                         sizeof(char));
rtn_tmp = list->IcmpList;
break;

case ETHERNET_TYPE_IP:
list->IpList = (RuleTreeNode *) calloc(sizeof(RuleTreeNode),
                                         sizeof(char));
rtn_tmp = list->IpList;
break;

}

/*把刚才填充的规则头的地址、端口等信息拷贝过来先*/
XferHeader(test_node, rtn_tmp);
rtn_tmp->head_node_number = head_count;

/* 先置空第四层链表，留待规则选项处理函数来搭建*/
rtn_tmp->down = NULL;

/* 添加规则的检测插件的函数*/
SetupRTNFuncList(rtn_tmp);

rtn_tmp->listhead = list;           /*回指针指向第二层*/
return;
}

/*如果链表已有节点存在，测试待填充的规则头是否已存在 */
match = TestHeader(rtn_idx, test_node);

```

```

while((rtn_idx->right != NULL) && !match)
{
    count++;
    match = TestHeader(rtn_idx, test_node);

    if(!match)
        rtn_idx = rtn_idx->right;
    else
        break;
}

match = TestHeader(rtn_idx, test_node);           /*是否重复了*/

if(!match)           /*没有重复，添加之*/
{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Building New Chain
head node\n"));

    head_count++;      /*计数器累加*/

    /*开始搭建新节点，为新的 RTN 节点分配空间*/
    rtn_tmp = (RuleTreeNode *) calloc(sizeof(RuleTreeNode),
sizeof(char));

    if(rtn_tmp == NULL)           /*分配失败*/
    {
        FatalError("Unable to allocate Rule Head Node!!\n");
    }
}

```

```

/*把已经填充好了的地址、端口等信息拷贝过来*/
XferHeader(test_node, rtn_tmp);

rtn_tmp->head_node_number = head_count;

rtn_tmp->down = NULL;

/* initialize the function list for the new RTN */

SetupRTNFuncList(rtn_tmp);

/* add link to parent listhead */
rtn_tmp->listhead = list;      /*回指针， 指向第二层链表*/

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
    "New Chain head flags = 0x%X\n", rtn_tmp->flags));;

/* 如果是 TCP/UDP 协议， 按照目的端口的升序排列插入新节点*/
if(protocol == IPPROTO_TCP || protocol == IPPROTO_UDP)
{
    rtn_idx = rtn_head_ptr;
    rtn_prev = NULL;
    insert_complete = 0;

    if(rtn_tmp->flags & EXCEPT_DST_PORT)      /*如果目的端口取反，则直接添
加至末尾*/
    {
        switch(protocol)
        {
            case IPPROTO_TCP:
                rtn_tmp->right = list->TcpList;
                list->TcpList = rtn_tmp;
                break;
        }
    }
}

```

```

        case IPPROTO_UDP:
            rtn_tmp->right = list->UdpList;
            list->UdpList = rtn_tmp;
            break;
        }

        rtn_head_ptr = rtn_tmp;
        insert_complete = 1;
    }

    else
    {
        while(rtn_idx != NULL)      /*遍历整个 rtn_idx, 比较链表中的目的低端口
(rtn_idx->ldp) 与待插入的目的低端口 (rtn_tmp->ldp) */
        {

            /*如果小于, 继续循环*/
            if(rtn_idx->flags & EXCEPT_DST_PORT ||
               rtn_idx->ldp < rtn_tmp->ldp)
            {
                rtn_prev = rtn_idx;
                rtn_idx = rtn_idx->right;
            }

            /*如果等于, 直接添加*/
            else if(rtn_idx->ldp == rtn_tmp->ldp)
            {
                rtn_tmp->right = rtn_idx->right;
                rtn_idx->right = rtn_tmp;
                insert_complete = 1;
                break;
            }
        }
    }
}

```

```

    }

/*如果是大于，则插入进来*/

else

{

    rtn_tmp->right = rtn_idx;

    if(rtn_prev != NULL)

    {

        rtn_prev->right = rtn_tmp;

    }

    else

    {

        switch(protocol)

        {

            case IPPROTO_TCP:

                list->TcpList = rtn_tmp;

                break;

            case IPPROTO_UDP:

                list->UdpList = rtn_tmp;

                break;

        }

        rtn_head_ptr = rtn_tmp;

    }

    insert_complete = 1;

}

break;
}

```

```

    }

}

/*循环完了，但是待插入的是最大的，则直接添加至末尾*/
if(!insert_complete)
{
    rtn_prev->right = rtn_tmp;
}

rtn_idx = rtn_head_ptr;

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
    "New %s node inserted, new order:\n",
    protocol == IPPROTO_TCP?"TCP":"UDP"));

#endif DEBUG

i = 0;

while(rtn_idx != NULL)
{
    if(rtn_idx->flags & EXCEPT_DST_PORT)
    {
        LogMessage("!");
    }

    DebugMessage(DEBUG_CONFIGRULES, "%d ", rtn_idx->ldp);
    rtn_idx = rtn_idx->right;
    if(i++ == 10)
    {
        DebugMessage(DEBUG_CONFIGRULES, "\n");
    }
}

```

```

        i = 0;

    }

}

DebugMessage(DEBUG_CONFIGRULES, "\n");

#endif

}

else /*如果不是 TCP/UDP 协议，直接添加*/
{
    rtn_idx->right = rtn_tmp;
}

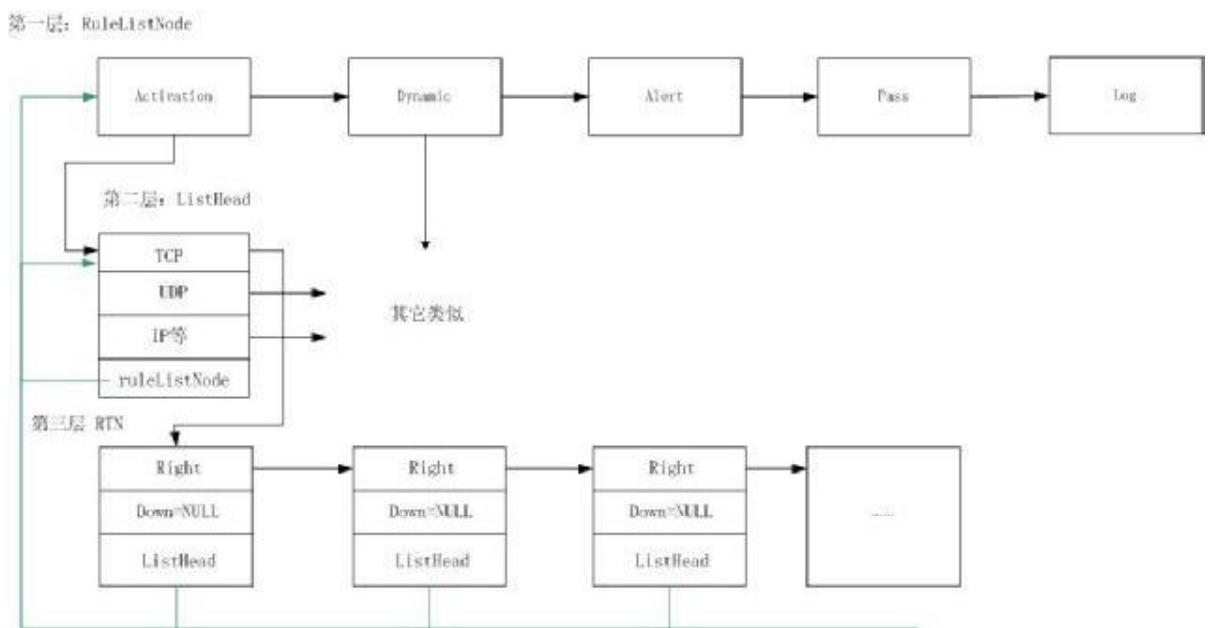
else /*处理规则头重复的情况*/
{
    rtn_tmp = rtn_idx;

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
                           "Chain head %d  flags = 0x%X\n", count, rtn_tmp->flags););

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,
                           "Adding options to chain head %d\n", count););
}
}

```

函数执行完毕后，规则链表的第三层（RTN）也搭建完成了，完成后的链表示意图如下所示：



## 5. 6. 2 规则链表头检测匹配函数列表的配置

在构建 RTN 节点的时候，调用了一个重要的功能函数 `SetupRTNFuncList`。它的作用是添加匹配规则头的函数列表。我们说过，一个规则头包含了地址、端口、数据流方向等要素，在进行数据包检测的时候，需要对被检测数据包的对应的这些要素一一匹配，这些都是由 RTN 的检测函数来完成的，如 `CheckBidirectional`、`CheckSrcIp` 等等。而 `SetupRTNFuncList` 函数的作用就是先把它们添加至对应 RTN 节点的函数列表当中，以待检测引擎匹配的时候来逐一调用。同样的，我们下一节将要将到的 OTN 节点，也是做了相同的处理。源码分析如下（有删节）：

```
void SetupRTNFuncList(RuleTreeNode * rtn)
{
    /*如果设置了双向地址操作符标识，则将对应检测函数 CheckBidirectional 添加到
    函数列表中*/
    if(rtn->flags & BIDIRECTIONAL)
    {
        AddRuleFuncToList(CheckBidirectional, rtn);
    }
    else
    {
        /*根据任意端口标识符和端口求反标识符的设置情况，添加目的端口检测函数
        */
        PortToFunc(rtn, (rtn->flags & ANY_DST_PORT ? 1 : 0),
                   (rtn->flags & EXCEPT_DST_PORT ? 1 : 0), DST);
        /*根据任意端口标识符和端口求反标识符的设置情况，添加来源端口检测函数
        */
        PortToFunc(rtn, (rtn->flags & ANY_SRC_PORT ? 1 : 0),
                   (rtn->flags & EXCEPT_SRC_PORT ? 1 : 0), SRC);
        /*添加来源地址检测函数*/
        AddrToFunc(rtn, SRC);
        /*添加目的地址检测函数*/
        AddrToFunc(rtn, DST);
    }
}
```

```

/*添加结束标志符，因为函数 RuleListEnd 什么也不做，只是 return 1*/
AddRuleFuncToList(RuleListEnd, rtn);
}

```

PortToFunc 函数的作用是为当前规则添加端口检测函数进列表。源码分析如下：

```

void PortToFunc(RuleTreeNode * rtn, int any_flag, int except_flag, int mode)
{
    if(any_flag)          /*若是任意端口，则直接返回*/
        return;

    /*如果是端口求反标志位 */
    if(except_flag)
    {
        switch(mode)      /*判断是来源端口还是目的端口，添加对应函数*/
        {
            case SRC:
                AddRuleFuncToList(CheckSrcPortNotEq, rtn);
                break;

            case DST:
                AddRuleFuncToList(CheckDstPortNotEq, rtn);
                break;
        }
        return;
    }

    /*判断是来源端口还是目的端口，添加对应函数*/
    switch(mode)
    {
        case SRC:
            AddRuleFuncToList(CheckSrcPortEqual, rtn);
            break;

        case DST:
            AddRuleFuncToList(CheckDstPortEqual, rtn);
            break;
    }
}

```

函数 AddrToFunc 的作用是为当前规则添加地址的检测函数进列表。源码分析如下：

```
void AddrToFunc(RuleTreeNode * rtn, int mode)
{
    /*
     * if IP and mask are both 0, this is a "any" IP and we don't need to
     * check it
     */
    switch(mode)      /*判断是来源地址还是目的地址，添加对应匹配函数*/
    {
        case SRC:
            if((rtn->flags & ANY_SRC_IP) == 0)
            {
                AddRuleFuncToList(CheckSrcIP, rtn);
            }
            break;

        case DST:
            if((rtn->flags & ANY_DST_IP) == 0)
            {
                AddRuleFuncToList(CheckDstIP, rtn);
            }
            break;
    }
}
```

从以上分析我们还可以看到，最终实现函数列表的添加的函数是 AddRuleFuncToList。这里，函数列表的组织算法，同样是采用了链表。函数源码分析如下：

```
void AddRuleFuncToList(int (*func) (Packet *, struct _RuleTreeNode *, struct _RuleFpList *), RuleTreeNode * rtn)
{
    RuleFpList *idx;

    idx = rtn->rule_func;           /*取得首节点地址*/
```

```

if(idx == NULL)      /*若首节点为空，分配空间，添加函数*/
{
    rtn->rule_func = (RuleFpList *) calloc(sizeof(RuleFpList), sizeof(char));
    rtn->rule_func->RuleHeadFunc = func;
}
else      /*否则循环至列表末尾，分配空间添加函数*/
{
    while(idx->next != NULL)
        idx = idx->next;

    idx->next = (RuleFpList *) calloc(sizeof(RuleFpList), sizeof(char));

    idx = idx->next;
    idx->RuleHeadFunc = func;
}
}

```

这样，RTN 的检测函数列表构建就完成了，在后面的章节中，我们会继续分析这些函数是如何被调用以及它们的具体的实现。

## 5. 7 ParseRuleOptions

### 5. 7. 1 函数流程分析

ParseRuleOptions 函数用于搭建第四层链表 OTN，也就是规则选项的相关内容。如 5.5 节分析中所述，系统在 ParseRul 函数中执行：

```

if (rule_type == RULE_UNKNOWN)
    ParseRuleOptions(rule, node->mode, protocol);
else
    ParseRuleOptions(rule, rule_type, protocol);

```

来调用 ParseRuleOptions 函数处理规则选项。

函数源代码分析如下：

```
void ParseRuleOptions(char *rule, int rule_type, int protocol)
{
    char **toks = NULL;
    char **opts = NULL;
    char *idx;
    char *aux;
    int num_toks, original_num_toks=0;
    int i;
    int num_opts;
    int found = 0;
    OptTreeNode *otn_idx;
    KeywordXlateList *kw_idx;
    THDX_STRUCT thdx;
    int one_threshold = 0;

    /* set the OTN to the beginning of the list */
    otn_idx = rtn_tmp->down;

    /*
     * make a new one and stick it either at the end of the list or hang it
     * off the RTN pointer
     */
    if(otn_idx != NULL)          /*OTN 不为空的话，循环遍历至链表末尾，再为新节点分配空间*/
    {
        /* loop to the end of the list */
        while(otn_idx->next != NULL)
```

```

    {

        otn_idx = otn_idx->next;

    }

/* setup the new node */

otn_idx->next = (OptTreeNode *) calloc(sizeof(OptTreeNode),
                                         sizeof(char));

/* set the global temp ptr */

otn_tmp = otn_idx->next;

if(otn_tmp == NULL)

{

    FatalError("Unable to alloc OTN: % s", strerror(errno));

}

otn_tmp->next = NULL;

opt_count++;


}

else      /*若为空，则直接为首节点分配空间*/

{

/* first entry on the chain, make a new node and attach it */

otn_idx = (OptTreeNode *) calloc(sizeof(OptTreeNode), sizeof(char));

bzero((char *) otn_idx, sizeof(OptTreeNode));

otn_tmp = otn_idx;

if(otn_tmp == NULL)

```

```

{
    FatalError("Unable to alloc OTN!\n");
}

otn_tmp->next = NULL;
rtn_tmp->down = otn_tmp;
opt_count++;

}

otn_tmp->chain_node_number = opt_count;      /*设置当前选项节点的节点编号*/
otn_tmp->type = rule_type;                  /*设置当前选项节点的规则类型字段值
*/
otn_tmp->proto_node = rtn_tmp;
otn_tmp->event_data.sig_generator = GENERATOR_SNORT_ENGINE;

/* add link to parent RuleTreeNode */
otn_tmp->rtn = rtn_tmp;          /*回指向父节点 RTN*/

/* find the start of the options block */
idx = index(rule, '(');        /*从规则选项字符串中查找开始位置*/
i = 0;

if(idx != NULL)           /*规则选项存在吗 ? */
{
    /* 设置这一大堆标志变量的作用在于，在设置选项节点对应的值的时候，确保它们有且只有执行了一次
*/
    int one_msg = 0;
    int one_logto = 0;
    int one_activates = 0;
    int one_activated_by = 0;
}

```

```

int one_count = 0;
int one_tag = 0;
int one_sid = 0;
int one_rev = 0;
int one_priority = 0;
int one_classtype = 0;
int one_stateless = 0;

idx++;

/*判断末尾字符的完整性*/
aux = strrchr(idx, ')');

if(aux == NULL)
{
    FatalError("%s(%d): Missing trailing ')' in rule: %s.\n",
              file_name, file_line, rule);
}

*aux = 0;

/*
 * 按分号提取规则选项的各关键字子串，这样每一个 toks[i]中就是一项** 关键字
 * 的配置，留等后面继续分解
 */
toks = mSplit(idx, ";", 64, &num_toks, '\\');
original_num_toks = num_toks;
DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "      Got %d tokens\n",
                        num_toks););
/* decrement the number of toks */
num_toks--;

```

```

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"Parsing options list: "));

while(num_toks)
{
    char* option_name = NULL;
    char* option_args = NULL;

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"      option: %s\n",
toks[i]));;

/*规则选项关键字和它们的参数用冒号": "分开*/
    opts = mSplit(toks[i], ":", 4, &num_opts, '\\');
    option_name = opts[0];
    option_args = opts[1];

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"      option  name:
% s\n", option_name));;
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"      option  args:
% s\n", option_args));;

    while(isspace((int) *option_name))           /*跳过空格*/
        option_name++;

/* 以下匹配各关键字 */
    if(!strcasecmp(option_name, "msg"))          /*若为 msg 关键字*/
    {
        ONE_CHECK (one_msg, option_name);

        if(num_opts == 2)

```

```

    {

        ParseMessage(option_args); /*函数的作用是将规则中的消息字符串
拷贝到当前规则链表选项节点结构(OTN)中的对应字段中*/
    }

    else

    {

        FatalError("\n%s(%d) => No argument passed to "
                  "keyword \"%s\"\nMake sure you didn't forget a ':' "
                  "or the argument to this keyword!\n", file_name,
                  file_line, option_name);

    }

}

else if(!strcasecmp(option_name, "logto")) /*若为 logto 关键字*/
{

    ONE_CHECK (one_logto, option_name);

    if(num_opts == 2)

    {

        ParseLogto(option_args); /*函数的功能是将规则指定的日志
文件名称拷贝到当前列表选项节点的对应字段中*/
    }

    else

    {

        FatalError("\n%s(%d) => No argument passed to "
                  "keyword \"%s\"\nMake sure you didn't forget a ':' "
                  "or the argument to this keyword!\n", file_name,
                  file_line, option_name);

    }

}

else if(!strcasecmp(option_name, "activates")) /*若为 activates 关键字*/
{

```

```

ONE_CHECK (one_activates, option_name);

if(num_opts == 2)
{
    ParseActivates(option_args);
    dynamic_rules_present++;
}

else
{
    FatalError("\n%s(%d) => No argument passed to "
              "keyword \"%s\"\nMake sure you didn't forget a ':' "
              "or the argument to this keyword!\n", file_name,
              file_line, option_name);
}

else if(!strcasecmp(option_name, "activated_by"))
{
    ONE_CHECK (one_activated_by, option_name);

    if(num_opts == 2)
    {
        ParseActivatedBy(option_args);
        dynamic_rules_present++;
    }

    else
    {
        FatalError("\n%s(%d) => No argument passed to "
                  "keyword \"%s\"\nMake sure you didn't forget a ':' "
                  "or the argument to this keyword!\n", file_name,
                  file_line, opts[0]);
    }
}

```

```

else if(!strcasecmp(option_name, "count"))

{
    ONE_CHECK (one_count, option_name);

    if(num_opts == 2)

    {
        if(otn_tmp->type != RULE_DYNAMIC)

            FatalError("The \"count\" option may only be used with "

                    "the dynamic rule type!\n");

        ParseCount(opts[1]);

    }

    else

    {
        FatalError("\n%s(%d) => No argument passed to "

                "keyword \"%s\"\nMake sure you didn't forget a ':' "

                "or the argument to this keyword!\n", file_name,

                file_line, opts[0]);

    }

}

else if(!strcasecmp(option_name, "tag"))

{
    ONE_CHECK (one_tag, opts[0]);

    if(num_opts == 2)

    {
        ParseTag(opts[1], otn_tmp);

    }

    else

    {
        FatalError("\n%s(%d) => No argument passed to "

                "keyword \"%s\"\nMake sure you didn't forget a ':' "

                "or the argument to this keyword!\n", file_name,

```

```

        file_line, opts[0]);
    }

}

else if(!strcasecmp(option_name, "threshold"))

{

ONE_CHECK (one_threshold, opts[0]);

if(num_opts == 2)

{

ParseThreshold2(&thdx, opts[1]);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

file_line, opts[0]);

}

}

else if(!strcasecmp(option_name, "sid"))

{

ONE_CHECK (one_sid, opts[0]);

if(num_opts == 2)

{

ParseSID(opts[1], otn_tmp);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

```

```

        file_line, opts[0]);
    }

}

else if(!strcasecmp(option_name, "rev"))

{

ONE_CHECK (one_rev, opts[0]);

if(num_opts == 2)

{

ParseRev(opts[1], otn_tmp);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

file_line, opts[0]);

}

}

else if(!strcasecmp(option_name, "reference"))

{

if(num_opts == 2)

{

ParseReference(opts[1], otn_tmp);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

file_line, opts[0]);

```

```

    }

}

else if(!strcasecmp(option_name, "priority"))

{

ONE_CHECK (one_priority, opts[0]);

if(num_opts == 2)

{

ParsePriority(opts[1], otn_tmp);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

file_line, opts[0]);

}

else if(!strcasecmp(option_name, "classtype"))

{

ONE_CHECK (one_classtype, opts[0]);

if(num_opts == 2)

{

ParseClassType(opts[1], otn_tmp);

}

else

{

FatalError("\n%s(%d) => No argument passed to "

"keyword \"%s\"\nMake sure you didn't forget a ':' "

"or the argument to this keyword!\n", file_name,

file_line, option_name);

```

```

        }

    }

    else if(!strcasecmp(option_name, "stateless"))

    {

        ONE_CHECK (one_stateless, opts[0]);

        otn_tmp->stateless = 1;

    }

    else

    {

        kw_idx = KeywordList;      /*取得指向关键字列表的指针*/

        found = 0;

        while(kw_idx != NULL)      /*若关键字列表不为空，遍历之*/

        {

            DEBUG_WRAP(DebugMessage(DEBUG_INIT, "comparing: \"%s\" =>

                \"%s\"\n",

                option_name, kw_idx->entry.keyword));


            /*如果找到当前节点中的关键字与待拆解规则的选项关键字匹配，则

            调用当前列表元素中的初始化函数。并设置是否查找到的标志 found*/



            if(!strcasecmp(option_name, kw_idx->entry.keyword))

            {

                if(num_opts == 2)

                {

                    kw_idx->entry.func(option_args, otn_tmp, protocol);

                }

                else

                {

                    kw_idx->entry.func(NULL, otn_tmp, protocol);

                }

            }

        }

    }

}

```

```

        DEBUG_WRAP(DebugMessage(DEBUG_INIT, "%s->",

kw_idx->entry.keyword));;

        found = 1;

        break;

    }

    kw_idx = kw_idx->next;

}

if(!found) /*遍历完了还没有找到*/
{
    /* Unrecognized rule option, complain */
    FatalError("Warning: %s(%d) => Unknown keyword '%s' in "
               "rule!\n", file_name, file_line, opts[0]);
}

}

mSplitFree(&opts,num_opts);

--num_toks;

i++;

}

DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"OptListEnd\n"));

AddOptFuncToList(OptListEnd, otn_tmp);

}

else

{
    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES,"OptListEnd\n"));

    AddOptFuncToList(OptListEnd, otn_tmp);

}

```

```

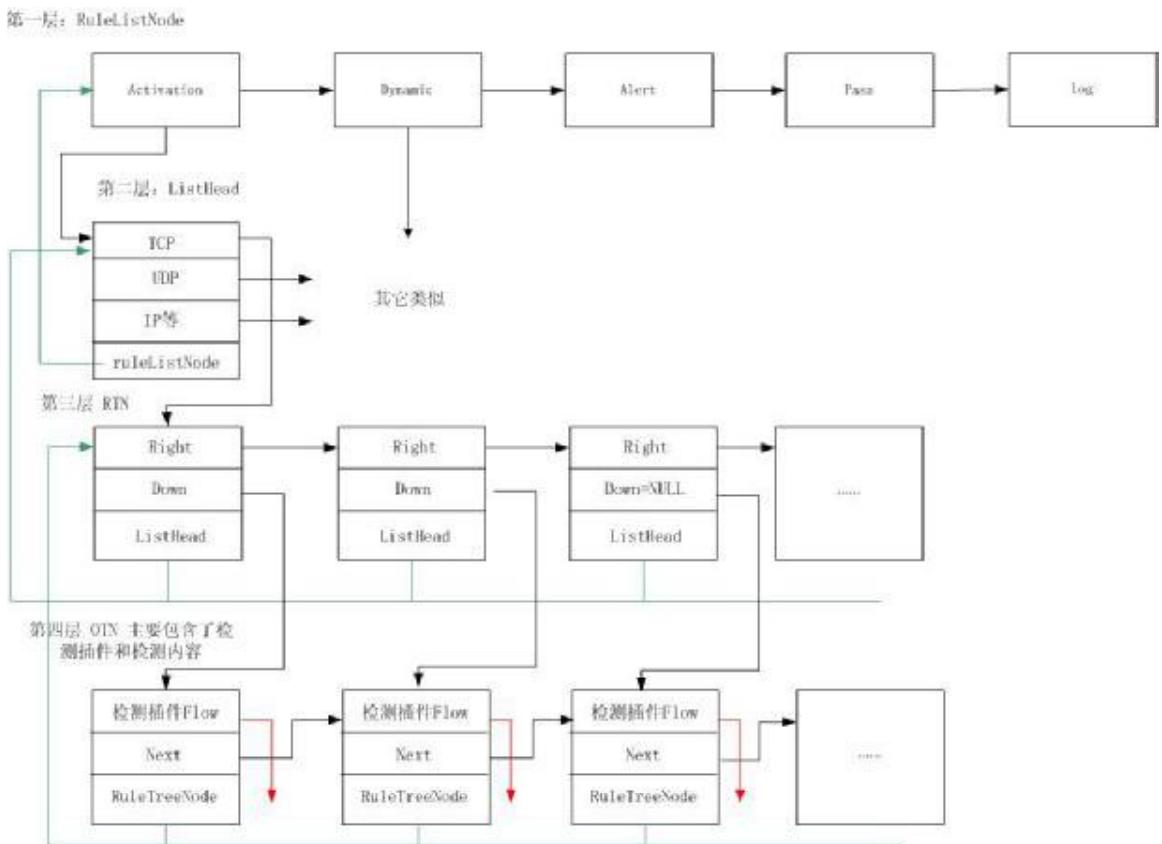
if( one_threshold )
{
    int rstat;
    thdx.sig_id = otn_tmp->sigInfo.id;
    thdx.gen_id = GENERATOR_SNORT_ENGINE;
    if( (rstat=sfthreshold_create( &thdx )) )
    {
        if( rstat == THD_TOO_MANY_THDOBJ )
        {
            FatalError("Rule-Threshold-Parse: could not create a threshold object -- only one
per sid, sid = %u\n",thdx.sig_id);

        }
        else
        {
            FatalError("Unable to add Threshold object for Rule-sid =  %u\n",thdx.sig_id);
        }
    }
}

if(idx != NULL)
{
    mSplitFree(&toks,original_num_toks);
}

```

这样，函数逐个提取关键字列表进行分析，然后调用对应处理函数进行拆解，OTN 节点的字段值一一被填充相应的值。这样，第四层链表就搭建完成了，完成后的链表如下图所示：



## 5. 7. 2 ParseMessage

该函数的功能是将规则中的消息字符串拷贝到当前规则链表选项节点结构中的对应字段中。源代码分析如下：

```
void ParseMessage(char *msg)
{
    char *ptr;
    char *end;
    int size;
    int count = 0;
    char *read;
    char *write;
```

```

/* figure out where the message starts */

ptr = index(msg, "");           /*找到字符串起始位置*/

if(ptr == NULL)
{
    ptr = msg;
}

else
    ptr++;

end = index(ptr, "");           /*找到开尾位置*/

if(end != NULL)
    *end = 0;

while(isspace((int) *ptr))      /*跳过对应的字符串*/
    ptr++;

read = write = ptr;

while(read < end)              /*重组字符串，跳过'\'*/
{
    if(*read == '\\')
    {
        read++;
        count++;
    }

    if(read >= end)
    {

```

```

        break;

    }

}

*write++ = *read++;

}

if(end)

{
    *(end - count) = '\x0';

}

/* find the end of the alert string */

size = strlen(msg) + 1;           /*取得 msg 的长度*/
DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "Message: %s\n", msg));

/* alloc space for the string and put it in the rule */
if(size > 0)

{
    otn_tmp->sigInfo.message = strdup(ptr);      /*填充 OTN 的 msg 字段值*/

    DEBUG_WRAP(DebugMessage(DEBUG_CONFIGRULES, "Rule message set to:
%s\n",
                           otn_tmp->sigInfo.message););
}

else

{
    ErrorMessage("%s(%d): bad alert message size %d\n", file_name,
                 file_line, size);
}

```

```
    }

    return;
}
```

其它的提取的 OTN 关键字的 ParseXXX 函数运行流程与此相似。读者可以自行分析。

这样，在逐层调用后，规则的初始化工作就完成了。以 Snort 的 1.X 版本中，至此，就开始检测了。每一个数据包经过嗅探引擎、拆包引擎以预处理后，就开始遍历这个链表。但是，从 2.0 版本开始，引入了快速匹配引擎设计。为什么不直接匹配规则链表，而要再多做一项工作呢？请参见下一章分析。

## 第6章 构建规则快速配匹引擎

### 6. 1 概述

#### 1、为什么要设计规则快速匹配引擎

设计规则快速成匹配引擎的目的主要在于：**通过某种技术手段，对规则链表进行二次分类，以提高规则的检测匹配效率。**

Snort 从 2.0 版本开始引入快速规则匹配引擎。其设计基础是如何有效地划分规则集合。系统采用的基本思想是通过规则中的目的端口和源端口值来划分类别，其要点如下：

- 1) 如果源端口值为特定值，目的端口值为 ANY，则该规则加入到源端口值对应的子集合中。如果目的端口也为特定值，则同时将该规则加入到目的端口对应的子集合中。
- 2) 如果目的端口为特定值，而源端口为任意值 ANY，则该规则加入到目的端口对应的子集合中。如果源端口也为特定值，则同时将该规则加入到源端口对应的子集合中。
- 3) 如果目的端口和源端口都为任意值 ANY，则将规则加入到通用子集合

中。

- 4) 对于规则中端口值求反操作或者指定值范围的情况，等同于端口值为 ANY 情况加以处理。

如果规则中不含端口值，如 ICMP 或 IP 协议类型的规则，采用如下处理方法：

- 1) 对于 ICMP 协议规则，如果规则中指定了 ICMP 类型值，则目的端口值指定为 ICMP 类型值，否则，规则为任意值 ANY；
- 2) 对于 IP 协议规则，如果规则中指定了 IP 高层类型值，则目的端口指定为 IP 高层协议类型值；否则，规则为任意值 ANY；
- 3) 两者规则划分时，源端口值都指定为任意值 ANY；

## 2、重要的数据结构。

快速规则匹配引擎，主要包含了四个重要的数据结构：

- a) PORT\_RULE\_MAP：顶层数据结构；
- b) PORT\_GROUP：负责存放根据特定端口值所划分的规则子集合；
- c) RULE\_NODE：规则节点链表；
- d) OTNX：

PORT\_RULE\_MAP 定义如下：

```
#define MAX_PORTS 64*1024  
#define ANYPORT -1  
  
typedef struct {  
  
    int      prmNumDstRules;           //各子集合计数器  
    int      prmNumSrcRules;  
    int      prmNumGenericRules;  
  
    int      prmNumDstGroups;  
    int      prmNumSrcGroups;  
  
    PORT_GROUP *prmSrcPort[MAX_PORTS]; //源端口子集合
```

```

PORT_GROUP *prmDstPort[MAX_PORTS];           //目的端口子集合

/* char      prmConflicts[MAX_PORTS]; */ //通用子集合

PORT_GROUP *prmGeneric;

} PORT_RULE_MAP;

```

这样，每一类协议（tcp/udp/icmp/ip）都对应了一个 PORT\_RULE\_MAP 结构，系统针对每个协议，定义了一个全局变量（fpCreate.c）：

```

static PORT_RULE_MAP*prmTcpRTNX = NULL;

static PORT_RULE_MAP*prmUdpRTNX = NULL;

static PORT_RULE_MAP*prmIpRTNX = NULL;

static PORT_RULE_MAP*prmIcmpRTNX= NULL;

```

也就是说，构建快速规则匹配引擎，主要就是针对这四个全局变量进行值的填充。

根据特定端口划分的子集合，主要存放在 PORT\_GROUP 结构中，PORT\_GROUP 结构定义在 pcrm.h 中：

```

typedef struct {

    /* Content List */

    RULE_NODE *pgHead, *pgTail, *pgCur;

    int pgContentCount;

    /* No-Content List */

    RULE_NODE *pgHeadNC, *pgTailNC, *pgCurNC;

    int pgNoContentCount;

    /* Uri-Content List */

    RULE_NODE *pgUriHead, *pgUriTail, *pgUriCur;

    int pgUriContentCount;
}

```

```

/* Setwise Pattern Matching data structures */

void * pgPatData;

void * pgPatDataUri;

int avgLen;

int minLen;

int maxLen;

int c1,c2,c3,c4,c5;

/*
** Bit operation for validating matches
*/
BITOP boRuleNodeID;

/*
* Not rule list for this group
*/
NOT_RULE_NODE *pgNotRuleList;

/*
** Count of rule_node's in this group/list
*/
int pgCount;

int pgNQEvents;

int pgQEvents;

}PORT_GROUP;

```

结构中主要包含了规则节点（三种规则节点 RULE\_NODE：包含 content，包含 uricontent 和没有内容匹配选项的规则）及各计数器；

RULE\_NODE 结构定义如下 (pcrm.h):

```
typedef struct _rule_node_ {  
  
    struct _rule_node_ * rnNext;  
  
    RULE_PTR rnRuleData;  
  
    int iRuleNodeID;  
  
} RULE_NODE;
```

其中，字段 mRuleData 定义为空指针类型，但是在实际操作中，其实赋予的是 OTNX 类型的指针值。而 OTNX 结构类型中包含的就是分别指向特定规则所对应的链表头节点和选项节点的指针值，其定义如下：

```
typedef struct _otnx_ {  
  
    OptTreeNode * otn;  
  
    RuleTreeNode * rtn;  
  
    unsigned int content_length;  
  
} OTNX;
```

从这个结构的定义，我们可以看出，根据划分标准而得到的各个规划子集合中，实际包含的都是 OTNX 结构类型的指针值，所指向的是初始规则链表结构中的对应结点。如此，将快速规则匹配引擎中的新构建的数据结构与原始的规则链表数据结构联系起来。其中，OTNX 数据结构就是关键的桥梁。

## 6. 2 fpCreateFastPacketDetection

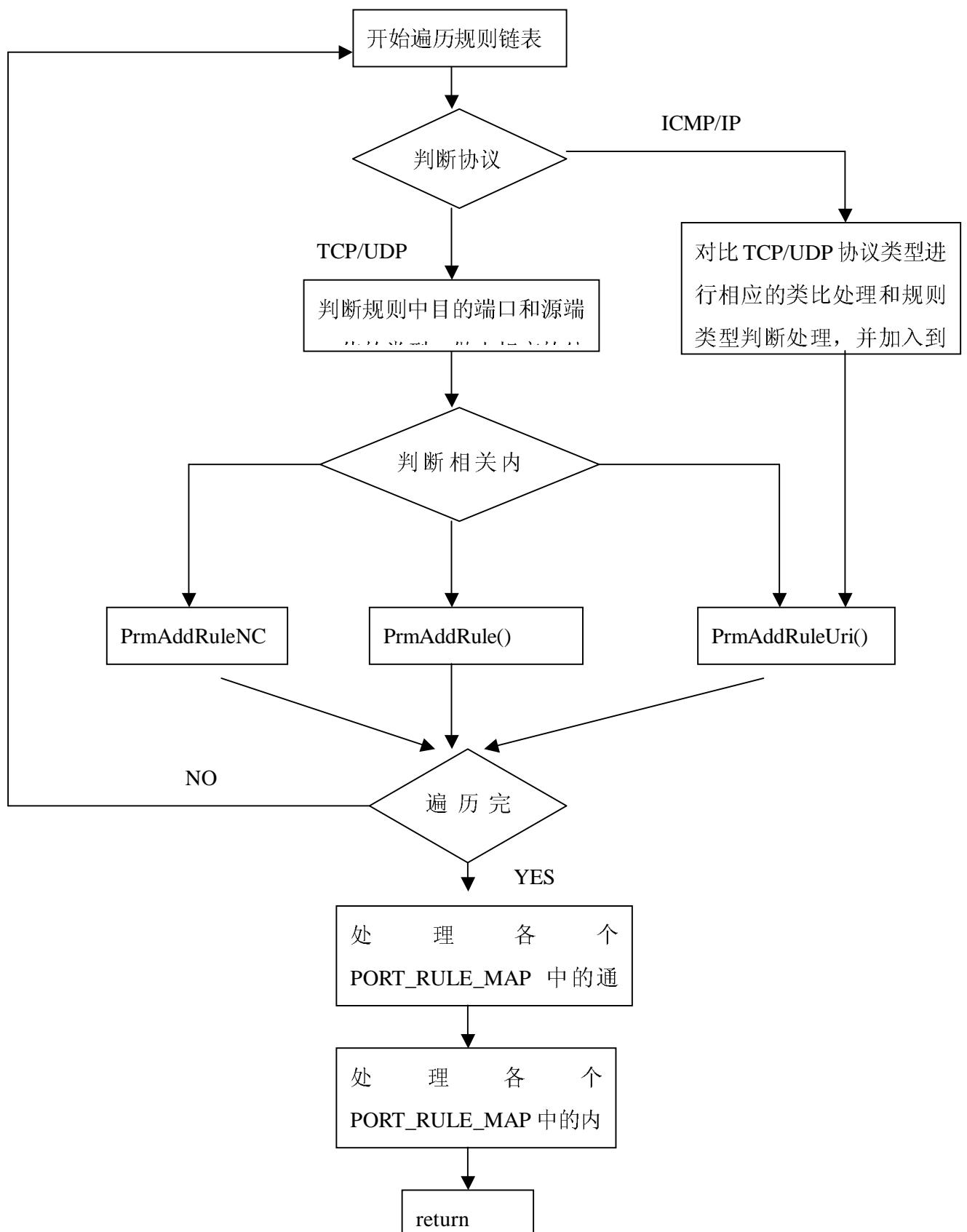
fpCreateFastPacketDetection 函数是整个快速匹配引擎构建的框架函数。其定

义在 fpCreate.c 中， 函数主体框架如下：

```
for (rule=RuleLists; rule; rule=rule->next)      //开始遍历规则链表
{
    if(!rule->RuleList)
        continue;
    if(rule->RuleList->TcpList)          //如果是 TCP 链， 遍历其 RTN 层
    {
        for(rtn = rule->RuleList->TcpList; rtn != NULL; rtn = rtn->right)
        {
            //检测源目的高低端口， 并判断之.....
            sport = CheckPorts(rtn->hsp, rtn->lsp);
            dport = CheckPorts(rtn->hdp, rtn->ldp);
            //遍历 OTN 链表， 按照 Content/UriContent, or NoContent 的不同//分别调
            //用对应的函数
            for( otn = rtn->down; otn; otn=otn->next )
            {
                otnx->otn = otn;      //指向特定规则所对应的选项节点
                otnx->rtn = rtn;      //指向特定规则所对应的链表头节点
                if( OtnHasContent( otn ) )      // 如 果 包 含 了 Content
                {
                    prmAddRule(prmTcpRTNX, dport, sport, otnx);
                }
                .....
            }
        }
        if(rule->RuleList->UdpList)          //如果是 UDP 链， 遍历其 RTN 层
        {
            .....
        }
        if(rule->RuleList->IcmpList)          //如果是 ICMP 链， 遍历其 RTN 层
```

```
{  
    .....  
}  
if(rule->RuleList->IpList)           //如果是 IP 链，遍历其 RTN 层  
{  
    .....  
}
```

算法流程如下图所示：



## 6. 3 prmAddRuleXX

如上分析，在函数 fpCreateFastPacketDetection 中，判断了规则是否包含 content 后，就调用 prmAddRuleXX 函数。

prmAddRuleXX 函数主要包括了 prmAddRule、prmAddRuleUri、prmAddRuleNC。它们实际执行规则集合的划分。同时，判断当前规则链选项节点是否包含内容匹配选项及类型，将当前规则节点加入到对应 PORT\_GROUP 结构中不同类型的规则节点链表中。

三个函数的算法流程相似：

```
//参数：PORT_RULE_MAP *p,待构造的全局变量
//      int dport, 划分标准——目的端口
//      int sport, 划分标准——源端口
//      RULE_PTR rd, 新规则集合与原来规则集合的桥梁, OTNX

int prmAddRuleXX(PORT_RULE_MAP *p,int dport,int sport,RULE_PTR rd)
{
    if(目标端口为特定值)
    {
        设定 PORT_GROUP *prmDstPort;
        prmxAAddPortRule();           //实际执行, 添加至目的
    }
    if(源端口为特定值)
    {
        设定 PORT_GROUP *prmSrcPort;
        prmxAAddPortRule();           //实际执行
    }
    if(源和目的都为任意值)
    {
        设定 PORT_GROUP *prmGeneric;
```

```

    prmxAddPortRule();           //实际执行
}

}

```

以 **prmAddRuleNC** 函数为例，函数主要完成不包含内容匹配的规则的划分：

```

int prmAddRuleNC( PORT_RULE_MAP * p, int dport, int sport, RULE_PTR rd )
{
    /*如果目的端口为固定，则添加至目的端口子集合中*/
    if( dport != ANYPORT && dport < MAX_PORTS ) /* dst=21,25,80,110,139 */ {
        p->prmNumDstRules++;
        if(p->prmDstPort[dport] == NULL)           //该端口对应的数组的值空
        {
            //分配空间
            p->prmDstPort[dport] =
                (PORT_GROUP *)calloc(1, sizeof(PORT_GROUP));
            if(p->prmDstPort[dport] == NULL)           //分配失败
            {
                return 1;
            }
        }
        //计数器累加
        if(p->prmDstPort[dport]->pgCount==0)      p->prmNumDstGroups++;
        //进一步调用 prmxAddPortRuleXX 执行实际赋值
        //如果目的端口为特定值，则将其添加到相应的 PORT_GROUP 数组当中//去，数组
        //编号以端口号来确定。
        prmxAddPortRuleNC( p->prmDstPort[ dport ], rd );
    }

    /*如果源端口为固定，则添加至源端口子集合中*/
    if( sport != ANYPORT && sport < MAX_PORTS ) /* src=ANY, SRC=80,21,25,etc. */
    {

```

```

.....      //代码类似， 略

    prmxAddPortRuleNC( p->prmSrcPort[ sport ], rd );

}

/*如果目的端口为固定，则添加至通用子集合中*/
if( sport == ANYPORT && dport == ANYPORT ) /* dst=ANY, src=ANY */
{
    .....
```

//代码类似， 略

```

    prmxAddPortRuleNC( p->prmGeneric, rd );
}

return 0;
}

```

## 6. 4 prmxAddPortRuleXX

在函数 `prmAddRuleXX` 进行了相应的端口判断后，`prmxAddPortRuleXX` 具体执行在指定的 `PORT_GROUP` 结构中的对应规则节点链表中加入当前的 `OTNX` 结构变量。

```

static int prmxAddPortRule( PORT_GROUP *p, RULE_PTR rd )
{
    if( !p->pgHead )          //头节点为空， 添加头节点
    {
        p->pgHead = (RULE_NODE*) malloc( sizeof(RULE_NODE) );
        if( !p->pgHead ) return 1;

        p->pgHead->rnNext      = 0;
        p->pgHead->rnRuleData  = rd;           //关键一步： 在这里做实际关联
    }
}

```

```

    p->pgTail          = p->pgHead;

}

else //添加尾结点

{

    p->pgTail->rnNext = (RULE_NODE*)malloc( sizeof(RULE_NODE) );
    if(!p->pgTail->rnNext) return 1;

    p->pgTail          = p->pgTail->rnNext;
    p->pgTail->rnNext = 0;
    p->pgTail->rnRuleData = rd;

}

/*
** Set RULE_NODE ID to unique identifier
*/
p->pgTail->iRuleNodeID = p->pgCount;

/*
** Update the total Rule Node Count for this PORT_GROUP
*/
p->pgCount++;

p->pgContentCount++;

return 0;
}

```

程序执行到这一步，即遍历完整个规则链表后，规则子集合的划分已经初步完成了。所有的规则节点到目前为止都加入到以下三种类型之一的子集合中[参

PORT\_RULE\_MAP 数据结构]:

- 1) 对应于特定源端口值的 PORT\_GROUP 结构的规则链表: PORT\_GROUP \*prmSrcPort[MAX\_PORTS];
- 2) 对应于特定目的端口值的 PORT\_GROUP 结构的规则链表: PORT\_GROUP \*prmDstPort[MAX\_PORTS];
- 3) 通用规则节点链表: PORT\_GROUP \*prmGeneric;

## 6. 5 prmCompileGroups 和 BuildMultiPatternGroups

### 1、prmCompileGroups

如上分析,为了提高检测时的效率,规则链表已被按端口和协议分类来重新划分成为四类协议(TCP/UDP/ICMP/IP),每种协议又有三种类型:prmSrcPort[MAX\_PORTS]、prmDstPort[MAX\_PORTS]、\*prmGeneric。

但是,对于未来的规则检测任务而言,每个等待检测数据包都有特定的源/目的端口值(ICMP/IP等没有端口的协议的划分请参照前文所述的子集合划分标准)。也就是说,可能这个数据包既符合了某条目的端口的集合中的规则,又符合通用集合中的某条规则,所以,有必要将通用规则链接到特定端口值的这两个链表的每一个子链表中去(这样说可能有点复杂,比如,一个TCP协议,源端口为21端口的规则构成了一个子链表,我们将通用规则集中链接到这个子链表的末尾,那么每一个符合proto=tcp and sport=21的规则的数据包,在遍历完了源端口为21的子链表后,再遍历通用规则链表一次,因为该数据包有可能符合多条规则的内容)。函数 prmCompileGroups 就是用来完成这一工作的,它将通用规则链表的各个规则节点加入到另外对应于特定端口值的两个规则链表中(即源端口和目的端口链表),前提条件是目标规则链表中已经存在节点项,即在PORT\_RULE\_MAP 结构中的PORT\_GROUP 数组中每个非空元素中加入。

源代码分析如下:

```
int prmCompileGroups( PORT_RULE_MAP * p )  
{  
    PORT_GROUP *pgGen, *pgSrc, *pgDst;
```

```

RULE_PTR    *prule;
int   i;

pgGen = p->prmGeneric;           //指到通用子集合链表首节点

if(!pgGen)           //如果该通用规则链表为空，则返回
    return 0;

for(i=0;i<MAX_PORTS;i++)      //遍历数组
{
    /* Add to the Unique Src and Dst Groups as well,
    ** but don't inc thier prmNUMxxx counts, we want these to be true Unique counts
    ** we can add the Generic numbers if we want these, besides
    ** each group has it's own count.
    */
}

if(p->prmSrcPort[i])           //源端口集合的该节点项存在
{
    pgSrc = p->prmSrcPort[i];

    prule = prmGetFirstRule( pgGen );
    //以下三个循环，将通用规则节点添加到源端口集合的链表中去，
    //这里按照规则是否包含 content、包含 uricontent 和没有内容匹配选项来划分
    //目的端口类似

    while( prule )
    {
        prmxAAddPortRule( pgSrc, prule );
        prule = prmGetNextRule( pgGen );
    }
}

```

```

prule = prmGetFirstRuleUri( pgGen );
while( prule )
{
    prmxAddPortRuleUri( pgSrc, prule );
    prule = prmGetNextRuleUri( pgGen );
}

prule = prmGetFirstRuleNC( pgGen );
while( prule )
{
    prmxAddPortRuleNC( pgSrc, prule );
    prule = prmGetNextRuleNC( pgGen );
}

if(p->prmDstPort[i])           //目的端口集合的该节点项存在
{
    pgDst = p->prmDstPort[i];

    prule = prmGetFirstRule( pgGen );
    while( prule )
    {
        prmxAddPortRule( pgDst, prule );
        prule = prmGetNextRule( pgGen );
    }

    prule = prmGetFirstRuleUri( pgGen );
    while( prule )
    {
        prmxAddPortRuleUri( pgDst, prule );
    }
}

```

```

prule = prmGetNextRuleUri( pgGen );

}

prule = prmGetFirstRuleNC( pgGen );
while( prule )
{
    prmxAddPortRuleNC( pgDst, prule );
    prule = prmGetNextRuleNC( pgGen );
}
}

return 0;
}

```

## 1、BuildMultiPatternGroups

为了适应多模式搜索引擎算法的需要，fpCreateFastPacketDetection 调用 BuildMultiPatternGroups 函数(fpcreate.c)来在每个 PORT\_GROUP 结构中构建多模式搜索引擎所需的数据结构：

```

void BuildMultiPatternGroups( PORT_RULE_MAP * prm )
{
    int i;
    PORT_GROUP * pg;

    for(i=0;i<MAX_PORTS;i++)
    {
        pg = prmFindSrcRuleGroup( prm, i );
        if(pg)
        {
            BuildMultiPatGroup( pg );
            BuildMultiPatGroupsUri( pg );
        }
    }
}

```

```

pg = prmFindDstRuleGroup( prm, i );

if(pg)
{
    BuildMultiPatGroup( pg );
    BuildMultiPatGroupsUri( pg );
}

}

pg = prm->prmGeneric;

BuildMultiPatGroup( pg );
BuildMultiPatGroupsUri( pg );
}

```

## 6. 6 小结

这样，遍历完原始规则链表后，就构建了一个新的规则链表数据结构，将规则按照端口结合重新分类，为下一步多模式匹配引擎检测做好准备。

**最顶层：**按照协议不同，TCP/UDP/ICMP/IP，定义了四个最顶层的 PORT\_RULE\_MAP 类型的全局变量；

**第二层：**按照端口的目的/来源以及端口的大小不同，将规则添加至顶层结构中的某一个 PORT\_GROUP 数组中。

比如：协议为 TCP、目的为 21 的所有规则，都添加进了

**prmTcpRTNX->prmSrcPort[21]的这个链表中；**

**第三层：**将新的数据结构与原始规则链相关连，主要是通过

**(void \*) PORT\_RULE\_MAP->PORT\_GROUP->RULE\_NODE->RULE\_PRT**

在实际运行中指向了 OTNX 结构。

而 OTNX 结构

**typedef struct \_otnx{**

```
OptTreeNode *otn;  
RuleTreeNode *rtn;  
}OTNX;
```

又指向了原始规则链表中的 RTN 和 OTN。这样，就把两者关联起来了。

# 第7章 数据包处理

## 7. 1 InterfaceThread

系统在所有初始化工作完成之后，就进入了数据包处理阶段，而完成这一阶段的任务，是通过主函数在末尾调用 InterfaceThread 函数来实现的。数据包的处理，主要指的是截获、拆包、预处理插件处理、检测引擎匹配、输出插件的日志输出等任务。

InterfaceThread 关键代码分析如下：

```
/*抓包，将数据包交由回调函数 ProcessPacket 处理*/
if(pcap_loop(pd, pv.pkt_cnt, (pcap_handler)ProcessPacket, NULL) < 0)
{
    if(pv.daemon_flag)
        syslog(LOG_CONS | LOG_DAEMON, "pcap_loop: %s", pcap_geterr(pd));
    else
        ErrorMessage("pcap_loop: %s\n", pcap_geterr(pd));
    CleanExit(1);
}
```

数据包截取是通过调用 LibPcap 的库函数 pcap\_loop 来实现的，截取后的数据包，都是交由回调函数 ProcessPacket 来处理。关于 pcap\_loop 的使用，请参考附录相关章节。

## 7. 2 ProcessPacket

回调函数 Processpacket 用于实际的数据包处理。该函数是 pcap\_handler 类型，pcap.h 中关于该函数的原型定义如下：

```
typedef void (*pcap_handler)(u_char *, const struct pcap_pkthdr*,const u_char *);
** 第 1 个参数这里没有什么用;
** 第 2 个参数为 pcap_pkthdr 结构指针，记录时间戳、包长、捕捉的长度;
```

\*\* 第 3 个参数字符串指针为数据包；

源代码分析如下：

```
void ProcessPacket(char *user, struct pcap_pkthdr * pkthdr, u_char * pkt)
{
    Packet p;

    p.packet_flags = 0;           //重置每个数据包的标志
    pc.total++;                  //数据包总数累计
    packet_time_update(pkthdr->ts.tv_sec); //处理包嗅探时间

    /* reset the thresholding subsystem checks for this packet */
    sfthreshold_reset();

    SnortEventqReset();
    //如果是以 Win32 服务进程运行
#ifndef defined(WIN32) && defined(ENABLE_WIN32_SERVICE)

    if( pv.terminate_service_flag || pv.pause_service_flag )
    {
        ClearDumpBuf(); /* cleanup and return without processing */
        return;
    }

#endif /* WIN32 && ENABLE_WIN32_SERVICE */

    /* 调用解码引擎，将数据包拆包后，放在 p 当中 */
    (*grinder) (&p, pkthdr, pkt);

    /* 是否输出到屏幕 */
    if(pv.verbose_flag)
    {
        if(p.iph != NULL)      //IP 包头不为空，输出之
```

```

PrintIPPkt(stdout, p.iph->ip_proto, &p);

else if(p.ah != NULL) //IP 包头为空，但以太网包头存在，输出之
    PrintArpHeader(stdout, &p);

else if(p.eplh != NULL)           //802.1X EAPOL 头
{
    PrintEapolPkt(stdout, &p);
}

else if(p.wifih && pv.showwifimgmt_flag)      //802.11 无线局域网协议 Wi-Fi
{
    PrintWifiPkt(stdout, &p);
}

}

switch(runMode)          //运行于何种模式？

{
    case MODE_PACKET_LOG:        //包记录模式，调用输出插件
        CallLogPlugins(&p, NULL, NULL, NULL);
        break;

    case MODE_IDS:
        //如果数据包的 TTL 值比用户定义的 min_ttl 还要小（即允许用户自定义最小
        //TTL 值），则跳过这个包
        if(pv.min_ttl && p.iph != NULL && (p.iph->ip_ttl < pv.min_ttl))
        {
            DEBUG_WRAP(DebugMessage(DEBUG_DECODE,
                                      "MinTTL reached in main detection loop\n"));
            return;
        }
        /* 调用检测引擎 */
        Preprocess(&p);
        break;
}

```

```
    default:  
        break;  
    }  
    ClearDumpBuf();  
}
```

可见，该函数先后完成了五步任务：

- 1、完成了数据包的初步处理，如累计计数等；
- 2、数据包解码；（请跳转至第八章）
- 3、如果需要输出到屏幕，则调用相应 Print 函数输出；（请跳转至 7.3 节）
- 4、如果是日志记录模式，则调用相应输出插件；（请跳转至 7.4 节）
- 5、如果是 IDS 模式，则调用 Preprocess 进行检测；（请跳转至 7.5 节）

为了不影响程序的叙述，本章将不进行解码函数的分析，关于程序的解码执行，本文将在第八章进行详细分析。

## 7. 3 嗅探模式的终端输出

如果用户在运行 Snort 的时候，用户使用了 -v 参数，Snort 将运行于嗅探模式之下，嗅探模式很简单，就是将截取的数据包输出到终端设备上。与之相应的开关，ParseCmdLine 在设置了 pv.verbose\_flag = 1；另外与此类似的，还有：

```
-d : 输出应用层数据; pv.data_flag = 1;  
-e: 输出链路层数层; pv.show2hdr_flag = 1;
```

数据包的输出比较简单，根据某种协议的数据结构将截取的数据直接输出到终端即可。如果读者对这些数据协议结构不太了解的话，可以参考相关资料，推荐《TCP/IP 协议详解 第一卷》

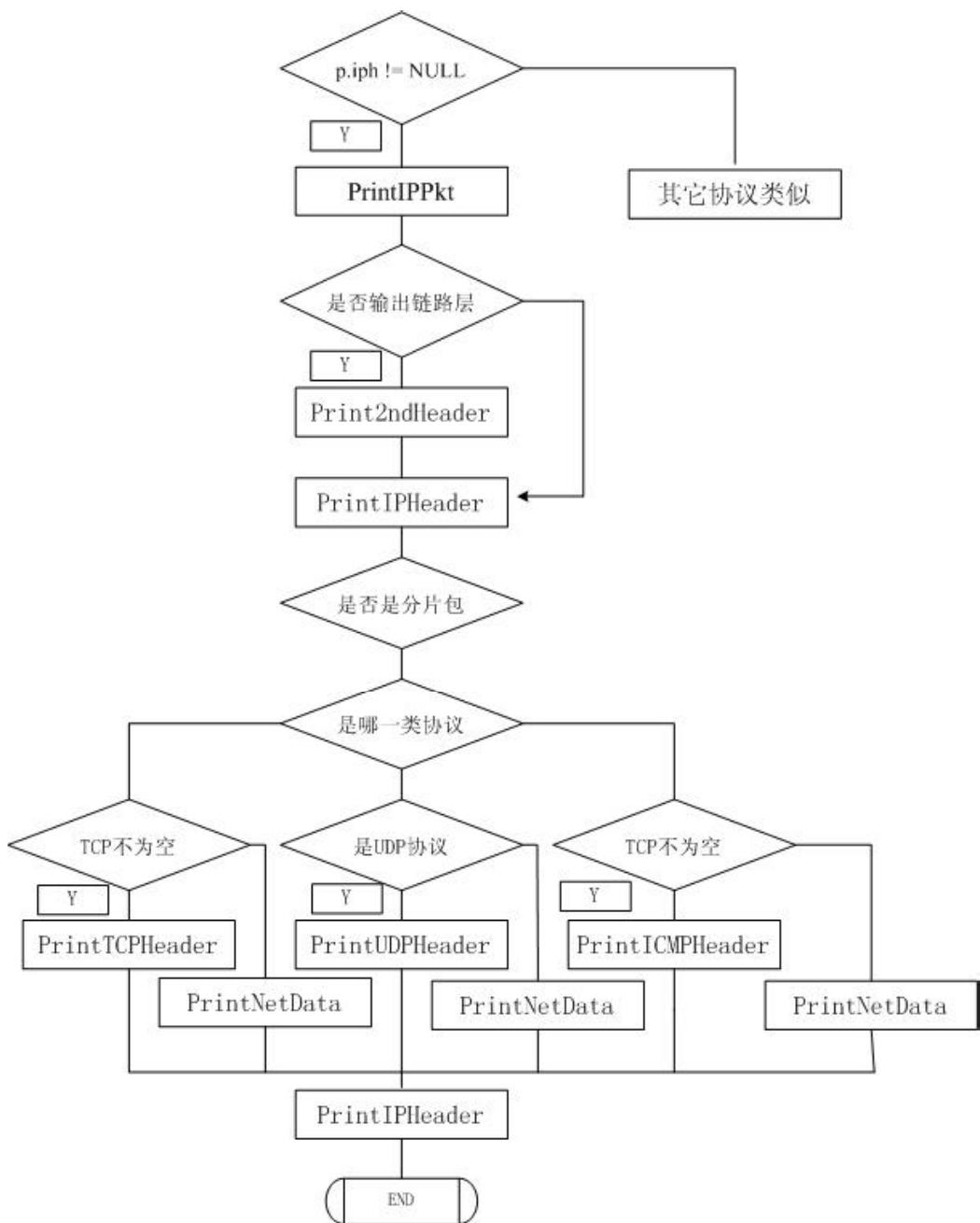
系统在数据分拆包完成后，ProcessPacket 函数调用

```
if(pv.verbose_flag) /*根据标志判断是否要输出*/  
{  
    if(p.iph != NULL) /*根据协议，调用相应的输出函数*/  
        PrintIPPkt(stdout, p.iph->ip_proto, &p);
```

```
else if(p.ah != NULL)
    PrintArpHeader(stdout, &p);
else if(p.eplh != NULL)
{
    PrintEapolPkt(stdout, &p);
}
else if(p.wifih && pv.showwifimgmt_flag)
{
    PrintWifiPkt(stdout, &p);
}
}
```

进入数据包终端输出模块。一共支持 IP、ARP、Eapol、Wifi 四种协议。以下，将以最为常用的 IP 包输出为例，分析数据终端输出流程。

IP 包头的输出流程如下图所示：



### PrintIPPKt

函数首先根据 `pv.show2hdr_flag` 标志来决定是否需要调用 `Print2ndHeader` 输出第二层的相关信息。然后再调用 `PrintIPHeader` 输出 IP 包头，最后判断高层协议是 TCP/UDP/ICMP，调用相应的输出函数处理之。

### Print2ndHeader

作用：输出数据链路层信息

函数主要算法为首先利用 `datalink` (OpenPcap 函数中赋值) 的值判断数据链路层协议 (一共支持四种协议: Ethernet、IEEE802\_11、Token Ring、Linux cooked sockets)，调用相应的输出函数进行处理。

比如，如果 `datalink` 为以太网的话，则调用相应的 `PrintEthHeader` 函数输出以太网包头。

以太网头共十四个字节: 前六个字节为来源 MAC, 后六个字节为目的 MAC, 最后两个字节为高层协议的类型, 这就不难理解以下函数了:

```
void PrintEthHeader(FILE * fp, Packet * p)
{
    /* 来源地址 */
    fprintf(fp, "%X:%X:%X:%X:%X:%X -> ", p->eh->ether_src[0],
            p->eh->ether_src[1], p->eh->ether_src[2], p->eh->ether_src[3],
            p->eh->ether_src[4], p->eh->ether_src[5]);

    /* 目的地直 */
    fprintf(fp, "%X:%X:%X:%X:%X:%X ", p->eh->ether_dst[0],
            p->eh->ether_dst[1], p->eh->ether_dst[2], p->eh->ether_dst[3],
            p->eh->ether_dst[4], p->eh->ether_dst[5]);

    /* 高层协议及包长*/
    fprintf(fp, "type:0x%X len:0x%X\n", ntohs(p->eh->ether_type), p->pkth->len);
}
```

其它所有的输出函数都与此类似，读者可以自行分析。

## 7. 4 日志记录模式的输出处理

Snort 运行模式主要有三类: 嗅探模式、日志记录模式和入侵检测模式; 第一种模式的输出我们在上一节中已经分析。Snort 的日志记录模式对应的命令行

参数 <-1 日志文件存储目录>，然后在 ParseCmdLine 中设置 pv 变量，接着根据 pv 变量的值，配置 runMode，ProcessPacket 函数根据 runMode 的值判断系统如果运行于日志记录模式的话，则调用相应的插件，进行日志记录，对应代码如下：

```
switch(runMode)          //运行于何种模式?  
{  
    case MODE_PACKET_LOG:  
        CallLogPlugins(&p, NULL, NULL, NULL);  
        break;  
    .....  
}
```

### CallLogPlugins

输出链表共有两条：LogList（日志）和 AlertLis（报警）。这里 CallLogPlugins 函数主要是遍历 LogList 的所有注册的功能函数，实现日志的输出。函数源代码分析如下：

```
void CallLogPlugins(Packet * p, char *message, void *args, Event *event)  
{  
    OutputFuncNode *idx;  
  
    idx = LogList;      /*idx 指向日志输出插件链表首节点*/  
  
    if(p != NULL)  
    {  
        if(pv.obfuscation_flag)  
            ObfuscatePacket(p);  
    }  
  
    pc.log_pkts++;
```

```

while(idx != NULL)           /*遍历链表中注册的所有功能函数*/
{
    idx->func(p, message, idx->arg, event);
    idx = idx->next;
}

return;
}

```

日志输出件有<skynet>, 这里我们将就最为常用的……来进行分析……

## 7. 5 Preprocess

Preprocess 函数处理整个数据检测及处理流程。主要处理“预处理->检测->输出”三个阶段。

源代码分析如下：

```

int Preprocess(Packet * p)
{
    PreprocessFuncNode *idx;
    int retval = 0;

    if(p->csum_flags)      //校验和是否出错?
    {
        return 0;
    }

    do_detect = 1;          //是否打开检测引擎变量标志
    idx = PreprocessList;   //指向预处理插件的顶层链表

```

```

/*重置适当的应用层协议字段*/
p->uri_count = 0;
UriBufs[0].decode_flags = 0;

/* 打开所有预处理器*/
p->preprocessors = PP_ALL;

while(idx != NULL)          //遍历预处理器插件，对数据包进行预处理
{
    assert(idx->func != NULL);

idx->func(p);

    idx = idx->next;
}

if(do_detect)           //进入检测引擎
Detect(p);

retval = SnortEventqLog(p);      //输出
SnortEventqReset();

otn_tmp = NULL;

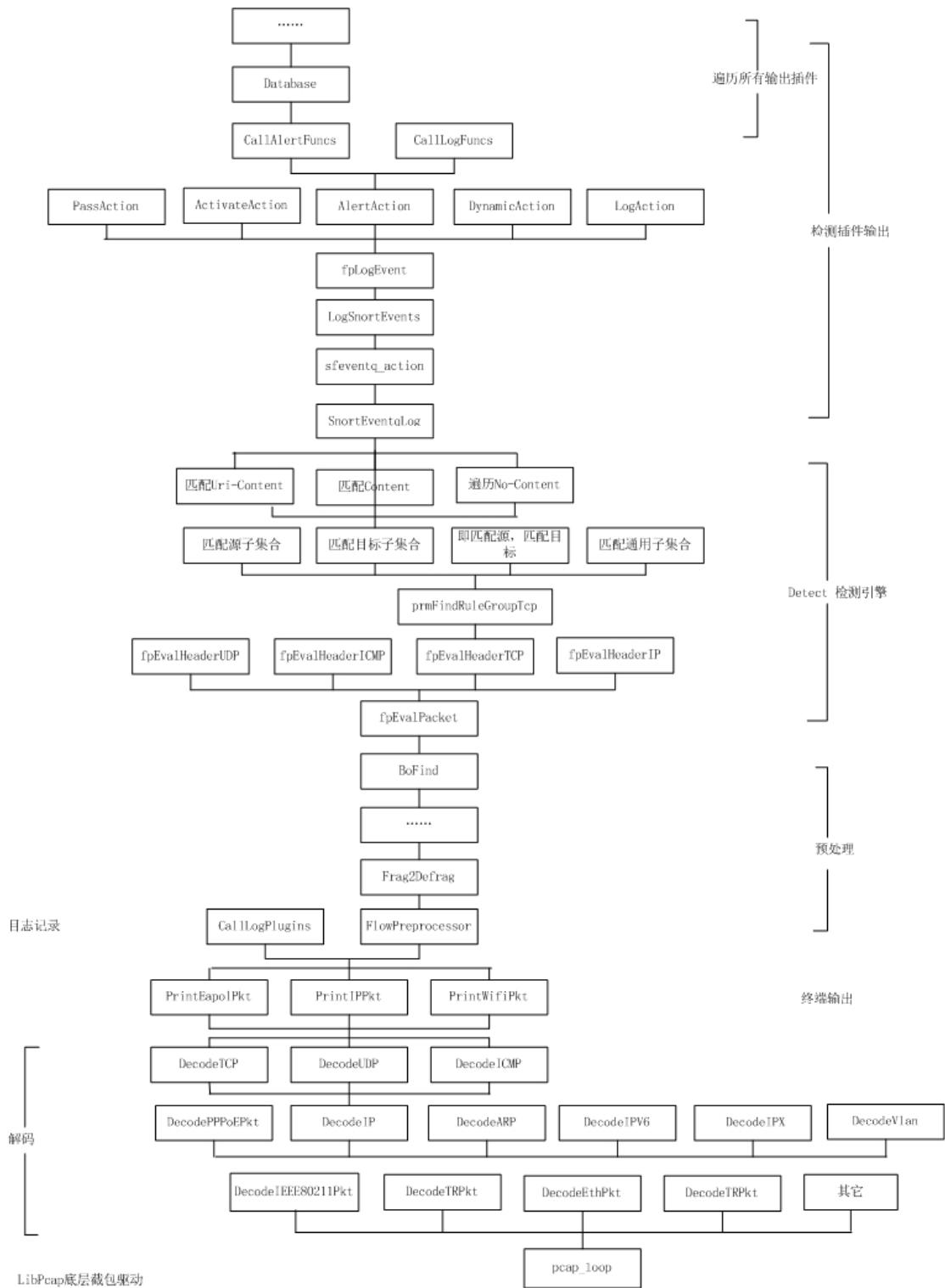
/*
** If we found events in this packet, let's flush
** the stream to make sure that we didn't miss any
** attacks before this packet.
*/
if(retval && p->ssnptr)
    AlertFlushStream(p);

```

```
/**  
 * See if we should go ahead and remove this flow from the  
 * flow_preprocessor -- cmg  
 */  
  
CheckFlowShutdown(p);  
  
return retval;  
}
```

## 7. 6 小结

函数 InterfaceThread 完成了整个数据包在 Snort 中处理的全部流程，可以用下图来表示：



图中标注了数据包在系统中所经历的重要的处理函数，有许多函数我们本章只是做了一个概念性的了解，后文将对这一流程进一步分析。

# 第8章 数据包解码引擎

我们在分析 ProcessPacket 函数时说过，系统在输出数据信息或转入检测引擎之前，需要先对数据包进行拆解。当时为了程序分析的连续性，并没有展开进行深入分析。本章将就 Snort 的数据包解码引擎进行详细论述分析。

数据包解码引擎是 Snort 的一个重要功能模块，它是数据包进一步处理的基础。要理解数据包解码引擎，有两个重要的地方：

1、系统在 OpenPcap 函数中，利用 snaplen = pcap\_snapshot(pd) 获取了数据链路层的值，然后在主函数中调用 SetPktProcessor 进行解码函数的关联处理。

SetPktProcessor 函数的主要处理流程是利用 switch case 语句，判断 snaplen 的值，然后将函数指针指向对应的实际解码函数，如：

```
switch(datalink)      /*判断数据链路层协议类型*/  
{  
    case DLT_EN10MB:    /* Ethernet */  
        .....  
        grinder = DecodeEthPkt;  
        break;  
    .....  
}
```

2、数据包解码引擎解码后的结果全部放在 Packet \*p，所以，理解结构 Packet 的含义，是理解数据包解码引擎及后续检测引擎的基础。关于 Packet，读者可以参考 1.4 相关内容，当然，要理解 Packet 结构的含义，还需要理解相关 TCP/IP 各协议的封包格式及相关知识，如果你对此不太了解，还是推荐你读一读《TCP/IP 详解》，因为经典嘛！

在 grinder 指向了实际的解码函数后，ProcessPacket 函数就调用

**(\*grinder) (&p, pkthdr, pkt)**

进入解码引擎了，相关函数定义在 decode.c 中。

解码引擎比较复杂，几乎涉及了 TCP/IP 中的所有协议，这里不可一一详细论述（也没有必要详细论述，因为各个协议模块解码的运行原理都是一样

的)，所以本文将就使用最广泛的以太网拆包为基础（拆解以太网包->拆解 IP 包->拆解 TCP/UDP/ICMP），分析数据包解码引擎的运行原理。其它协议，读者可以在此基础上，对照自行分析。

## 8. 1 DecodeEthPkt

函数主要功能是拆解以太网封包，以太网包头只有十四个字节：6 字节目的地直，6 字节源地地，2 字节上层协议值。所以 DecodeEthPkt 分拆起来就比较简单了，结构定义如下 decode.h):

```
typedef struct _EtherHdr
{
    u_int8_t ether_dst[6];
    u_int8_t ether_src[6];
    u_int16_t ether_type;
} EtherHdr;
```

函数源码分析如下：

```
// Packet * p: 存放解码结果

// struct pcap_pkthdr * pkthdr: libpcap 截包的包头，包含时间，长度等信息;

// u_int8_t * pkt: 抓回来的数据包，我们要拆包，就是拆它啦

void DecodeEthPkt(Packet * p, struct pcap_pkthdr * pkthdr, u_int8_t * pkt)
{
    //函数首先进行一些前期处理，

    p->pkth = pkthdr;           //libpcap 包头给 p
    p->pkt = pkt;               //原始数据包头给 p

    pkt_len = pkthdr->len;     /* 总包长 */
    cap_len = pkthdr->caplen;   /* 嗅探包长 */
```

```

if(snaplen < pkt_len)           //实际包头比预定义最大包头还要长
    pkt_len = cap_len;

/* 嗅探到的包小于最小以太网头大小 */
if(cap_len < ETHERNET_HEADER_LEN)
{
    .....

/*预处理完成后，就开如拆包，拆包很简单，一句话搞定： */
p->eh = (EtherHdr *) pkt;

/*在拆解了以太网封包头后，就要进行下一步拆解上层封包做准备了，函数利用一个 switch
case 语句，判断 p->eh->ether_type 的值，调用相应的拆包函数，例如如果是 IP 包的话：
*/
switch
{
    .....

    case ETHERNET_TYPE_IP:
        DecodeIP(p->pkt + ETHERNET_HEADER_LEN,
            cap_len - ETHERNET_HEADER_LEN, p);
        .....
}

```

进入 IP 包解码模块。

**p->pkt + ETHERNET\_HEADER\_LEN:** 指针移到 IP 包头开始的位置；

**cap\_len - ETHERNET\_HEADER\_LEN:** 总长度减去以太网包头 14 个字节；

## 8. 2 DecodeIP

DecodeIP 函数的功能是拆解 IP 包，其算法流程与 DecodeEthPkt 函数相似，首先对照 IP 协议的封包格式，理解这一个结构先：

```

typedef struct _IPHdr
{
    u_int8_t ip_verhl;      /* IP 版本 & 包头长度 */
    u_int8_t ip_tos;        /* 服务类型 */
    u_int16_t ip_len;       /* 数据报长度 */
    u_int16_t ip_id;        /* identification */
    u_int16_t ip_off;       /* 分片偏移*/
    u_int8_t ip_ttl;        /* 生存期 TTL */
    u_int8_t ip_proto;      /* 上层协议的类型, TCP/UDP/ICMP */
    u_int16_t ip_csum;      /*较验和 */

    struct in_addr ip_src;  /* 来源 IP 地址 */
    struct in_addr ip_dst;  /* 目的 IP 地址*/
}
IPHdr;

```

函数源码分析如下（有删节）：

```

void DecodeIP(u_int8_t * pkt, const u_int32_t len, Packet * p)
{
    p->iph = (IPHdr *) pkt;           //同样，拆包也是一句话

    if(len < IP_HEADER_LEN) //如果数据包总长度太小 (<20)，即畸形包
    .....
    if(IP_VER(p->iph) != 4)          //可以以后版本会支持 IPV6 的，毕竟 Windows
                                      //和 Linux 现在都支持了
    .....
    /*将包头长度左移两位， 得到实际长度，一般是 5<<2=20 啦*/
    ip_len = ntohs(p->iph->ip_len);

    hlen = IP_HLEN(p->iph) << 2;
    if(hlen < IP_HEADER_LEN)          //小于了 20， 畸形包
    {
        /*是 IDS 运行模式的话， 需要输出一个事件： 收到一个畸形包*/
        .....

```

```

}

.....
if (ip_len != len)
{
    /*这里处理如果 IP 包头中获取长度值与实际的不符的情况，如果是大于，就让他
    强行等于 ip_len = len，否则就只好报错了*/
    .....
}

if(ip_len < hlen)          //IP dgm len < IP hdr
.....
if(pv.checksums_mode & DO_IP_CHECKSUMS)      //如果需要进行较验和检验
{
    csum = in_chksum_ip((u_short *)p->iph, hlen);
}
.....
/* 接下来都是按照 IP 包头的封包格式要求，对包的各个部份进行合法性检测读者可以
参考 IP 封包自行分析之*/
.....
p->frag_offset = ntohs(p->iph->ip_off);      /*检测是否是分片包*/
.....
/*设置相关标志位
rf: IP 保留位， dr: 不是分片包标志； mr: 后续分片标志*/
p->rf = (u_int8_t)((p->frag_offset & 0x8000) >> 15);
p->df = (u_int8_t)((p->frag_offset & 0x4000) >> 14);
p->mf = (u_int8_t)((p->frag_offset & 0x2000) >> 13);
.....
if(!(p->frag_flag))      //如果不是分片包，继续分析上层协议
{

```

```

switch(p->iph->ip_proto)
{
    case IPPROTO_TCP:           //是 TCP
        pc.tcp++;
        DecodeTCP(pkt + hlen, ip_len, p); //继续拆分 TCP 包
        ClearDumpBuf();
        return;

    case IPPROTO_UDP:
        pc.udp++;
        DecodeUDP(pkt + hlen, ip_len, p); //继续拆分 UDP 包
        ClearDumpBuf();
        return;

    case IPPROTO_ICMP:
        pc.icmp++;
        DecodeICMP(pkt + hlen, ip_len, p); //继续拆分 ICMP 包
        ClearDumpBuf();
        return;

    default:
        pc.other++;
        p->data = pkt + hlen;
        p->dsize = (u_short) ip_len;
        ClearDumpBuf();
        return;
}

else
{

```

```

/* set the payload pointer and payload size */

p->data = pkt + hlen;

p->dsize = (u_short) ip_len;

}

}

```

IP 包的解码需要注意的一点就是关于 IP 分片包的处理：

```

p->frag_offset = ntohs(p->iph->ip_off);      /*检测是否是分片包*/

/*设置相关标志位

rf: IP 保留位, dr: 不是分片包标志; mr: 后续分片标志*/
p->rf = (u_int8_t)((p->frag_offset & 0x8000) >> 15);
p->df = (u_int8_t)((p->frag_offset & 0x4000) >> 14);
p->mf = (u_int8_t)((p->frag_offset & 0x2000) >> 13);

if(!(p->frag_flag))    /*如果不是分片包, 继续分析上层协议*/
{
    .....
}

else          /*是分片包, 则只记录, 不继续拆包了*/
{
    p->data = pkt + hlen;
    p->dsize = (u_short) ip_len;
}

```

可以看到，如果是一个分片包，则在进行了标志位计算、设置，计数器的设置后，根据 p->frag\_flag 分片标志位，不再继续进行上层协议的拆包。

那分片的数据包是怎么检测的呢？Snort 专门有一个用于分片重组的预处理插件 frag2，其具体实现我们会在下一章详细分析。这里只提一点，就是分片包进入 frag2 进行重组，重组完成后，在其 RebuildFrag 函数中会调用：

```
ProcessPacket(NULL, defrag_pkt->pkth, defrag_pkt->pkt);
```

重新进入数据包处理模块，再次进入解码引擎，其流程就像处理未分片的数据包一样了。

## 8. 3 DecodeTCP

同样地，还是先来看看 TCP 包头结构的定义：

```
typedef struct _TCPHdr
{
    u_int16_t th_sport;      /* source port */
    u_int16_t th_dport;      /* destination port */
    u_int32_t th_seq;        /* sequence number */
    u_int32_t th_ack;        /* acknowledgement number */
    u_int8_t th_offx2;       /* offset and reserved */
    u_int8_t th_flags;
    u_int16_t th_win;        /* window */
    u_int16_t th_sum;        /* checksum */
    u_int16_t th_urp;        /* urgent pointer */

}           TCPHdr;
```

函数源码分析如下：

```
void DecodeTCP(u_int8_t * pkt, const u_int32_t len, Packet * p)
{
    struct pseudoheader      /* 要计算 TCP 的校验和，需要定义一个伪头 */
    {
        u_int32_t sip, dip;    /* IP addr */
        u_int8_t zero;         /* checksum placeholder */
        u_int8_t protocol;     /* protocol number */
        u_int16_t tcplen;      /* tcp packet length */
    };
}
```

```

u_int32_t hlen;           /* TCP header length */

u_short csum;             /* checksum */

struct pseudoheader ph;   /* pseudo header declaration */

if(len < 20)              //数据包长度检测，至少应该有 20 个字节,
{
    //输入错误信息，添加 IDS 事件等
}

/* 拆包 */
p->tcpiph = (TCPHdr *) pkt;
if(hlen < 20)              //包头长度检验
{
    //输出错误信息，添加 IDS 事件等
}

if(hlen > len)            /*包头长度比包长度还要大*/
{
    //输出错误信息，添加 IDS 事件等
}

if(pv.checksums_mode & DO_TCP_CHECKSUMS) /*如果要进行校验和检测*/
{
    .....
    csum = in_chksum_tcp((u_int16_t *)&ph, (u_int16_t *)(p->tcpiph), len);
    .....
}

//关于 TCP 等数据包头校验和的计算，请参考附录相关内容
}

p->sp = ntohs(p->tcpiph->th_sport);      //源端口
p->dp = ntohs(p->tcpiph->th_dport);        //目的端口

```

```

p->tcp_options_len = hlen - 20;           //计算选项字段: 总长度减去默认头长度
if(p->tcp_options_len > 0)
{
    p->tcp_options_data = pkt + 20;        //选项字段存在, decode it
    DecodeTCPOptions((u_int8_t *) (pkt + 20), p->tcp_options_len, p);
}
else
{
    p->tcp_option_count = 0;
}

/* set the data pointer and size */
p->data = (u_int8_t *) (pkt + hlen);      //指针位移至 TCP 包头之后, 即数据区

if(hlen < len)
{
    p->dsize = (u_short)(len - hlen);
}
else
{
    p->dsize = 0;
}
}

```

相对于 TCP 协议来说, UDP 和 ICMP 协议是非常简单的, 拆解起来也非常地容易, 由于其结构流程与 TCP 几乎是完全相同的, 这里就不一一赘述了。

# 第9章 预处理插件的工作

在前面的章节中，我们对预处理插件的注册、初始化、调用都做了一一详细地讲解。但是预处理插件的魅力，还是在它的具体实现上面。由于每一类预处理插件都是一个庞大的系统，算法难易程度也不尽相同，笔者在本章中将挑选几类典型而重要的插件，向读者一一详述。包括了：

- Stream4：状态维护和会话重组
- Frag2：分片重组和攻击检测
- Postscan：端口扫描
- BO：bo 后门
- Arpspoof：反 ARP 欺骗

## 9. 1 Stream4 TCP 状态维护和会话重组

## 9. 2 frag2 分片重组和攻击检测

我们知道，每种类型的网络硬件具有不同的最大传输单元（MTU）。MTU 表明了在此链路上，最大承载的分组大小。比如：IP 包最大为 0xFFFF(65536 字节)，而以太网的 MTU 为 1500 字节，则每一个大于 1500 字节的 IP 数据包都将会被分片。然后到对端再重组。

因为每一个 IP 分片包都具备完整的 IP 包头，这样，攻击者就有可能构造特殊的分片包攻击 IDS 或者是逃过 IDS 的检测。简单地举个例，IDS 认为 ‘ABC’ 是一个攻击特征码，攻击者将攻击包分成三个包，分别包含 ’A’、’B’、’C’。每个单独的分片都不会匹配特征。而对端主机会重组这些分片包，入侵者就达到了攻击的目的。

Snort 引入了预处理插件 frag2，它能够对分片包进行重组来定位这类攻击。它的工作原理是将所有的分片重组构造成一个包含完整信息的包，再将这个包传给检测引擎。

另外，frag2 还可以检测基于分片的某些 DoS 攻击。如 teardrop 。

在 snort.conf 中可以使用以下参数打开 frag2:

preprocessor frag2: [参数 1], [参数 2], .....

frag2 预处理器模块使用 Frag2Data 结构来存储当前模块的参数配置情况, 这个结构已经在 frag2 的初始化函数中设置了相应的值。结构定义如下:

```
typedef struct _Frag2Data
{
    u_int8_t os_flags;
    u_int32_t memcap;           /*内存限额*/
    u_int32_t frag_timeout;     /*超时限制*/
    u_int32_t last_prune_time;
    u_int8_t stop_traverse;

    u_int8_t min_ttl; /* 可接受的最小 TTL 值 */
    u_int8_t ttl_limit; /* 进行逃避攻击检测时可接受的最大 TTL 值 */
    char frag2_alerts; /* frag2 的报警是否打开 */

    u_int32_t sp_threshold;
    u_int32_t sp_period;

    u_int32_t suspend_threshold;
    u_int32_t suspend_period;

    char state_protection;

    SPMemControl frag_sp_data; /* 内存使用控制 */
}
```

## 9. 2. 1 重要的数据结构

假如陆续有 100 个分片包进入 Snort, 这些包在分片重组尚未完成之前, 存

在两种可能：

- 1、这 100 个分片同时可能属于多个数据包（也许它们全是一个包的分片，也许分别属于 100 个分片包）；
- 2、而反过来，对于每个数据包来说，又有多个分片；

这样，我们存储分片包的数据结构至少需要二维的，第一维用来存储数据包的特征，如地址、端口、协议、ID 等，第二维用来存储该数据包对应的所有分片，比如我们可以采用一条二维链表。不过，Frag2 采用的数据结构是一个两级的二叉树：主二叉树和二级二叉树。主二叉树用来存储数据包信息。二级二叉树用来存储每一个数据包（对应主二叉树的某一个节点）的所有分片信息。

对于主二叉树而言，其每个节点都指向一个 FragTracker 结构，该结构定义如下：

```
typedef struct _FragTracker
{
    ubi_trNode Node;          /*二叉树节点*/

    u_int32_t sip;            /* 源 IP 地址 */
    u_int32_t dip;            /* 目的 IP 地址 */
    u_int16_t id;             /* IP 包的 ID 标志 */
    u_int8_t protocol;        /* IP 协议类型 */

    u_int8_t ttl;              /* 用于检测逃避攻击的 TTL */
    u_int8_t alerted;         /* 是否报警的开关 */

    u_int32_t frag_flags;      /* 分片标志位 */
    u_int32_t last_frag_time; /* 最后一个分片出现的时间 */

    u_int32_t frag_bytes;      /* 分片大小 */
    u_int32_t calculated_size; /* 计算出的重组后数据包的大小 */
    u_int32_t frag_pkts;       /* 分片数目 */
```

```

ubi_trRoot fraglist;           /*二级二叉树根节点*/
ubi_trRootPtr fraglistPtr;     /*指向二级二叉树根节点的指针*/
} FragTracker;

```

主二叉树的每一个节点对应结构 ubi\_trNode，结构定义如下：

```

#define ubi_trNode    ubi_btNode
typedef struct ubi_btNodeStruct
{
    struct ubi_btNodeStruct *Link[ 3 ];    /*指向左节点、右节点、父节点*/
    char                  gender; /*表明这是一个左节点还是右节点*/
    char                  balance;
} ubi_btNode;

```

主二叉树通过结构中的 fraglist 和 fraglistPtr 字段来管理二级二叉树。其结构定义如下：

```

typedef struct {
    ubi_btNodePtr  root;      /* A pointer to the root node of the tree */
    ubi_btCompFunc cmp;      /* A pointer to the tree's comparison function */
    unsigned long   count;    /* A count of the number of nodes in the tree */
    char           flags;    /* Overwrite Y|N, Duplicate keys Y|N... */
} ubi_btRoot;

```

在实际中，二级二叉树中的每个节点又都指向一个 Frag2Frag 结构，该结构定义如下所示：

```

typedef struct _Frag2Frag
{
    ubi_trNode Node;        /*二级二叉树节点*/

    u_int8_t data;          /*分片内容*/
    u_int16_t size;         /*分片大小*/
    u_int16_t offset;       /*分片在原数据包的偏移量*/

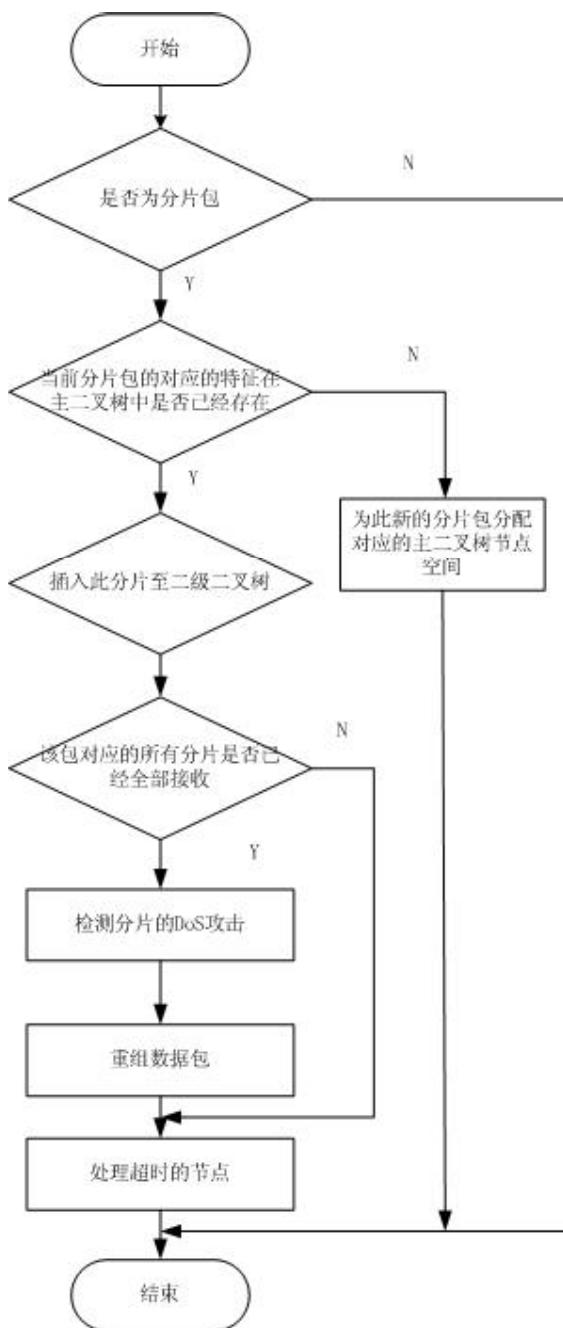
```

```
} Frag2Frag;
```

这样，采用两级二叉树，frag2 预处理模块就可以在内存中跟踪和保存完整的分片信息。具体地说，主二叉树 FragTracker 结构实现了跟踪特定 IP 数据包的功能，而对应二级二叉树的 Frag2Frag 结构则负责保存对应于该 IP 数据包的所有分片数据。我们根据这样的数据结构，就可以假设：Frag2 对于新来的分片数据包，判断它对应的特征是否匹配主二叉树的节点，匹配的话，即它是这个节点对应的后续分片包，将它加入二级二叉树，如果不匹配，则为一个新的分片，就为它分配一个对应的主二叉树节点，再把这个分片的信息，如偏移量等，加入这个新节点的第一个分片。而数据包的重组，则判断二级二叉树是否已完整地收集了该数据包的所有分片，如果收集完了，则把这些级点的数据重组即可。那 Frag2 是不是按这样的思路来处理分片包的呢？请看下文分解。

## 9. 2. 2 frag2 算法分析

frag2 的主功能模块主要通过 Frag2Defrag 函数实现，Frag2Defrag 的主要流程算法，可以通过以下流程图来描述：



函数源代码分析如下（有删节）：

```

void Frag2Defrag(Packet *p)
{
    .....

    if(!(p->preprocessors & PP_FRAG2)) /*如果标识字段指示无须进行分片重组预处理操

```

```
作，则立即返回*/
```

```
.....
```

```
/* 如果这是一个空包、或非 IP 包，或非分片的 IP 包，则退出*/
```

```
if(p == NULL || p->iph == NULL || !p->frag_flag)  
{  
    return;  
}
```

```
if(p->csum_flags & CSE_IP)      /* 检验和*/
```

```
{  
    .....  
    return;  
}
```

```
/*因为 Snort 会嵌套调用本函数，设置 PKT_REBUILT_FRAG 标志是为了防止重复处理重组的数据包*/
```

```
if(p->packet_flags & PKT_REBUILT_FRAG)  
{  
    return;  
}
```

```
/* IP 包头选项存在*/
```

```
if(p->ip_options_len)  
{  
    .....  
}
```

```
/* 检验 TTL，如果比预设的最小值还要小，就退出*/
```

```
if(p->iph->ip_ttl < f2data.min_ttl)  
.....
```

```

/*在主二叉树中进行查询，是否存在满足当前数据包所指示 IP 源/目的地址条件的二叉
树节点*/
ft = GetFragTracker(p);

if(ft == NULL)           /* 没有找到，则表明是一个新的分片数据包*/
{
    .....
    /* 为此新的数据包构造一个新节点，为了避免对 IDS 的分片攻击，构造包的限制
条件是当前出现的新的 IP 分片包是第一个分片 */
    if((f2_emergency.status == OPS_NORMAL) || (p->mf && p->frag_offset == 0))
    {
        ft = NewFragTracker(p);
        .....
    }

    /* 以下处理 ft 不为空的情况 */
    if(ft->alerted == 1)
    {
        /* 已经报过警了，退出 */
        return;
    }
    .....
    if(InsertFrag(p, ft) == -1)           /*如果这个分片包的特征已在存在二叉树的节点中，则
将其插入到二级的二叉树*/
    {
        .....          /*插入失败，退出*/
    }
    else
    {
        CompletionData compdata;

```

```

/*初始化一些标志位*/

compdata.complete = 0;
compdata.teardrop = 0;
compdata.outoforder = 0;

/* 检测一级二叉树节点中对应的二级二叉树中包含的所有分片是否接* 收完整,
以便可以进行重组操作。 */

if(FragIsComplete(ft, &compdata))

{
    /*如果已经检测到是 teardrop 攻击, 则报报警, 检测是在 FragIsComplete 中实
现的*/
    if(compdata.teardrop)

    {
        SnortEventqAdd(GENERATOR_SPP_FRAG2, FRAG2_TEARDROP,
        1, 0, 5, FRAG2_TEARDROP_STR, 0);

        ft->alerted = 1;      /*置已经报过警了的标志位*/
        DisableDetect(p);
        return;
    }

    else if(f2data.frag2_alerts &&
            compdata.outoforder &&
            !(p->packet_flags & PKT_FRAG_ALERTED))

    {
        SnortEventqAdd(GENERATOR_SPP_FRAG2, FRAG2_OUTOFORDER,
        1, 0, 5, FRAG2_OUTOFORDER_STR, 0);

        DisableDetect(p);
        p->packet_flags |= PKT_FRAG_ALERTED;
        return;
    }
}

```

```

    }

RebuildFrag(ft, p);           /* 重组分片 */
} else {
    DEBUG_WRAP(DebugMessage(DEBUG_FRAG2, "Fragment not complete\n););
}
}

/* 二叉树节点超时，删除该节点，时限是用户在 Snort.conf 中自定义的 */
if( (u_int32_t)(p->pkth->ts.tv_sec) >
    (f2data.last_prune_time + f2data.frag_timeout))
{
    PruneFragCache(ft, p->pkth->ts.tv_sec, 0);/*删除超时的主二叉树节点*/
    f2data.last_prune_time = p->pkth->ts.tv_sec;/*更新记录的时间*/
}

return;
}

```

如以上分析，Frag2 的主要功能执行函数 Frag2Defrag 的实现思路就很清晰了。不过，要了解它，还要了解它的几个重要的函数：

**GetFragTracker:** 检查该分片包的特征是否已经在主二叉树中存在

**NewFragTracker:** 分配一个新的主二叉树节点

**Insert2Flag:** 插入分片包至二级二叉树

**FragIsComplete:** 某一个数据包的所有分片是否已经接收完成

**RebuildFrag:** 重组数据包

## 9. 2. 3 GetFragTracker

GetFragTraker 用于检测当前等检测的分片包的特征（如 IP 地址、上层协议、

ID 值等) 在主二叉树中是否已经存在(即该包是一个新的分片包呢还是一个后续的分片包)。存在的话，返回一个 Frag2Tracker 结构变量。

函数的流程很简单，主要是通过调用 ubi\_sptFind 函数来实现的判断，函数层层调用，具体遍历主二叉树是在函数 qFind 中实现的，源码如下：

```
static ubi_btNodePtr qFind( ubi_btCompFunc cmp,
                            ubi_btItemPtr FindMe,
                            register ubi_btNodePtr p )
{
    int tmp;

    while( (NULL != p)
          && ((tmp = ubi_trAbNormal( (*cmp)(FindMe, p) )) != ubi_trEQUAL) )
        p = p->Link[tmp];

    return( p );
}
```

函数通过一个 While 循环来遍历二叉树。而 p->link 又实际指向了节点的左节点、右节点和父节点。所以 tmp 的值就很关键，如何决定 tmp 的值呢（左、右、父？）？

首先 需要理解的是函数指针(\*cmp)实际指向的是实际执行节点比较的函数 Frag2CompareFunc，该函数判断节点是否匹配，返回相应的值，而 ubi\_trAbNormal 是一个定义很巧妙的宏 (ubi\_BinTree.h):

```
#define ubi_trAbNormal(W) ((char)( ((char)ubi_btSgn( (long)(W) )) \
                           + ubi_trEQUAL ))
```

将判断后返回的值交由 ubi\_btSgn 函数来处理，而 ubi\_btSgn 函数定义如下：

```
long ubi_btSgn( register long x )
{
```

```
    return( (x)?((x>0)?(1):(-1)):(0) );  
}
```

这样，就一目了然了，tmp 根据宏运算后的值，决定了主二叉树节点匹配时的走向。

而单个节点的匹配函数 Frag2CompareFunc 实现也很简单，源码分析如下：

```
static int Frag2CompareFunc(ubi_trItemPtr ItemPtr, ubi_trNodePtr NodePtr)  
{  
    FragTracker *nFt;  
    FragTracker *iFt;  
  
    nFt = (FragTracker *) NodePtr;           /*树中待匹配的节点*/  
    iFt = (FragTracker *) ItemPtr;           /*当前等匹配的数据包*/  
  
    .....  
  
    /*如果不匹配，则根据大于或小于关系返回 1 或者是-1*/  
    if(nFt->sip < iFt->sip) return 1;  
    if(nFt->sip > iFt->sip) return -1;  
    if(nFt->dip < iFt->dip) return 1;  
    if(nFt->dip > iFt->dip) return -1;  
    if(nFt->id < iFt->id) return 1;  
    if(nFt->id > iFt->id) return -1;  
    if(nFt->protocol < iFt->protocol) return 1;  
    if(nFt->protocol > iFt->protocol) return -1;  
  
    return 0;      /*如果是返回 0，则表明查找到了*/  
}
```

## 9. 2. 4 NewFragTracker 和 InsertFrag

如上文所述，如果查找函数没有在主二叉树中找到与当前分片包匹配的节点，则表明这是一个新的分片包，则需要调用 NewFragTracker 函数来为之在主二叉树中分配新的空间。

NewFragTracker 首先调用 Frag2Alloc 来为主二叉树分配一个节点空间，然后将当前包的地址、协议、ID 等信息赋给 Frag2Tracker 结构的相应变量。接着调用 ubi\_trInitTree 函数初始化 ubi\_btRoot 结构，比较重要的是这里关联了节点比较的函数 RootPtr->cmp = CompFunc；

这样，节点建立好了过后，就要把当前分片包的分片信息写入已分配了的节点对应的二级二叉树中去。完成这一功能的是函数 InsertFrag。

InsertFrag 用于将分片信息（即结构 Frag2Frag 写入一个对应的主二叉树节点），源码分析如下：

```
int InsertFrag(Packet *p, FragTracker *ft)
{
    Frag2Frag *returned;
    Frag2Frag *newfrag;
    F2SPControl sp;

    sfPerf.sfBase.iFragInserts++;

    if(ft->frag_bytes + p->dsize > 70000) /* 如果重组后的数据包超过最大值*/
    {
        .....
    }

    /* 如果这是第一个分片包，设置其标志位*/
    if(p->mf && p->frag_offset == 0)
    {
```

```

if(ft->frag_flags & FRAG_GOT_FIRST)
{
    if(f2data.frag2_alerts)
    {
        SnortEventqAdd(GENERATOR_SPP_FRAG2, FRAG2_DUPFIRST,
                       1, 0, 5, FRAG2_DUPFIRST_STR, 0);

        p->packet_flags |= PKT_FRAG_ALERTED;
    }
    return -1;
}

else
{
    ft->frag_flags |= FRAG_GOT_FIRST;
}

}

else if(!p->mf && p->frag_offset > 0)
{
    ft->frag_flags |= FRAG_GOT_LAST;
    ft->calculated_size = (p->frag_offset << 3) + p->dsize;      /*计算出重组后的数
据包的大小*/
    .....
}
}

if(!(ft->frag_flags & FRAG_GOT_FIRST))
{
    DEBUG_WRAP(DebugMessage(DEBUG_FRAG2, "setting out of order!"));
    ft->frag_flags |= FRAG_OUTOFORDER;
}

```

```

ft->last_frag_time = p->pkth->ts.tv_sec;

ft->frag_pkts++;

ft->frag_bytes += p->dsize;

DEBUG_WRAP(DebugMessage(DEBUG_FRAG2, "ft->frag_bytes: %u %u\n",
ft->frag_bytes, f2data.frag_sp_data.mem_usage));
```

```

sp.ft = ft;

sp.cur_time = p->pkth->ts.tv_sec;

f2data.frag_sp_data.control = &sp;
```

/\*为该分片包分配空间\*/

```

newfrag = (Frag2Frag *) SPAlloc(sizeof(Frag2Frag), &(f2data.frag_sp_data));
```

```

newfrag->data = (u_int8_t *) SPAlloc(p->dsize, &(f2data.frag_sp_data)); /*分片的内容*/
```

```

memcpy(newfrag->data, p->data, p->dsize);
```

```

newfrag->offset = p->frag_offset << 3; /*分片在原数据包中的偏移位*/
newfrag->size = p->dsize;
```

```

returned = (Frag2Frag *) ubi_sptFind(ft->fraglistPtr, /*在二级二叉树中查找当前分片是否已经存在*/
(ubi_btItemPtr)newfrag);
```

```

if(returned != NULL) /*如果该分片已经在对应的二级二叉树中存在*/
{
    .....
    return 0;
}
```

```

/*调用 ubi_sptInsert 将该分片包插入二级二叉树中*/

if(ubi_sptInsert(ft->fraglistPtr, (ubi_btNodePtr)newfrag,
                  (ubi_btNodePtr)newfrag, NULL) == ubi_trFALSE)

{

.....



}

return 0;
}

```

可以，节点插入的实现是由 `ubi_sptInsert` 来完成的，而 `ubi_sptInsert` 又是通过调用 `ubi_btInsert` 函数来实际执行：

```

ubi_trBool ubi_btInsert( ubi_btRootPtr RootPtr,
                         ubi_btNodePtr NewNode,
                         ubi_btItemPtr ItemPtr,
                         ubi_btNodePtr *OldNode )

{
.....

```

```

/* 找到应该插入节点的父节点的位置 parent, tmp 决定了是应该插入至左节点还是右节点
*/

```

```
*OldNode = TreeFind(ItemPtr, (RootPtr->root), &parent, &tmp, (RootPtr->cmp));
```

```

/* 插入新的节点进二级二叉树 */

if( NULL == (*OldNode) )

{
    if( NULL == parent )

        RootPtr->root = NewNode; /*作为根节点插入*/

    else

    {

```

```

parent->Link[(int)tmp]      = NewNode; /*插入新节点*/
NewNode->Link[ubi_trPARENT] = parent; /*回指父节点*/
NewNode->gender           = tmp;
}

(RootPtr->count)++;

return( ubi_trTRUE );
}

.....
}

```

可见，插入节点最重要的是遍历二叉树，定位插入位置，即 TreeFind 的实现。TreeFind 遍历二叉树，其原理与我们上一节讲到查找时遍历二叉树的方法是一样的，在函数 TreeFind 中，使用：

```

while( (NULL != tmp_p)
    && (ubi_trEQUAL != (tmp_cmp = ubi_trAbNormal((*CmpFunc)(findme, tmp_p)))) )

```

来实现了二叉树的遍历与比较。

## 9. 2. 5 FragIsComplete

函数 FragIsComplete 用于判断主二叉树节点所对应的二级二叉树中包含的所有分片数据是否收集完整，以便可以进行下一步的重组操作。源代码如下：

```

int FragIsComplete(FragTracker *ft, CompletionData *compdata)
{
    compdata->complete = 1;

    sfPerf.sfBase.iFragCompletes++;

    if(ft->frag_flags & FRAG_REBUILT) /*数据包是否已经被重建了*/
    {
        .....
    }
}

```

```

    }

    if((ft->frag_flags & (FRAG_GOT_FIRST|FRAG_GOT_LAST)) ==
       (FRAG_GOT_FIRST|FRAG_GOT_LAST))

    {

        /* global data for CompletionTraverse */

        next_offset = 0;

        /* 遍历该数据包对应的主二叉树下的二级二叉树的所有节点*/

        (void)ubi_trTraverse(ft->fraglistPtr, CompletionTraverse, compdata);

        return compdata->complete;

    }

    else

    {

        return 0;

    }

}

```

可见，二级二叉树的遍历，主要是通过 ubi\_trTraverse 来实现的。函数源码如下：

```

unsigned long ubi_btTraverse( ubi_btRootPtr    RootPtr,
                               ubi_btActionRtn EachNode,
                               void            *UserData )

{

    ubi_btNodePtr p = ubi_btFirst( RootPtr->root );           /*指向首节点*/

    unsigned long count = 0;

    /*从二级二叉树的首节点开始，遍历整棵树*/

    while( NULL != p )

    {

        (*EachNode)( p, UserData );      /*检测每个分片包的偏移量*/

        count++;

    }

```

```

if(RootPtr->count != 0)

    p = ubi_btNext( p );

else

    return count;

}

return( count );
}

```

要理解这个函数，有个重要的地方：(\*EachNode)( p, UserData )。该指向函数的指针调用了函数 CompletionTraverse 来检验每个分片包的偏移量，以检测该数据包对应的二级二叉树所包含的所有分片是否完整。

CompletionTraverse 函数源代码分析如下：

```

static void CompletionTraverse(ubi_trNodePtr NodePtr, void *complete)

{

Frag2Frag *frag;

CompletionData *comp = (CompletionData *) complete;

frag = (Frag2Frag *) NodePtr;

/*全局变量 next_offset 表示分片重组中的当前偏移地址值*/

if(frag->offset == next_offset)

{

/*当前分片偏移量正常，更新全局变量*/

next_offset = frag->offset + frag->size;

}

/*当前分片与前面的分片偏移量重叠，表明发生了 TearDrop*/

else if(frag->offset < next_offset)

{

/* flag a teardrop attack detection */

comp->teardrop = 1;
}

```

```

if(frag->size + frag->offset > next_offset)
{
    next_offset = frag->offset + frag->size;
}
}

/*当前分片与前一个分片之间存在“空白”，表明分片数据并不完整*/
else if(frag->offset > next_offset)

{
    DEBUG_WRAP(DebugMessage(DEBUG_FRAG2, "Holes in completion check... (%u
> %u)\n",
                           frag->offset, next_offset););
    comp->complete = 0; /*置标志位，表明分片还不完全，还不能重组*/
}
return;
}

```

## 9. 2. 5 RebuildFrag

函数 RebuildFrag 是 frag2 预处理引擎的最后一个重要的函数，它实现了数据包的重组。当一个分片数据包进入引擎后，当引擎通过调用 FragIsComPlete 函数发现所有分片数据包已收集完毕，则调用 RebuildFrag 进行数据包的重组。

<skynet>

## 9. 3 BO 后门

## 9. 4 ARP 欺骗检测

ARP 欺骗攻击的原理，这里就不再赘述了，读者可以查阅其它相关资料。Snort 检测 ARP 欺骗的算法也很简单：在 Snort.conf 中建立一个 MAC-IP 的对照表，

收集所有的 ARP 请求包或应答包，先判断这些包的合法性，然后把实际的包中的地址信息与对照表中的地址信息作比较，如果不一致，则表明发生了 ARP 攻击。

SetupARPspooft 函数中，调用了两个初始化函数：

- ARPspooftInit
- ARPspooftHostInit

因为用户使用该插件，需要先在 snort.conf 中定义一个 MAC-IP 表，所以，插件在检测前，需要先将其读取出来：

```
void ARPspooftHostInit(u_char *args)
{
    /*为节点分配空间*/
    ipmel = (IPMacEntryList *)SnortAlloc(sizeof(IPMacEntryList));

    /* 拆分参数*/
    ParseARPspooftHostArgs(args);

    check_overwrite = 1;

    return;
}
```

ParseARPspooftHostArgs 将定义的参数读取出来，填充 ipmel 链表：

```
void ParseARPspooftHostArgs(char *args)
{
    char **toks;
    char **macbytes;
    int num_toks, num_macbytes;
    int i;
    struct in_addr IP_struct;
    IPMacEntry *ipme = NULL;
```

```

if (ipmel == NULL)           /*链表首节点为空*/
{
    FatalError("%s(%d) => Please activate arpspoof before trying to "
               "use arpspoof_detect_host\n", file_name, file_line);
}

/*拆分字符串*/
toks = mSplit(args, " ", 2, &num_toks, '\\');

if (num_toks != 2)
{
    FatalError("Arpspoof %s(%d) => Invalid arguments to "
               "arpspoof_detect_host\n", file_name, file_line);
}

/* 为临时节点分配空间 */
ipme = (IPMacEntry *)SnortAlloc(sizeof(IPMacEntry));

/*提取出 IP 地址，顺便判断一下地址的合法性*/
if ((IP_struct.s_addr = inet_addr(toks[0])) == -1)
{
    FatalError("Arpspoof %s(%d) => Invalid IP address as first argument of "
               "IP/MAC pair to arpspoof_detect_host\n", file_name, file_line);
}

/*填充节点对应的 IP 地址成员值*/
ipme->ipv4_addr = (u_int32_t)IP_struct.s_addr;

/*拆分 MAC 地址*/
macbytes = mSplit(toks[1], ":", 6, &num_macbytes, '\\');

```

```

if (num_macbytes < 6) /*MAC 地址不合法*/
{
    FatalError("Arpspoof %s(%d) => Invalid MAC address as second "
               "argument of IP/MAC pair to arpspoof_detect_host\n",
               file_name, file_line);

}

else
{
    for (i = 0; i < 6; i++) /*填充节点对应的 MAC 地址成员值*/
        ipme->mac_addr[i] = (u_int8_t) strtoul(macbytes[i], NULL, 16);

}

AddIPMacEntryToList(ipmel, ipme); /*填充节点*/

mSplitFree(&toks, num_toks);
mSplitFree(&macbytes, num_macbytes);

#if defined(DEBUG)
    PrintIPMacEntryList(ipmel);
#endif

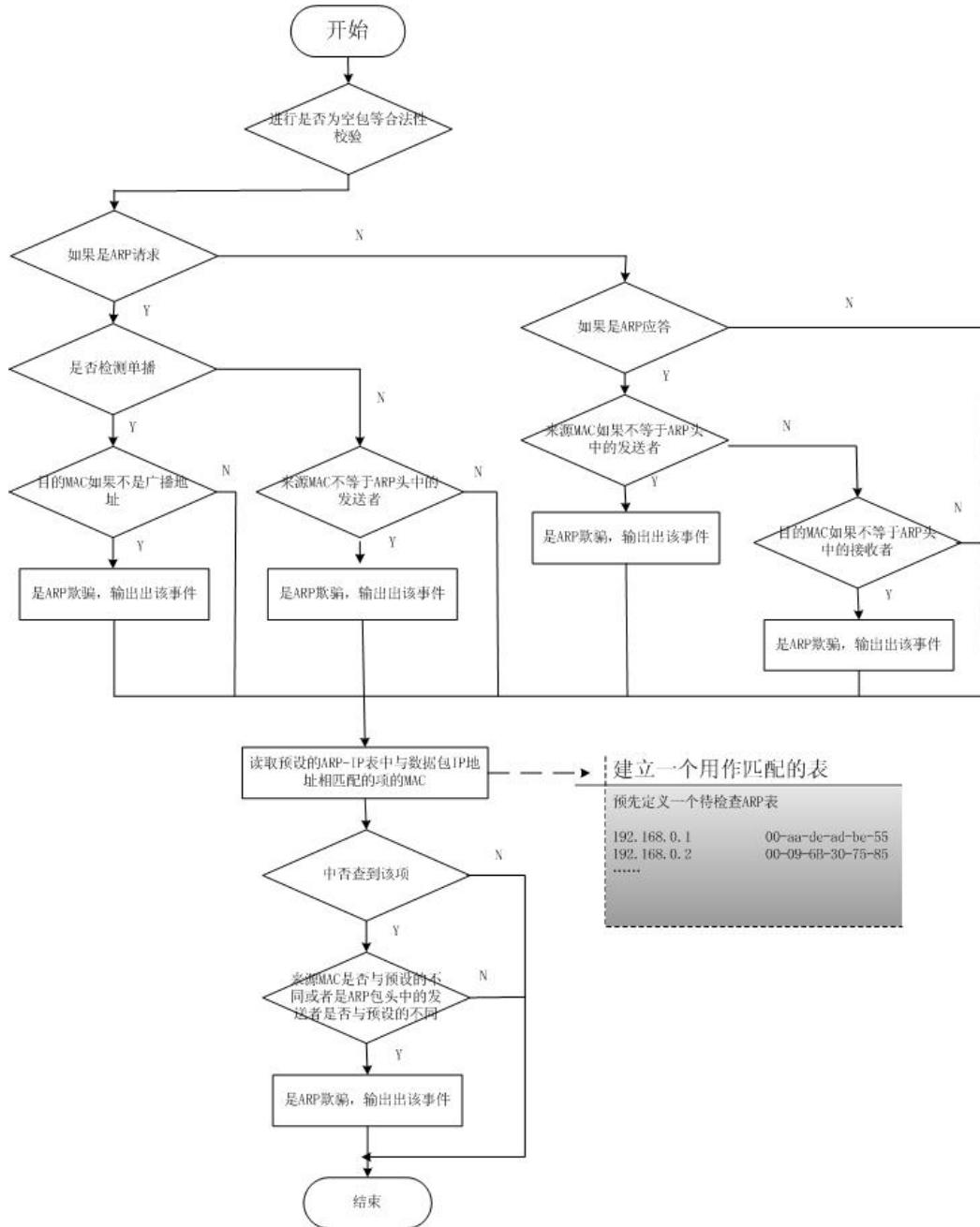
return;
}

```

这样，定义的表项被填入了链表 ipmel 对应的节点当中。检测，也就是遍历这个链表的过程了。

检测工作主要是在 DetectARPAttacks 函数中完成的，读者可以对照算法流程

图，分析其代码：



这样，每一个带有 ARP 欺骗性质的包都将被 Snort 及时地检测出来。

# 第10章 数据包检测引擎

系统在进行了数据包的截取、解码、嗅探模式的处理、日志模式的处理，如果系统是运行在 IDS 模式，系统将会进入检测引擎，配匹预定义的规则，检测攻击行为。

## 10. 1 Detect

Detect 函数是检测引擎的入口。如上文所述，在函数 Preprocess 中，通过调用 Detect 函数，进入实际的数据包检测：

```
if(do_detect)
{
    retval = Detect(p);
}
```

Detect 调用 fpEvalPacket 函数进入多模式匹配引擎。源码分析如下：

```
int Detect(Packet * p)
{
    int detected = 0;
    Event event;
    RuleListNode *rule;
    rule = RuleLists;
    check_tags_flag = 1;
    if(p && p->iph == NULL)           /*如果 IP 包头为空*/
        return 0;

    detected = fpEvalPacket(p);      //调用 fpEvalPacket 实际检测

    DEBUG_WRAP(DebugMessage(DEBUG_FLOW, "Checking tags list (if "
    "check_tags_flag = 1)\n"));

    /* 如果没有任何配匹的规则，检查 tag list */
    if(check_tags_flag == 1)
    {
```

```

DEBUG_WRAP(DebugMessage(DEBUG_FLOW, "calling CheckTagList\n"));

if(CheckTagList(p, &event))
{
    DEBUG_WRAP(DebugMessage(DEBUG_FLOW, "Matching tag node found,
"
                           "calling log functions\n"));

    /* 发现匹配，输出之*/
    CallLogFuncs(p, "Tagged Packet", NULL, &event);

    return 1;
}
return detected;
}

```

## 10. 2 fpEvalPacket

`fpEvalPacket` 函数是检测引擎的接口，主要是判断待检测的数据包是 TCP、UDP、ICMP/还是 IP 包，然后调用相应的 `fpEvalHeaderXXX` 检测函数。源码分析如下：

```

int fpEvalPacket(Packet *p)
{
    int ip_proto = p->iph->ip_proto;           //取得上层协议值
    switch(ip_proto)   //分别判断 TCP/UDP/ICMP，调用相应 fpEvalHeaderXXX 函数
    {
        case IPPROTO_TCP:
            DEBUG_WRAP(DebugMessage(DEBUG_DETECT,
                                      "Detecting on TcpList\n"));

            if(p->tcph == NULL)  /*如果是 tcp 协议但是协议头为空*/
            {
                ip_proto = -1;
                break;
            }

```

```

return fpEvalHeaderTcp(p);      //检测 TCP 包

case IPPROTO_UDP:
    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,
                           "Detecting on UdpList\n"));

    if(p->udph == NULL)
    {
        ip_proto = -1;
        break;
    }

return fpEvalHeaderUdp(p);      //检测 UDP 包

case IPPROTO_ICMP:
    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,
                           "Detecting on IcmpList\n"));

    if(p->icmph == NULL)
    {
        ip_proto = -1;
        break;
    }

return fpEvalHeaderIcmp(p);     //检测 ICMP 包

default:
    break;
}

return fpEvalHeaderIp(p, ip_proto); //进行 IP 包检测
}

```

可见，在进行了协议判断后，进行相应的协议检测的函数有四个 `fpEvalHeaderTcp`, `fpEvalHeaderUdp`, `fpEvalHeaderIcmp`, `fpEvalHeaderIp`。这四个函数的流程都是非常类似的。下文将以最为常用的 Tcp 包协议检测的 `fpEvalHeaderTcp` 函数进行分

析，其它三个函数，读取可以对比后自行分析。

## 10. 3 fpEvalHeaderTcp

### 10. 3. 1 函数流程分析

系统判断了当前数据包如果是 tcp 包的话，将调用 fpEvalHeaderTcp 函数将该数据包与规则子集合进行匹配。函数首先调用 prmFindRuleGroupTcp 函数获取适合当前规则的子集合，然后调用 InitMatchInfo 函数对初始化模块进行进一步优化，最后调用 fpEvalHeaderSW 函数进入多模式搜索匹配工作引擎。函数源码分析如下：

```
static INLINE int fpEvalHeaderTcp(Packet *p)
{
    PORT_GROUP *src, *dst, *gen;
    int retval;
    /*查找当前数据包适合哪一个子集合*/
    retval = prmFindRuleGroupTcp(p->dp, p->sp, &src, &dst, &gen);

    switch(retval)
    {
        case 0:           /*全部不配匹，退出*/
            return 0;
        case 1:
            InitMatchInfo( &omd );          /*对检测规则进行进一步优化*/

            /* 在目的端口组中,继续进行检测 */
            if(fpEvalHeaderSW(dst, p, 1))
            {
                return 1;
            }
            break;
        case 2:
            InitMatchInfo( &omd );

            /* 在源端口组中 */
            if(fpEvalHeaderSW(src, p, 1))
```

```

{
    return 1;
}
break;

case 3:
InitMatchInfo( &omd );

/* 既匹配目的端口组，又符合源端口组 */
if(fpEvalHeaderSW(dst, p, 1))
{
    return 1;
}
if(fpEvalHeaderSW(src, p, 1))
{
    return 1;
}
break;

case 4:
InitMatchInfo( &omd );

/* 通用规则组 */
if(fpEvalHeaderSW(gen, p, 1))
{
    return 1;
}
break;

default:
return 0;
}

return fpFinalSelectEvent(&omd, p);
}

```

## 10. 3. 2 prmFindRuleGroupTcp

函数 prmFindRuleGroupTCP 的作用是查找与当前待检测的 TCP 数据包相匹

配的规则子集合。查找算法是根据建立规则快速匹配引擎时的划分依据来的。上文提到，在上层函数中调用

```
retval = prmFindRuleGroupTcp(p->dp, p->sp, &src, &dst, &gen);  
p->dp/p->sp 表示目的/源端口，即为查找的依据，而查找的结果，是存放在  
&src、&dst,或者&gen 当中的。函数源码如下：
```

```
int prmFindRuleGroupTcp(int dport, int sport, PORT_GROUP ** src,  
PORT_GROUP ** dst, PORT_GROUP ** gen)  
{  
    return prmFindRuleGroup( prmTcpRTNX, dport, sport, src, dst, gen);  
}
```

由源码可见，函数 `prmFindRuleGroupTcp` 的作用是转向，调用 `prmFindRuleGroup` 在全局变量 `prmTcpRTNX` 中查找。

函数 `prmFindRuleGroup` 用于查找与此数据包匹配的规则子集合，定义如下：

```
int prmFindRuleGroup( PORT_RULE_MAP * p, int dport, int sport, PORT_GROUP ** src,  
PORT_GROUP ** dst, PORT_GROUP ** gen)
```

- `PORT_RULE_MAP * p:` 在哪儿找
- `int dport, int sport:` 目的/源端口，
- `PORT_GROUP ** src, PORT_GROUP ** dst, PORT_GROUP ** gen:` 如果查到匹配，则将其置于相应的 src/dst/gen 中。

当然，有可能一个数据包，即符合 `src`，又符合 `dst`，所以，要留意一下函数的返回值：

```
int - 0: No rules  
      1: Use Dst Rules  
      2: Use Src Rules  
      3: Use Both Dst and Src Rules  
      4: Use Generic Rules
```

其源码分析如下：

```
int prmFindRuleGroup( PORT_RULE_MAP * p, int dport, int sport, PORT_GROUP ** src,
```

```

PORT_GROUP **dst , PORT_GROUP ** gen)

{
    int stat= 0;

/*检测 dport 的合法性: (dport != ANYPORT && dport < MAX_PORTS), 并且符合 dst
子集合: p->prmDstPort[dport]*/
if( (dport != ANYPORT && dport < MAX_PORTS) && p->prmDstPort[dport] )

{
    *dst = p->prmDstPort[dport];
    stat = 1;
}

}else{

    *dst=NULL;
}

/*检测 sport 的合法性: (sport != ANYPORT && sport < MAX_PORTS), 并且符合 src
子集合: p->prmSrcPort[sport]*/
if( (sport != ANYPORT && sport < MAX_PORTS ) && p->prmSrcPort[sport] )

{
    *src = p->prmSrcPort[sport];
    stat |= 2;           //看看是否是即符合 dst,又是符合 src 的

}

}else{

    *src = NULL;
}

/* If no Src/Dst rules - use the generic set, if any exist */
if( !stat && (p->prmGeneric > 0) )

{

```

```

*gen = p->prmGeneric;

stat = 4;

}else{

*gen = NULL;

}

return stat;

}

```

其它协议的规则子集合的匹配查找与 TCP 协议完全相同，读者可以自行分析。

### 10. 3. 3 InitMatchInfo

InitMatchInfo 中涉及到一个很重要的变量 omd。在 fpdetect.c 中定义了全局变量： static OTNX\_MATCH\_DATA omd，而 OTNX\_MATCH\_DATA 结构定义如下 (fpdetect.c)：

```

typedef struct

{
    PORT_GROUP * pg;
    Packet * p;
    int check_ports;

    MATCH_INFO *matchInfo;
    int iMatchInfoArraySize;
} OTNX_MATCH_DATA;

```

该结构主要用于记录在整个数据包匹配期间涉及到的相关信息。

PORT\_GROUP \* pg： 规则快速匹配引擎的规则子集合

Packet \* p: 检测的数据包

int check\_ports: 是否需要检测端口 (即判断是 UDP/TCP 还是 ICMP 协议)

int iMatchInfoArraySize: 规则类型的数量, 默认情况下为五类(Alert、Pass……)

MATCH\_INFO \*matchInfo: 该结构用于存储某种类型规则下的事件匹配的相关信息, 定义如下 (fpdetect.c):

```
typedef struct {  
    OTNX *MatchArray[MAX_EVENT_MATCH];  
    int iMatchCount;  
    int iMatchIndex;  
    int iMatchMaxLen;  
}MATCH_INFO;
```

函数 InitMatchInfo 主要用于初始化 OTNX\_MATCH\_DATA 结构, 其实主要是初始化 MATCH\_INFO 结构的:

```
static INLINE void InitMatchInfo(OTNX_MATCH_DATA *o)  
{  
    int i = 0;  
    for(i = 0; i < o->iMatchInfoArraySize; i++)  
    {  
        o->matchInfo[i].iMatchCount = 0;  
        o->matchInfo[i].iMatchIndex = 0;  
        o->matchInfo[i].iMatchMaxLen = 0;  
    }  
}
```

omd 的其它部份成员已经在函数 OtnXMatchDataInitialize (fpdetect.c)中进行了初始化, 如下:

```
omd.iMatchInfoArraySize = pv.num_rule_types;  
if(!(omd.matchInfo = calloc(omd.iMatchInfoArraySize,  
                           sizeof(MATCH_INFO))))
```

```

{
    FatalError("Out of memory initializing detection engine\n");
}

```

## 10. 4 fpEvalHeaderSW

### 10. 4. 1 函数流程

fpEvalHeaderSW 是检测引擎中最为最要的一个函数，其主要特点是在于引入了多模式搜索匹配工作引擎，另外加强了对 TCP 流状态的处理。

系统经过前面的处理，根据协议、端口查找到了被检测数据包对应的根据端口划分的规则子集合。对于每一类规则子集合（PORT\_GROUP 结构）来讲，包含了三类节点：包含 content，包含 uricontent 和没有包含 content 的规则节点。所以，对一个数据包来讲，需要对这三类规则节点，进行逐一匹配。函数源码分析如下：

```

/*
**  PORT_GROUP *port_group: 待匹配的端口子集合
**  Packet *p: 待检测的数据包
**  int check_ports: 源/目的端口是否应被检测，由些可以判断是 udp/tcp 还是 icmp 协议
*/
static  INLINE  int  fpEvalHeaderSW(PORT_GROUP  *port_group,  Packet  *p,  int
check_ports)
{
    RULE_NODE *rnWalk;
    OTNX *otnx = NULL;
    void * so;

    extern HttpUri  UriBufs[URL_COUNT]; /* decode.c */

/*一个 TCP 攻击数据包将有可能被分成多个数据包，以逃避入侵检测的检测。Snort 采
用 Stream4 插件完成 TCP 会话重组，实现跨多个包的攻击检测。但是，为了避免一个
包被重庆检测，所以，我们必须等到其重组完成后，才能进入 Content 或 Uri-Content

```

```

的检测*/
if(fpDetect->inspect_stream_insert ||
   !(p->packet_flags & PKT_STREAM_INSERT))
{
    /* 如果 http_decode 插件至少找到了一个 uri, 则进行 Uri-Content 匹配*/
    if( p->uri_count > 0)
    {
        int i;
        so = (void *)port_group->pgPatDataUri; /*取得对应的节点*/

        if( so ) /* Do we have any URI rules ? */
        {
            mpseSetRuleMask( so, &port_group->boRuleNodeID );

            /*遍历规则子集合, 匹配所有 Uri-Content */
            for( i=0; i<p->uri_count; i++)
            {
                if(UriBufs[i].uri == NULL)
                    continue;

                omd.pg = port_group;
                omd.p = p;
                omd.check_ports= check_ports;

                mpseSearch (so, UriBufs[i].uri, UriBufs[i].length,
                           otnx_match, &omd);
            }
        }
    }

    if(UriBufs[0].decode_flags & HTTPURI_PIPELINE_REQ)
    {
        boResetBITOP(&(port_group->boRuleNodeID));
        return 0;
    }

    so = (void *)port_group->pgPatData;
    /* 这里有一个 p->packet_flags & PKT_ALT_DECODE 的判断 ,

```

PKT\_ALT\_DECODE 标志位用来表示数据包是否已经被规范化处理，在目前版本中，主要是涉及到预处理插件 telnet\_negotiation 对 telnet 会话协商的规范化处理。因为被规范化处理后的数据是放在全局变量 DecodeBuffer 当中的，所以这里要加这样一个判断：如果已被规范化处理，则传递给检测函数的数据包内容为 DecodeBuffer，否则为 p->data

```

*/
if((p->packet_flags & PKT_ALT_DECODE) && so && p->alt_dsize)
{
    mpseSetRuleMask( so, &port_group->boRuleNodeID );

    omd.pg = port_group;
    omd.p = p;
    omd.check_ports= check_ports;

    mpseSearch ( so, DecodeBuffer, p->alt_dsize,
                 otnx_match, &omd );

    boResetBITOP(&(port_group->boRuleNodeID));
}

if( so && p->data && p->dsiz)
{
    mpseSetRuleMask( so, &port_group->boRuleNodeID );

    omd.pg = port_group;
    omd.p = p;
    omd.check_ports= check_ports;

    mpseSearch ( so, p->data, p->dsiz, otnx_match, &omd );
}

boResetBITOP(&(port_group->boRuleNodeID));
}

/* 在进行了 Uri-Content 或 Content 节点的匹配后，我们就需要遍历 No-Content 的
规则节点了*/
for(rnWalk = port_group->pgHeadNC; rnWalk; rnWalk = rnWalk->rnNext)
{
    doe_ptr = NULL;
}

```

/\*在构建快速规则匹配引擎的时候，我们就说过，RULE\_NODE 的字段 rnRuleData 定为空指针类型，在实际操作中，其实赋予的是 OTNX 类型的指针值。所以，这里，将值取到\*/

```
otnx = (OTNX *)rnWalk->rnRuleData;
```

/\*先调用 fpEvalOTN 函数进行规则选项节点的匹配，对于 OTN 的匹配，因为我们之前已进行了 Content 和 Uri\_Content 的匹配，所以，接下来的匹配的，就是各个选项关键字，而对于选项关键字的匹配，主要是依靠调用选择关键字插件函数来实现的，这一点，我们在 4.3 节中，已有涉及\*/

```
if(fpEvalOTN(otnx->otn, p))
{
    /* 如果 OTN 匹配，再检测 RTN */
    if(fpEvalRTN(otnx->rtn, p, check_ports))
    {
        port_group->pgQEvents++;
        UpdateQEvents();

        if( fpAddMatch(&comd, otnx, 0) )
        {
            continue;
        }
    }
    else
    {
        port_group->pgNQEvents++;
        UpdateNQEvents();
    }

    continue;
}
}

return 0;
}
```

从上文的分析我们可以看到，Uri-Content 或者是 Content 的匹配，主要是通过 mpseSearch 函数（10.4.2 节）来实现的。而 No-Content 的规则节点匹配，主

要是通过 fpEvalOTN 和 fpEvalRTN 函数（10.4.3 节）来实现的。

## 10. 4. 2 mpseSearch

fpEvalHeaderSW 在进行 uri-content 或 content 模式匹配的时候，都调用了函数 mpseSearch 来进行进一步的匹配。其共包含了五个参数：

void \*pv: 实际传递过来的是 port\_group->pgPatData, 这中间包含了检测算法等，具体赋值在是建立快速规则匹配引擎调用 BuildMultiPatternGroups 时指向的（具体请参考相关章节）。

unsigned char \* T: 等检测数据包的内容，不包含包头  
int n: 等检测数据包的大小，不包含包头；  
int ( \*action )(void\*id, int index, void \*data): 实际传递过来的是函数  
otnx\_match;  
void \* data: 空指针，调用的时候，实际指向的是传递的&omd  
omd.pg = port\_group; //规则子集合  
omd.p = p; //等检测的包  
omd.check\_ports= check\_ports; //等检测端口

函数算法流程很简单，根据预先设定的检测模式，调用相应的匹配函数：

```
switch( p->method )
{
    .....
    case MPSE_MWM:
        return mwmSearch( p->obj, T, n, action, data );
}
```

mwmSearch 函数再来判断检测算法，一般是采用 Boyer-Moore 算法，或者是 Wu-Manber 算法。关于算法的具体实现，笔者就不继续深入阐述了。感兴趣的读者可以对照附录中关于 Boyer-Moore 算法的介绍，继续分析下去。

## 10. 4. 3 fpEvalOTN 和 fpEvalRTN

fpEvalOTN 函数实现对规则链表 OTN 节点的匹配，判断是否存在与当前数据包匹配的规则。对于这些选项关键字的匹配，主要是依靠这些关键字的检测插件来实现的。我们在构建 OTN 节点链的时候，已经讲过其封装了。

```
static INLINE int fpEvalOTN(OptTreeNode *List, Packet *p)
{
    if(List == NULL)      /*如果指定的选项链表指针为空，则立即返回*/
        return 0;

    DEBUG_WRAP(DebugMessage(DEBUG_DETECT, "    => Checking Option Node
%d\n",
                           List->chain_node_number););
    if(List->type == RULE_DYNAMIC && !List->active_flag)
    {
        return 0;
    }

    /*如果当前选项节点结构中的处理函数列表指针为空，则显示致命错误信息*/
    if(List->opt_func == NULL)
    {
        FatalError("List->opt_func was NULL on option #%d!\n",
                  List->chain_node_number);
    }

    if((snort_runtime.capabilities.stateful_inspection == 1) &&
       (List->established == 1) &&
       ((p->packet_flags & PKT_STREAM_EST) == 0))
    {
        /* this OTN requires an established connection and it isn't
         * in that state yet, so continue to the next OTN
         */
        return 0;
    }

    /*调用相应的检测插件，匹配当前数据包*/
    if(!List->opt_func->OptTestFunc(p, List, List->opt_func))
```

```

    {
        return 0;
    }
    else
    {
        /* rule match actions are called from EvalHeader */
        return 1;
    }

    return 0;
}

```

这样，每一个数据包都将逐一匹配本条规则的所有选项关键字的检测插件函数，关于这些函数的具体实现，我们将在第 12 章进行分析。

当 OTN 节点匹配后，会调用 fpEvalRTN 函数 (fpdetect.c) 进一步进行 RTN 节点的匹配。源码分析如下：

```

static INLINE int fpEvalRTN(RuleTreeNode *rtn, Packet *p, int check_ports)
{
    if(rtn == NULL)
    {
        return 0;
    }

    /*先做一个简单的端口匹配：等检测数据包的端口比规则中低端口还要小，则不匹配当前规则，退出*/
    if(check_ports)
    {
        if(!(rtn->flags & EXCEPT_DST_PORT) && !(rtn->flags & BIDIRECTIONAL)
        && (p->dp < rtn->lport))
        {
            return 0;
        }
    }

    if(rtn->type == RULE_DYNAMIC)
    {
        if(!active_dynamic_nodes)

```

```

    {
        return 0;
    }

    if(rtn->active_flag == 0)
    {
        return 0;
    }
}

/*遍历检测函数列表，进行数据包的检测*/
if(!rtn->rule_func->RuleHeadFunc(p, rtn, rtn->rule_func))
{
    return 0;
}

return 1;
}

```

## 10. 5 检测事件的输出

这里所要提到的输出，并不是输出到终端等，而是从检测引擎中输出，留待输出插件来调用。

检测事件的输出，依赖于我们在分析 `InitMatchInfo` 函数时涉及到的重要变量 `omd`，因为它存储了相关重要的信息。这里，共涉及的重要函数有三个：`fpAddMatch`、`fpFinalSelectEvent` 和 `SnortEventqAdd`。

### 10. 5. 1 fpAddMatch

系统在检测完成后，首先会调用 `fpAddMatch` 来将事件添加进适当的匹配的队例：Alert、Pass 或者是 Log。

如果是进行 Content 或 Uri-Content 的匹配，会在函数 `otnx_match` (`fpdetect.c`) 中调用：

```
fpAddMatch(omd, otnx, pmd->pattern_size );
```

如果是 No-Content 的匹配，会在 OTN 和 RTN 节点成功匹配后调用：

```

if( fpAddMatch(&omd, otnx, 0) )
{
    continue;
}

```

函数有三个参数，omd 将用来存储检测事件的相关信息。Otnx 包含了一个 OTN 结构，而 OTN 结构中的 sigInfo 结构成员，则包含了事件的相关信息。我们前面提到过，omd 是一个 OTNX\_MATCH\_DATA 结构，该结构的成员变量 MatchArray，则为 OTNX 结构的指针，所以，我们就不难看出它们之间的关系了。而至于第三个参数，则表示模式匹配的长度，我们看到，如果是 No-Content 的匹配，则第三个参数为 0。

函数首先取得当前 OTN 规则节点对应的规则类型的索引：

```
evalIndex = otnx->otn->rtn->listhead->ruleListNode->evalIndex;
```

进而，取得存储当前规则类型事件信息的结构空间：

```
pmi = &omd->matchInfo[evalIndex];
```

在进行合法性判断后，将事件添加进适当的列表当中：

```
pmi->MatchArray[ pmi->iMatchCount ] = otnx;
```

然后，给相应成员变量赋值：

```

if(pLen > 0)          /*如果是 Content/Uri-Content rule*/
{
    if( pmi->iMatchMaxLen < pLen )      /*如果触发多个事件，将以匹配的长度来
确定优先级，当前规则匹配长度大于最大匹配长度，则更新最大长度记录，赋相应索引
值*/
    {
        pmi->iMatchMaxLen = pLen;
        pmi->iMatchIndex  = pmi->iMatchCount;
    }
}

pmi->iMatchCount++;      /*更新匹配次数*/

```

这样，函数执行完后，当前事件的相关信息就被添加进全局变量 omd 了。

## 10. 5. 2 fpFinalSelectEvent

前面我们在分析 fpEvalHeaderTcp 时，就已经提到，在进行完规则匹配之后，函数在末尾处调用：

```
fpFinalSelectEvent(&omd, p);  
来输出检测事件。
```

FpFinalSelectEvent 的主要作用是遍历 omd，以输出所有事件队例。函数源码分析如下：

```
static INLINE int fpFinalSelectEvent(OTNX_MATCH_DATA *o, Packet *p)  
{  
    int i;  
    int j;  
    int k;  
    OTNX *otnx;  
  
    for(i = 0; i < o->iMatchInfoArraySize; i++)      /*遍历所有规则类型*/  
    {  
        if(o->matchInfo[i].iMatchCount)  
        {  
            for(j=0; j < o->matchInfo[i].iMatchCount; j++) /*遍历事件队列*/  
            {  
                /*取得事件信息*/  
                otnx = o->matchInfo[i].MatchArray[j];  
  
                for(k = 0; k < j; k++)      /*防止事件被匹配多次*/  
                {  
                    if(o->matchInfo[i].MatchArray[k] == otnx)  
                    {  
                        otnx = NULL;  
                        break;  
                    }  
                }  
  
                if(otnx && otnx->otn)  
                {  
                    /*
```

```

        ** 添加事件
        */
        SnortEventqAdd(otnx->otn->sigInfo.generator,
                       otnx->otn->sigInfo.id,
                       otnx->otn->sigInfo.rev,
                       otnx->otn->sigInfo.class_id,
                       otnx->otn->sigInfo.priority,
                       otnx->otn->sigInfo.message,
                       (void *)otnx);
    }
}

return 1;
}
}

return 0;
}

```

可以看到，单一事件的输出，是靠 SnortEventqAdd 函数来实现的。

## 10. 5. 3 SnortEventqAdd

SnortEventqAdd 函数 (event\_queue.c) 不仅是在检测引擎中被调用，其实在解码引擎、预处理引擎当中，当数据包触发了某个条件后，系统都会调用 SnortEventqAdd 函数添加一个报警事件。

SnortEventqAdd 的参数主要包括两块：事件信息和一个 otnx 结构变量。事件信息被定义成一个 SigInfo 结构 (signature.h):

```

typedef struct _SigInfo
{
    u_int32_t generator;
    u_int32_t id;
    u_int32_t rev;
    u_int32_t class_id;
    ClassType *classType;
}

```

```
    u_int32_t priority;  
    char *message;  
    ReferenceNode *refs;  
} SigInfo;
```

对照 Snort 用户手册，不难理解这些成员变量的含义，这里给出从原文摘抄的相关内容：

#### ***reference***

这个关键字允许规则包含一个外面的攻击识别系统。这个插件目前支持几种特定的系统，它和支持唯一的URL一样好。这些插件被输出插件用来提供一个关于产生报警的额外信息的连接。

确信先看一看如下地方：

<http://www.snort.org/snort-db>

格式：

*reference: <id system>, <id>;*

例子：

```
alert tcp any any -> any 7070 (msg: "IDS411/dos-realaudio"; flags: AP; content:  
"/ffff4 fffd 06/"; reference: arachNIDS, IDS411;)  
alert tcp any any -> any 21 (msg: "IDS287/ftp-wuftp260-venglin-linux"; flags: AP;  
content: "|31c031db 31c9b046 cd80 31c031db|"; reference: arachNIDS, IDS287;  
reference: bugtraq, 1387; reference: cve, CAN-2000-1574; )
```

#### ***Sid***

这个关键字被用来识别snort规则的唯一性。这个信息允许输出插件很容易的识别规则的ID号。

*sid* 的范围是如下分配的：

<100 保留做将来使用

100-1000,000 包含在snort发布包中

>1000,000 作为本地规则使用

文件*sid-msg.map* 包含一个从*msg*标签到*snort*规则ID的映射。这将被*post-processing* 输出模块用来映射一个ID到一个报警信息。

格式:

*sid*: <snort rules id>;

#### **rev**

这个关键字是被用来识别规则修改的。修改，随同*snort*规则ID，允许签名和描述被较新的信息替换。

格式:

*rev*: <revision integer>

#### **Classtype**

这个关键字把报警分成不同的攻击类。通过使用这个关键字和使用优先级，用户可以指定规则类中每个类型所具有的优先级。具有*classification*的规则有一个缺省的优先级。

格式:

*classtype*: <class name>;

在文件*classification.config*中定义规则类。这个配置文件使用如下的语法:

*config classification*: <class name>, <class description>, <default priority>

#### **Priority**

这个关键字给每条规则赋予一个优先级。一个*classtype*规则具有一个缺省的优先级，但这个优先级是可以被一条*priority*规则重载的。

格式:

*priority*: <priority integer>;

219. 153. 6. 145/255. 255. 255. 248 219. 153. 6. 150

#### **msg**

把*msg*选项的内容包含进阻塞通知信息中

*SnortEventqAdd* 函数主要是把上层函数传递过来的相关信息赋给 *EventNode* 类型的变量 *en*，再由 *sfeventq\_add* 函数来负责添加

## 10. 4. 4 sfeventq\_add

函数 sfeventq\_add 在 sfeventq.c 中实现。事件的组织采用的是双向链表，链表头是全局变量 s\_eventq。

s\_eventq 定义在 sfeventq.c 中：

```
static SF_EVENTQ s_eventq;
```

我们先来看看 SF\_EVENTQ 结构的定义：

```
typedef struct s_SF_EVENTQ
{
    /*
     ** Handles the actual ordering and memory
     ** of the event queue and it's nodes.
     */

    SF_EVENTQ_NODE *head;
    SF_EVENTQ_NODE *last;
    SF_EVENTQ_NODE *node_mem;
    char           *event_mem;

    /*
     ** The reserve event allows us to allocate one extra node
     ** and compare against the last event in the queue to determine
     ** if the incoming event is a higher priority than the last
     ** event in the queue.
     */

    char           *reserve_event;

    /*
     ** Queue configuration
     */

    int max_nodes;
    int log_nodes;
    int event_size;

    /*
     ** This function orders the events as they
     ** arrive.
     */
}
```

```

*/
int (*sort)(void *event1, void *event2);

/*
** This element tracks the current number of
** nodes in the event queue.
*/

int cur_nodes;
int cur_events;
} SF_EVENTQ;

```

而其重要的成员变量，如 head 等，是一个 SF\_EVENTQ\_NODE 结构，定义如下：

```

typedef struct s_SF_EVENTQ_NODE
{
    void    *event;
    struct s_SF_EVENTQ_NODE *prev;
    struct s_SF_EVENTQ_NODE *next;
} SF_EVENTQ_NODE;

```

了解了这两个结构，我们就不难理解其源码了：

```

int sfeventq_add(void *event)
{
    SF_EVENTQ_NODE *node;
    SF_EVENTQ_NODE *tmp;

    if(!event)
        return -1;

    /*获取节点的空间*/
    node = get_eventq_node(event);
    if(!node)
        return 0;

    node->event = event;           /*取得事件信息*/
    node->next  = NULL;
    node->prev  = NULL;

    /*如果是首节点，直接添加之*/

```

```

if(s_eventq.cur_nodes == 1)      /*添加首节点*/
{
    s_eventq.head = s_eventq.last = node;
    return 0;
}

/*  查找插入位置 */
for(tmp = s_eventq.head; tmp; tmp = tmp->next)
{
    if(s_eventq.sort(event, tmp->event))
    {
        /*插入节点*/
        if(tmp->prev)
            tmp->prev->next = node;
        else
            s_eventq.head = node;

        node->prev = tmp->prev;
        node->next = tmp;

        tmp->prev = node;
    }
}

/* 如果是最后一个节点的话*/
node->prev = s_eventq.last;

s_eventq.last->next = node;
s_eventq.last = node;

return 0;
}

```

这样，检测事件就被构建成一个双向链表，留待输出插件来调用了。

# 第11章 规则链表头的检测函数

## 11. 1 概述

RTN 的检测函数的作用是做地址、端口、方向等的检测匹配。主要涉及了以下几个函数：

- CheckBidirectional : 如果规则中设置了双向地址操作标识，检查 IP 地址和端口值的匹配情况。
- CheckSrcPortNotEq : 检测当前规则链表头结构中的源端口是否与数据包中的源端口不相等；
- CheckDstPortNotEq : 检测当前规则链表头结构中的目的端口是否与数据包中的目的端口不相等；
- CheckSrcPortEqual : 检测当前规则链表头结构中的源端口是否与数据包中的源端口相等；
- CheckDstPortEqual : 检测当前规则链表头结构中的目的端口是否与数据包中的目的端口相等；
- CheckSrcIP : 检测来源 IP 地址；
- CheckDstIP : 检测目的 IP 地址；

## 11. 2 CheckBidirectional

Snort 规则头配置支持双向操作符“<>”，它告诉snort把地址/端口号对既作为源，又作为目标来考虑。这对于记录/分析双向对话很方便，例如telnet或者pop3会话。用来记录一个telnet会话的两侧的流的范例如下：

```
log !192.168.1.0/24 any <> 192.168.1.0/24 23
```

CheckBidirectional 函数的功能是在设置了双向地址操作标识符的情况下，检

测 IP 地址和端口值的匹配情况。它的工作特点是对于正向和反向地址操作的情况下，分别对地址和端口值的匹配情况进行检查。源码分析如下：

```
int CheckBidirectional(Packet *p, struct _RuleTreeNode *rtn_idx,
                      RuleFpList *fp_list)
{
    /*正向匹配：先检测来源地址及端口*/
    if(CheckAddrPort(rtn_idx->sip, rtn_idx->hsp, rtn_idx->lsp, p,
                      rtn_idx->flags, CHECK_SRC))
    {
        /*进一步检测目的地址及端口，如果匹配的话，则正向匹配成功，否则，把目的做来源，来源做目的，进行反向检测*/
        if(! CheckAddrPort(rtn_idx->dip, rtn_idx->hdp, rtn_idx->ldp, p,
                           rtn_idx->flags, CHECK_DST))
        {
            /*反向检测： Dst->Src */
            if(CheckAddrPort(rtn_idx->dip, rtn_idx->hdp, rtn_idx->ldp, p,
                             rtn_idx->flags, (CHECK_SRC | INVERSE)))
            {
                /*若匹配，继续检测： Src->Dst */
                if(!CheckAddrPort(rtn_idx->sip, rtn_idx->hsp, rtn_idx->lsp, p,
                                  rtn_idx->flags, (CHECK_DST | INVERSE)))
                {
                    return 0;
                }
                else      /*反向检测，匹配*/
                {
                    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,      "Inverse
addr/port match\n"));
                }
            }
            else      /*反向检测失败*/
            {
                return 0;
            }
        }
        else
        {
            DEBUG_WRAP(DebugMessage(DEBUG_DETECT,      "dest      IP/port

```

```

match\n"););
}
}
else /*算法类似正向检测*/
{
    if(CheckAddrPort(rtn_idx->dip, rtn_idx->hdp, rtn_idx->lsp, p,
                      rtn_idx->flags, CHECK_SRC | INVERSE))
    {
        if(! CheckAddrPort(rtn_idx->sip, rtn_idx->hsp, rtn_idx->lsp, p,
                           rtn_idx->flags, CHECK_DST | INVERSE))
        {
            return 0;
        }
        else
        {
            DEBUG_WRAP(DebugMessage(DEBUG_DETECT,
                                    "Inverse addr/port match\n"));
        }
    }
    else
    {
        return 0;
    }
}
return 1;
}

```

**提示：**我们可以从分级调式 `DEBUG_WRAP` 函数中输出的提示符就可以很顺利地了解函数的算法了。

实际的检测函数是通过 `CheckAddrPort` 函数来实现的，它将规则头中的地址、端口与数据包中的地址、端口进行匹配，若成功，则返回 1，否则返加 0。源码分析如下：

```

int CheckAddrPort(IpAddrSet *rule_addr, u_int16_t hi_port, u_int16_t lo_port,
                  Packet *p, u_int32_t flags, int mode)
{
    u_long pkt_addr;          /* packet IP address */
    u_short pkt_port;         /* packet port */

```

```

int global_except_addr_flag = 0; /* global exception flag is set */
int any_port_flag = 0;           /* any port flag set */
int except_addr_flag = 0;        /* any addr flag set */
int except_port_flag = 0;        /* port exception flag set */
int ip_match = 0;               /* flag to indicate addr match made */
IpAddrSet *idx;    /* ip addr struct indexer */

if(mode & CHECK_SRC) /*如果是检测来源地址*/
{
    pkt_addr = p->iph->ip_src.s_addr; /*获取来源地址*/
    pkt_port = p->sp;                 /*获取来源地址*/
    if(mode & INVERSE) /*如果设置了反向标志符*/
    {
        /*根据设置的目的地址求反标志，设置对应变量值*/
        global_except_addr_flag = flags & EXCEPT_DST_IP;
        /*根据设置的任意目的地址标志，设置对应变量值*/
        any_port_flag = flags & ANY_DST_PORT;
        /*根据设置的目的端口求反标志，设置对应变量值*/
        except_port_flag = flags & EXCEPT_DST_PORT;
    }
    else /*否则，设置相应变量值，类似*/
    {
        global_except_addr_flag = flags & EXCEPT_SRC_IP;
        any_port_flag = flags & ANY_SRC_PORT;
        except_port_flag = flags & EXCEPT_SRC_PORT;
    }
}
else /*如果是检测目的地址，类似*/
{
    pkt_addr = p->iph->ip_dst.s_addr;
    pkt_port = p->dp;

    if(mode & INVERSE)
    {
        global_except_addr_flag = flags & EXCEPT_SRC_IP;
        any_port_flag = flags & ANY_SRC_PORT;
        except_port_flag = flags & EXCEPT_SRC_PORT;
    }
}

```

```

        else
        {
            global_except_addr_flag = flags & EXCEPT_DST_IP;
            any_port_flag = flags & ANY_DST_PORT;
            except_port_flag = flags & EXCEPT_DST_PORT;
        }
    }

idx = rule_addr;
if(!(global_except_addr_flag)) /*如果地址值未取反*/
{
    while(idx != NULL)      /*循环匹配数据包与规则中的地址列表*/
    {
        except_addr_flag = idx->addr_flags & EXCEPT_IP;
        if(((idx->ip_addr == (pkt_addr & idx->netmask)) ^ except_addr_flag))
        {
            idx = idx->next;
        }
        else
        {
            ip_match = 1;      /*匹配，跳出循环*/
            goto bail;
        }
    }
}
else /*如果设置了地址取反标志， 算法类似*/
{
    while(idx != NULL)
    {
        except_addr_flag = idx->addr_flags & EXCEPT_IP;
        if(((idx->ip_addr == (pkt_addr & idx->netmask)) ^
            except_addr_flag))
        {
            return 0;
        }
        idx = idx->next;
    }
    ip_match = 1;
}

```

```

    }

bail:
if(!ip_match) /*循环结束，没有找到匹配，退出*/
{
    return 0;
}

/*如设置了任意端口标志，则不用再继续匹配了，成功退出*/
if(any_port_flag)
{
    return 1;
}

/*如果数据包端口大于高端口又小于低端口*/
if((pkt_port > hi_port) || (pkt_port < lo_port))
{
    /*但是又没有设置取反标志，则不匹配，退出，反之则匹配，继续检查*/
    if(!except_port_flag)
    {
        return 0;
    }
}

else /*数据包在检测范围之内*/
{
    /*但是又设置了取反标志，则不匹配，退出，反之则匹配，继续检查*/
    if(except_port_flag)
    {
        return 0;
    }
}

/*地址、端口均匹配，成功返回*/
return 1;
}

```

## 11. 3 端口的检测

关于端口的检测，需要匹配来源端口、目的端口，同时，如果规则中有不等标志“！”，还需要检测不等的情况。共涉及以下四个函数：

- CheckSrcPortNotEq
- CheckDstPortNotEq
- CheckSrcPortEqual
- CheckDstPortEqual

这四个函数的算法都非常地类似，这里，以检测来源端口是否相等的函数 CheckSrcPortEqual 为例进行分析：

```
int CheckSrcPortEqual(Packet *p, struct _RuleTreeNode *rtn_idx,
                      RuleFpList *fp_list)
{
    /*如果等检测端口在规则端口范围之内，则匹配成功，继续检测，否则表示不匹配，退出*/
    if( (p->sp <= rtn_idx->hsp) && (p->sp >= rtn_idx->lsp) )
    {
        DEBUG_WRAP(DebugMessage(DEBUG_DETECT, " SP match!\n"));
        return fp_list->next->RuleHeadFunc(p, rtn_idx, fp_list->next);
    }
    else
    {
        DEBUG_WRAP(DebugMessage(DEBUG_DETECT, " SP mismatch!\n"));
    }

    return 0;
}
```

## 11. 4 地址的检测

地址检测主要包含源地址的匹配和目的地址的匹配，另外，还包含需要处理设置了取反标志符“!”的情况，共涉及两个函数：CheckSrcIP 和 CheckDstIP。这里以 CheckSrcIP 为例，分析其具体实现：

```
int CheckSrcIP(Packet * p, struct _RuleTreeNode * rtn_idx, RuleFpList * fp_list)
{
    IpAddrSet *idx; /* ip address indexer */
    /*检测是否设置了取反标志符 */
    if(!(rtn_idx->flags & EXCEPT_SRC_IP))
```

```

{
    /* 遍历地址列表，匹配数据包*/
    for(idx=rtn_idx->sip; idx != NULL; idx=idx->next)
    {
        if( ((idx->ip_addr == (p->iph->ip_src.s_addr & idx->netmask))
            ^ (idx->addr_flags & EXCEPT_IP)) )

        {

#define DEBUG
        if(idx->addr_flags & EXCEPT_IP) {
            DebugMessage(DEBUG_DETECT, "  SIP exception match\n");
        }
        else
        {
            DebugMessage(DEBUG_DETECT, "  SIP match\n");
        }

        DebugMessage(DEBUG_DETECT, "Rule: 0x%X           Packet:
0x%X\n",
                    idx->ip_addr, (p->iph->ip_src.s_addr & idx->netmask));
#endif /* DEBUG */

        /* 当前数据包与规则匹配，继续检测*/
        return fp_list->next->RuleHeadFunc(p, rtn_idx, fp_list->next);
    }
}
else
{
    /* global exception flag is up, we can't match on *any*
     * of the source addresses
     */
    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,"  global exception flag,
\n"));
}

/* do the check */
for(idx=rtn_idx->sip; idx != NULL; idx=idx->next)
{
    if( ((idx->ip_addr == (p->iph->ip_src.s_addr & idx->netmask))

```

```

        ^ (idx->addr_flags & EXCEPT_IP)) )
{
    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,"address matched,
failing on SIP\n"););
    /* got address match on globally negated rule, fail */
    return 0;
}
DEBUG_WRAP(DebugMessage(DEBUG_DETECT,"no matches on SIP,
passed\n"););

return fp_list->next->RuleHeadFunc(p, rtn_idx, fp_list->next);
}

DEBUG_WRAP(DebugMessage(DEBUG_DETECT," Mismatch on SIP\n"););

/* return 0 on a failed test */
return 0;
}

```

## 第12章 规则选项关键字插件的实现

### 12. 1 负载长度的检测

负载长度关键字 (dszie) 选项用于定义被捕获包的负载长度或长度范。可以是大于、小于、等于，或者在一个范围内。比如我们定义了如下规则：

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 较大的 ICMP 包";
dszie:>800;.....
```

那么我们发送一个大于 800 字节的 icmp 包，将触发 Snort 发送 "ICMP 较大的 ICMP 包" 的警告信息。

Dsize 关键字对应的检测插件实现在 sp\_dsize\_check.c 中。共包含：

```
void SetupDsizeCheck(void);
void DsizeCheckInit(char *, OptTreeNode *, int);
```

```

void ParseDsize(char *, OptTreeNode *);

int CheckDsizeEq(Packet *, struct _OptTreeNode *, OptFpList *);

int CheckDsizeGT(Packet *, struct _OptTreeNode *, OptFpList *);

int CheckDsizeLT(Packet *, struct _OptTreeNode *, OptFpList *);

int CheckDsizeRange(Packet *, struct _OptTreeNode *, OptFpList *);

```

七个函数。前两个函数用于插件的注册和初始化，这里不再赘述。第三个函数用于解析对应的规则配置参数。后四个函数用于分别处理等于、大于、小于或者在某个范围内的四个情况。

函数 ParseDsize 的功能是解析 dsize 关键字的参数并存入对应的 OTN 结构字段中，然后根据解析结果，在规则选项节点中链入对应的处理函数：

```

void ParseDsize(char *data, OptTreeNode *otn)
{
    DsizeCheckData *ds_ptr;
    char *pcEnd;
    char *pcTok;
    int iDsize = 0;

    /* 使指针 ds_ptr 指向选项节点结构字段中适当的位置*/
    ds_ptr = (DsizeCheckData *)otn->ds_list[PLUGIN_DSIZE_CHECK];
    /*清除参数字符串前起始的空格符，直至第一个不为空格的字符*/
    while(isspace((int)*data)) data++;

    /* 判断是否是定义了一个范围值*/
    if(isdigit((int)*data) && strchr(data, '<') && strchr(data, '>'))
    {
        pcTok = strtok(data, " <>");
        if(!pcTok)
        {
            FatalError("%s(%d): Invalid 'dsize' argument.\n",
                      file_name, file_line);
        }

        iDsize = strtol(pcTok, &pcEnd, 10);
        if(iDsize < 0 || *pcEnd)
        {

```

```

        FatalError("%s(%d): Invalid 'dsiz' argument.\n",
                   file_name, file_line);
    }

    ds_ptr->dsiz = (unsigned short)iDsize;

    pcTok = strtok(NULL, " <>");
    if(!pcTok)
    {
        FatalError("%s(%d): Invalid 'dsiz' argument.\n",
                   file_name, file_line);
    }

    iDsize = strtol(pcTok, &pcEnd, 10);
    if(iDsize < 0 || *pcEnd)
    {
        FatalError("%s(%d): Invalid 'dsiz' argument.\n",
                   file_name, file_line);
    }

    ds_ptr->dsiz2 = (unsigned short)iDsize;
    /*如果当前字符是范围值，链入对应的处理函数 CheckDsizeRange */
    AddOptFuncToList(CheckDsizeRange, otn);
    return;
}

else if(*data == '>') /*如果当前字符是'>'，链入对应的处理函数 CheckDsizeGT */
{
    data++;
    AddOptFuncToList(CheckDsizeGT, otn);
}

else if(*data == '<') /*如果当前字符是'<'，链入对应的处理函数 CheckDsizeLT */
{
    data++;
    AddOptFuncToList(CheckDsizeLT, otn);
}

else /*剩下的就是等于的情况了*/
{
    AddOptFuncToList(CheckDsizeEq, otn);
}

```

```

    }

while(isspace((int)*data)) data++;

iDsize = strtol(data, &pcEnd, 10);
if(iDsize < 0 || *pcEnd)
{
    FatalError("%s(%d): Invalid 'dsiz' argument.\n",
               file_name, file_line);
}
ds_ptr->dsiz = (unsigned short)iDsize;
}

```

对应的四个检测函数，它们在算法上都是一样的。这里以 CheckDsizeEq 为例进行分析：

```

int CheckDsizeEq(Packet *p, struct _OptTreeNode *otn, OptFpList *fp_list)
{
    /* dsiz 关键字对于经过重组生成的包是无效的*/
    if(p->packet_flags & PKT_REBUILT_STREAM)
    {
        return 0;
    }
    /*如果规则中设置的数值等于被捕获包的负载长度*/
    if(((DsizeCheckData *)otn->ds_list[PLUGIN_DSIZE_CHECK])->dsiz == p->dsiz)
    {
        /*继续调用下一个处理函数，然后返回 */
        return fp_list->next->OptTestFunc(p, otn, fp_list->next);
    }
    else
    {
        DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "Not equal\n"));
    }
    return 0;
}

```

## 12. 2 Session 关键字

面对攻击时，Session 选项关键字是最有用的功能之一，用于从 TCP 会话中抽取用户数据。要检查用户在 telnet，rlogin，ftp 或 web sessions 中的用户输入，这个规则选项特别有用。它对应的检测插件在 sp\_session.c 中实现，结构与 dsize 类似，包含了：

```
void SetupSession(void);
void SessionInit(char *, OptTreeNode *, int);
void ParseSession(char *, OptTreeNode *);
int LogSessionData(Packet *, struct _OptTreeNode *, OptFpList *);
void DumpSessionData(FILE *, Packet *, struct _OptTreeNode *);
FILE *OpenSessionFile(Packet *);
```

六个函数。

ParseSession 函数用于关键字参数配置的解析：

```
void ParseSession(char *data, OptTreeNode *otn)
{
    SessionData *ds_ptr; /* data struct pointer */

    /* 取得规则选项结构节点的对应字段值 */
    ds_ptr = otn->ds_list[PLUGIN_SESSION];

    /* 去掉参数字符串前的空格符*/
    while(isspace((int)*data))
        data++;

    /*判断参数，设置对应的标识。在传统的 printable 和 all 参数上，Snort2.2 增加了一个 binary 参数*/
    if(!strncasecmp(data, "printable", 9))
    {
        ds_ptr->session_flag = SESSION_PRINTABLE;
        return;
    }

    if(!strncasecmp(data, "binary", 6))
    {
        ds_ptr->session_flag = SESSION_BINARY;
```

```

        return;
    }

    if(!strncasecmp(data, "all", 3))
    {
        ds_ptr->session_flag = SESSION_ALL;
        return;
    }
    FatalError("%s(%d): invalid session modifier: %s\n", file_name, file_line, data);
}

```

函数 LogSessionData 的功能是调用相应的子例程来记录进程的信息：

```

int LogSessionData(Packet *p, struct _OptTreeNode *otn, OptFpList *fp_list)
{
    FILE *session;           /* session file ptr */

    /* 对数据包负载、大于、分片等进行较验 */
    if((p != NULL && p->dsize != 0 && p->data != NULL) || p->frag_flag != 1)
    {
        session = OpenSessionFile(p);      /*打开一个用来记录进程信息的文件*/

        if(session == NULL)   /*打开失败，则调用下一个检测函数进行继续检测*/
        {
            return fp_list->next->OptTestFunc(p, otn, fp_list->next);
        }
        /*文件打开操作成功，调用 DumpSessionData 来记录相应的进程信息*/
        DumpSessionData(session, p, otn);

        fclose(session);          /*关闭文件*/
    }
    /*继续调用下一个处理函数*/
    return fp_list->next->OptTestFunc(p, otn, fp_list->next);
}

```

函数 DumpSessionData 的功能是处理进程信息的实际记录工作：

```

void DumpSessionData(FILE *fp, Packet *p, struct _OptTreeNode *otn)
{
    u_char *idx;

```

```

u_char *end;

char conv[] = "0123456789ABCDEF"; /* xlation lookup table */

/*若包负载为空或是空包或分片包，则返回*/
if(p->dsize == 0 || p->data == NULL || p->frag_flag)
    return;

idx = p->data;           /*取得数据包负载的起始指针*/
end = idx + p->dsize;    /*设置内容负载的结束位置指针*/
/*如果设置了 SESSION_PRINTABLE 标志，对应参数 printable ，则在负载中遍历所有的文本字符，并逐一写入记录文件中*/
if(((SessionData *) otn->ds_list[PLUGIN_SESSION])->session_flag == SESSION_PRINTABLE)
{
    while(idx != end)
    {
        /*可以看出，还包含了回车和换行符*/
        if((*idx > 0x1f && *idx < 0x7f) || *idx == 0x0a || *idx == 0x0d)
        {
            fputc(*idx, fp);
        }
        idx++;
    }
}
/*如果是设置了标识 SESSION_BINARY ，直接写入二进制数据*/
else if(((SessionData *) otn->ds_list[PLUGIN_SESSION])->session_flag == SESSION_BINARY)
{
    fwrite(p->data, p->dsize, sizeof(char), fp);
}
else /*如果是参数 all*/
{
    while(idx != end)
    {
        /*对于文本字符（包括换行和回车符），逐个写入到记录文件中*/
        if((*idx > 0x1f && *idx < 0x7f) || *idx == 0x0a || *idx == 0x0d)
        {
            /* Escape all occurrences of '\' */
            if(*idx == '\\')

```

```

        fputc('\\', fp);
        fputc(*idx, fp);
    }

/*对于其他二进制值，则将其转化为 16 进制的字符形式写入到文件中。
其中用到的一个技巧是使用了一个专门的转化字符数组，其对应 0~15 下标位置的元素为 16 进制表示的对应字符*/
else
{
    fputc('\\', fp);
    fputc(conv[((*idx&0xFF) >> 4)], fp);
    fputc(conv[((*idx&0xFF)&0x0F)], fp);
}

idx++;
}

}
}

```

`OpenSessionFile` 的作用是打开用来记录进程信息的文件。函数处理的关键是如何确定存储目录及文件名。这个是根据数据包的地址字段信息来确定的。关于这个函数，读者可以自行分析。

## 12. 3 ICMP 选项集合

`Snort` 的攻击特征描述中有四种与 ICMP 有关的选项，用来检测 ICMP 包中不同字段，包括 ICMP 的类型、代码、ID 和序列号。它们对应的插件的源文件是 `sp_icmp_type_check.c`、`sp_icmp_code_check`、`sp_icmp_id_check.c` 和 `sp_icmp_seq_check`。

IP 选项集合和 TCP 选项集合与 ICMP 比较类似，分析完本节后，读者可以对照自行分析以下关键字插件的实现：

```

sp_tcp_ack_check.c
sp_tcp_flag_check.c
sp_tcp_seq_check.c
sp_tcp_win_check.c

```

```
sp_ip_fragbits.c  
sp_ip_id_check.c  
sp_ip_proto.c  
sp_ip_same_check.c  
sp_ip_tos_check.c  
sp_ipoption_check.c  
sp_ttl_check.h
```

## 12. 3. 1 ID

ICMP ID 用于匹配 ICMP 回响请求(ICMP\_ECHO)中的 ID 域。根据 Snort 开发组的解释，这个选项用于检测那些利用 ICMP 包进行通信的欺骗性程序。它的格式为：icmp\_id:参数值。

sp\_icmp\_id\_check.c 中共包含了注册、初始化、参数解析和检测四个函数。这里重点讲解检测函数 IcmpIdCheck 的实现：

```
int IcmpIdCheck(Packet *p, struct _OptTreeNode *otn, OptFpList *fp_list)  
{  
    if(!p->icmph)      /*若 ICMP 包头为空，则返回*/  
        return 0;  
    /*只检测 ICMP 的请求与回应类型*/  
    if(p->icmph->type == ICMP_ECHO || p->icmph->type == ICMP_ECHOREPLY)  
    {  
        /* 如果 ICMP 包中的 ID 值与规则的 ID 值相匹配，则继续调用下一个处理函  
数，然后返回 */  
        if(((IcmpIdData *) otn->ds_list[PLUGIN_ICMP_ID_CHECK])->icmpid ==  
            p->icmph->s_icmp_id)  
        {  
            return fp_list->next->OptTestFunc(p, otn, fp_list->next);  
        }  
        else  
        {  
            DEBUG_WRAP(DebugMessage(DEBUG_PLUGIN, "ICMP ID check  
failed\n"););  
        }  
    }  
    return 0;  
}
```

同样地，序列号的检测函数 IcmpSeqCheck 的实现与 ID 检测是一模一样的，读者可以参照分析。

## 12. 3. 2 代码选项

代码(icode)选项用于检测 ICMP 包的代码域的值。本选项通常有两种方法，一种是定义为某个特定含义的值，二是定义成一个非法值。后者对于发现欺骗、洪流、DoS 攻击有所帮助。源码分析如下（有删节）：

```
int IcmpCodeCheck(Packet *p, struct _OptTreeNode *otn, OptFpList *fp_list)
{
    .....

    switch(ds_ptr->operator)      /*判断操作符*/
    {
        case ICMP_CODE_TEST_EQ:    /*如果是等于*/
            /*比较规则中的参数值与 ICMP 包的 code 值*/
            if (ds_ptr->icmp_code == p->icmph->code)
                success = 1;          /*匹配成功*/
            break;

        case ICMP_CODE_TEST_GT:
            if (p->icmph->code > ds_ptr->icmp_code)
                success = 1;
            break;

        case ICMP_CODE_TEST_LT:
            if (p->icmph->code < ds_ptr->icmp_code)
                success = 1;
            break;

        case ICMP_CODE_TEST_RG:
            if (p->icmph->code > ds_ptr->icmp_code &&
                p->icmph->code < ds_ptr->icmp_code2)
                success = 1;
            break;

    }

    if (success)
    {
        /*若匹配成功，继续调用下一个处理函数，然后返回*/
    }
}
```

```
    return fp_list->next->OptTestFunc(p, otn, fp_list->next);
}
/*否则， 匹配失败*/
return 0;
}
```

类型选项插件的实现函数 IcmpTypeCheck，与代码选项完成相同，读者可以参照分析。

## 12. 4 实时响应

动态响应的选项 resp (sp\_respond.c) 允许 Snort 自动阻断触发规则的连接，它是目前 Snort 中最为强大的基于协议的规则体选项。使用动态响应修饰符的格式为：resp: 修饰符 1, 修饰符 2, 修饰符 3……。

下面是基于 TCP 的可用修饰符：

rst\_all: 阻断 TCP 连接的双方。  
rst\_recv: 阻断 TCP 连接的接收方。  
rst\_send: 阻断 TCP 连接的发送方。

ICMP 的可用修饰符包括：

icmp\_all: 阻断 ICMP 传输的双方。  
icmp\_host: 发送 ICMP 主机不可达到传输的客户端。  
icmp\_net: 发送 ICMP 网络不可达到传输的客户端。  
icmp\_port: 发送 ICMP 端口不可达到传输的客户端。

ParseResponse 函数用于配置参数的解析，并设置相应的标识符，我们可以根据这些修饰符，很容易地理解该函数。

Respond 函数根据当前节点中响应标识的设置情况，进行不同的响应操作。  
源码分析如下：

```
int Respond(Packet *p, RspFpList *fp_list)
```

```

{
    RespondData *rd = (RespondData *)fp_list->params;

    if(!p->iph)
    {
        return 0;
    }

    if(rd->response_flag)          /*设置了实时响应标志*/
    {
        if(rd->response_flag & (RESP_RST SND | RESP_RST RCV))
        {
            if(p->iph->ip_proto == IPPROTO_TCP && p->tcph != NULL)
            {
                if((p->tcph->th_flags & (TH_SYN | TH_RST)) != TH_RST)
                {
                    /*若设置了 rst_send 修饰符，则使用 SendTCP_RST，向发送方
                     主机发送终止连接请求的数据包*/
                    if(rd->response_flag & RESP_RST SND)
                    {
                        SendTCP_RST(p->iph->ip_dst.s_addr,
                                    p->iph->ip_src.s_addr,
                                    p->tcph->th_dport, p->tcph->th_sport,
                                    p->tcph->th_ack,
                                    htonl(ntohl(p->tcph->th_seq) + p->dsiz));
                    }
                    /*若设置了 rst_send 修饰符，则使用 SendTCP_RST，向发送方
                     主机发送终止连接请求的数据包*/
                    if(rd->response_flag & RESP_RST RCV)
                    {
                        SendTCP_RST(p->iph->ip_src.s_addr,
                                    p->iph->ip_dst.s_addr,
                                    p->tcph->th_sport, p->tcph->th_dport,
                                    p->tcph->th_seq,
                                    htonl(ntohl(p->tcph->th_ack) + p->dsiz));
                    }
                }
            }
        }
    }
}

/*需要解释的是：如果设置了 rst_all，在解析函数中会配置标识
 符 response_flag |= (RESP_RST SND | RESP_RST RCV)，所以会同时匹配以上两个 if*/

```

```

        }

    }

}

if((p->icmph == NULL) ||
   (p->icmph->type == ICMP_ECHO) ||
   (p->icmph->type == ICMP_TIMESTAMP) ||
   (p->icmph->type == ICMP_INFO_REQUEST) ||
   (p->icmph->type == ICMP_ADDRESS))
{
    /*发送网络不可到达*/
    if(rd->response_flag & RESP_BAD_NET)
        SendICMP_UNREACH(ICMP_UNREACH_NET,
p->iph->ip_dst.s_addr,
                p->iph->ip_src.s_addr, p);
    /*发送主机不可到达*/
    if(rd->response_flag & RESP_BAD_HOST)
        SendICMP_UNREACH(ICMP_UNREACH_HOST,
p->iph->ip_dst.s_addr,
                p->iph->ip_src.s_addr, p);
    /*发送端口不可到达*/
    if(rd->response_flag & RESP_BAD_PORT)
        SendICMP_UNREACH(ICMP_UNREACH_PORT,
p->iph->ip_dst.s_addr,
                p->iph->ip_src.s_addr, p);
}
return 1; /* always success */
}

```

可见，最后实际封包发送的功能是由 SendTCP\_RST 和 SendICMP\_UNREACH 完成的。这两个函数主要采用了 libpcap 的 libnet\_write\_ip 库函数来完成。关于 libpcap 发送封包的内容，读者可以参考前文相关介绍。

# 第13章 输出检测结果

## 13. 1 报警日志处理流程

Snort 有许多的输出插件，比较常用的是数据库插件。本节将就 Snort 是如何实现向 Database 输出报警日志的处理流程做一个简单的分析，而数据库插件的实现笔者将在下一节以最常用的 Mysql 做一个详细的分析。其它插件的实现大都类似，读者感兴趣可以阅读完本章后自行分析。

我们已经说过 Snort 除了进行报警日志的输出之外，也可以对整个数据包进行记录，就像是我们在第七章讲过的日志记录模式的输出一样。

如上文分析，函数 Preprocess(Packet \* p) 在检测引擎执行完成后，调用

```
retval = SnortEventqLog(p);
```

来执行输出插件。

函数 SnortEventLog (event\_queue.c) 的实现很简单：

```
int SnortEventqLog(Packet *p)
{
    if(sfeventq_action(LogSnortEvents, (void *)p) > 0)
        return 1;

    return 0;
}
```

SnortEventLog 先调用 sfeventq\_action 函数来判断是否需要进行日志事件的记录。而判断后如果需要进行日志处理，则是通过其第一个参数——函数 LogSnortEvents 来实现的。sfeventq\_action 具体实现如下：

```
int sfeventq_action(int (*action_func)(void *, void *), void *user)
{
    SF_EVENTQ_NODE *node;
```

```

int           logged = 0;

/*先判断如果实际处理的函数不存在的话，则退出*/
if(!action_func)          //指向函数的指针实际指向的是 LogSnortEvents

    return -1;

if(!(s_eventq.head))

    return 0;

for(node = s_eventq.head; node; node = node->next)

{

    if(logged >= s_eventq.log_nodes)

        return 1;

if(action_func(node->event, user))      //在这里实际执行

    return -1;

logged++;

}

return 1;
}

```

前文我们已经分析了，当前数据包所触发的事件，都存储在 `s_eventq.head` 为首的链表之中。全局变量 `s_eventq` 是一个 `struct s_SF_EVENTQ` 类型，其成员 `head` 是一个 `struct s_SF_EVENTQ_NODE` 类型的数据结构。该结构的一个成员 `void *event` 是一个空指针，在实际应用中指向的是一个 `struct _EventNode` 结构，该结构包含了一个事件对应的所有特征。所以，函数在确定了链表不为空的情况下，遍历该链表，调用 `action_func(node->event, user)` 转向。

`action_func` 实际指向的是 `LogSnortEvents` (`event_queue.c`) 函数，它是日志

处理的实际接口函数：

```
static int LogSnortEvents(void *event, void *user)

{
    Packet      *p;
    EventNode  *en;
    OTNX       *otnx;

    if(!event)
        return 0;

    en = (EventNode )event;           /*取出事件描述*/

    p  = (Packet *)user;

    /*
    **  Log rule events differently because we have to.
    */

    if(en->rule_info)
    {
        otnx = (OTNX *)en->rule_info;

        if(!otnx->rtn || !otnx->otn)
            return 0;

        fpLogEvent(otnx->rtn, otnx->otn, p);

    }
    else
    {
        GenerateSnortEvent(p, en->gid, en->sid, en->rev,
                            en->classification, en->priority, en->msg);
    }
}
```

```

sfthreshold_reset();

return 0;
}

```

函数在取得与事件相适的 otnx 后，转入 fpLogEvent 函数进行处理，如果 en->rule\_info 没有内容，则直接调用 GenerateSnortEvent 输出相关信息。

fpLogEvent 函数（fpdetect.c）在进行了一些前期处理与判断后，利用

```
switch(rtn->type)
```

根据匹配到的规则节点类型进行不同处理，如果是报警事件：

```
case RULE_ALERT:
```

```
    AlertAction(p, otn, &otn->event_data);
```

则调用 AlertAction 函数，而 AlertAction 函数则分别调用 CallAlertFuncs、CallLogFuncs 来处理报警和日志，实现如下：

```

int AlertAction(Packet * p, OptTreeNode * otn, Event *event)

{
    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,
                           "          <!> Generating alert! \"%s\"\n", otn->sigInfo.message));

    /* Call OptTreeNode specific output functions */

    if(otn->outputFuncs)
        CallSigOutputFuncs(p, otn, event);

CallAlertFuncs(p, otn->sigInfo.message, otn->rtn->listhead, event);

    DEBUG_WRAP(DebugMessage(DEBUG_DETECT, "      => Finishing alert
packet!\n"));

CallLogFuncs(p, otn->sigInfo.message, otn->rtn->listhead, event);

    DEBUG_WRAP(DebugMessage(DEBUG_DETECT,"      => Alert packet finished,
returning!\n"));

    return 1;
}

```

函数 CallAlertFuncs 遍历所有注册的报警插件，以实现报警事件的输出：

```
while(idx != NULL)          //遍历输出插件链表
{
    idx->func(p, message, idx->arg, event);
    idx = idx->next;
}
```

这里的四个参数：

p：待处理的数据包；

message：报警的摘要信息；

idx->arg：输出插件参数；

event：该事件的一些信息的结构，包含了事件编号、参考信息编号等等；

下一节将就 Snort 使用得最多的 Database 插件，向 Mysql 数据库输出日志的实现进一步分析。

## 14. 2 向 MySQL 输出报警日志的实现