Programming Guide

eyoung.father@gmail.com

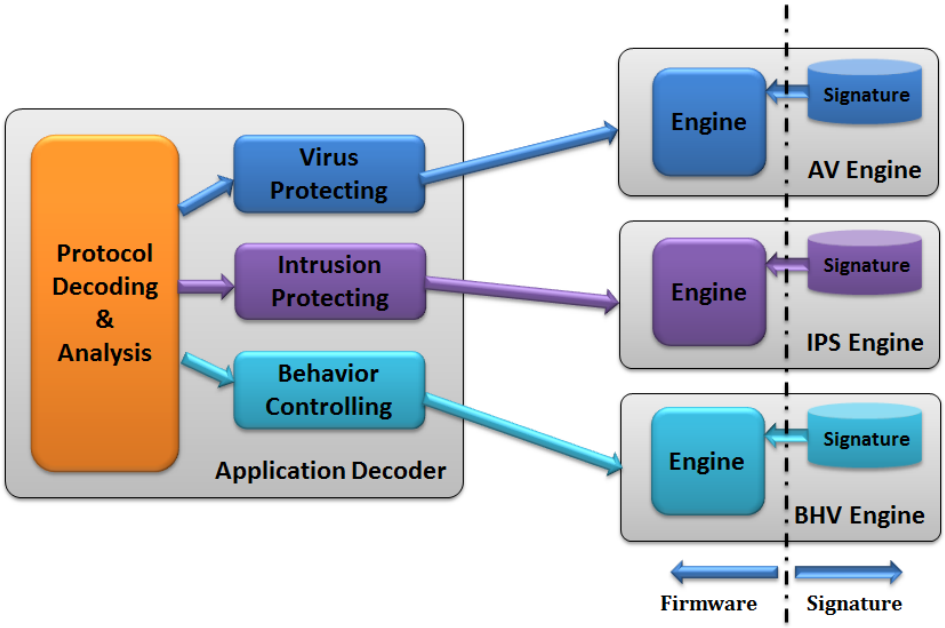
2014/3/21

目录

1. eyoung 介绍

1.1. 为什么设计 eyoung

下图是常见应用层处理架构：



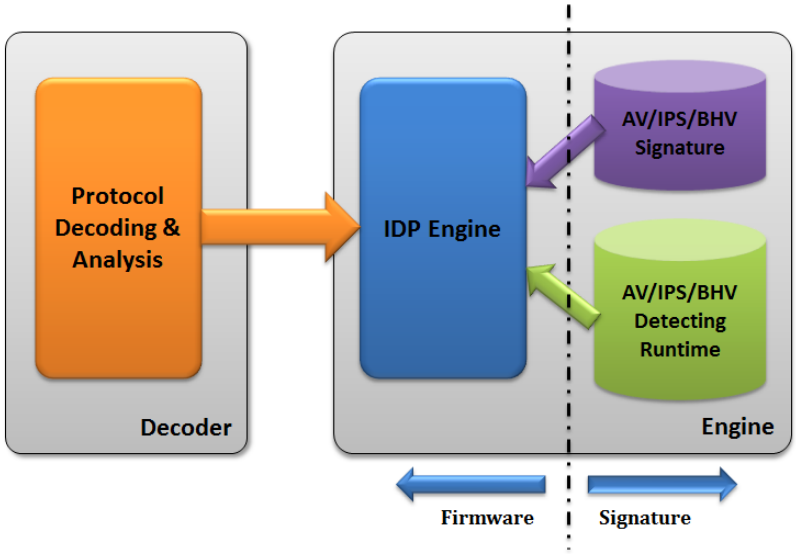
图中的虚线是 Firmware（软件固件）与 Signature（规则）的分界线。Firmware 的特点是相对稳定不会频繁变化，其变换周期可能是以月计算的；Signature 是通常意义的 IPS 规则、病毒库等会频繁更新的内容，它们的更新频率通常以天或小时计算。Firmware 部分通常由应用层协议分析器（Application Decoder）和引擎（Engine）组成，Application Decoder 负责进行协议分析和一部分内容检测工作，Engine 的主要作用是匹配 Signature。以 Snort 为代表的开源 IPS 系统，它们的 IPS Signature 主要是以字符串匹配、正则表达式匹配为核心描述手段的。这种描述手段的特点是简单、适合硬件加速，但是它的弱点也是非常明显，那就是描述能力的不足：

首先，许多攻击是无法使用正则表达式描述的，最典型的就是缓冲区溢出攻击。很明显，使用正则表达式无法涵盖所有的机器指令的组合，后果是误报和漏报都非常严重。另外，分布式拒绝服务供给（DDoS）也是一种典型的无法用正则表达式描述的攻击。

其次，正则表达式描述能力不够精确、全面，例如 SQL 注入攻击。正则表达式能够描述一个小的 SQL 片段，但是对于本质上是上下文无关语言的 SQL 语言，这种做法是不精确的，其结果必然会同时带来误报和漏报。

最后，正则表达式无法描述业务的流程。Layer7 的数据变化很快，例如网页结构、表单结构、参数编码等等，这些都是跟具体的业务相关联的，而通用性的攻击防护规则，无法描述这些业务层的变化。为了能够描述业务的特点，在传统架构下只能将业务数据（例如 WebMail）的解析放到 Application Decoder 中——这就带来了 **Firmware 一侧设计的冗余和不通用**，系统性能也会受拖累。由于 Firmware 的发布通常意味着大软件版本的发布，是很慢的，所以在快速变化的移动互联网领域，**Firmware 的变化速度会严重滞后于业务的变化**。

为了解决上述攻击可描述性、业务可描述性问题，eyoung 将传统的 Layer7 处理架构调整为：



eyoung 系统主要是指图中 Firmware 部分，包含两个大的模块：协议分析器（Decoder）和 IPS 引擎（Engine）。其中，eyoung Decoder 是一个单纯的 Layer7 协议分析器。与传统系统中的 Application Decoder 不同的是，eyoung Decoder 不需要理解业务的语义，而仅仅根据 RFC 的定义完成 Layer7 协议数据的解析即可。eyoung Engine 是一个事件分析引擎，它负责加载 eyoung 规则，并根据 eyoung Decoder 协议分析的结果进行事件分析，既可以做深层次的协议分析和协议控制，也可以做攻击检测，所以这里 eyoung Engine 也被称为 eyoung IPS 引擎。eyoung IPS 引擎的核心特性是支持**规则的可编程性和业务的可描述性**：

首先，eyoung 规则本身可编程。eyoung 支持 C99 兼容的语法，即规则开发者可以使用 C99 兼容的语法将复杂的分析描述代码、攻击检测代码嵌入到规则文件中。

其次，eyoung IPS 引擎内嵌了 JIT 运行时编译器，可以将内嵌的 C 代码在运行时编译成机器指令，并动态链接到当前的进程镜像，从而极大地提高执行效率和检测效率。

最后，eyoung 规则支持二进制库的动态导入，例如可以将 WebMail 解析这样复杂的业务描述程序，以动态链接库的形式动态加载到 eyoung IPS 引擎中，做到单独加载、单独升级。这样做一方面简化了 Layer7 Decoder 的实现，**真正实现了协议分析和业务分析的分离，另一方面扩展了规则的描述能力，加快了对**

Layer7 业务变化的响应速度。

eyoung 在设计上，实际上将 Firmware 的部分尽可能地固定下来，将快速变化的部分都推送到 Signature 一侧，以实现更强大的描述能力、扩展能力和分析能力。

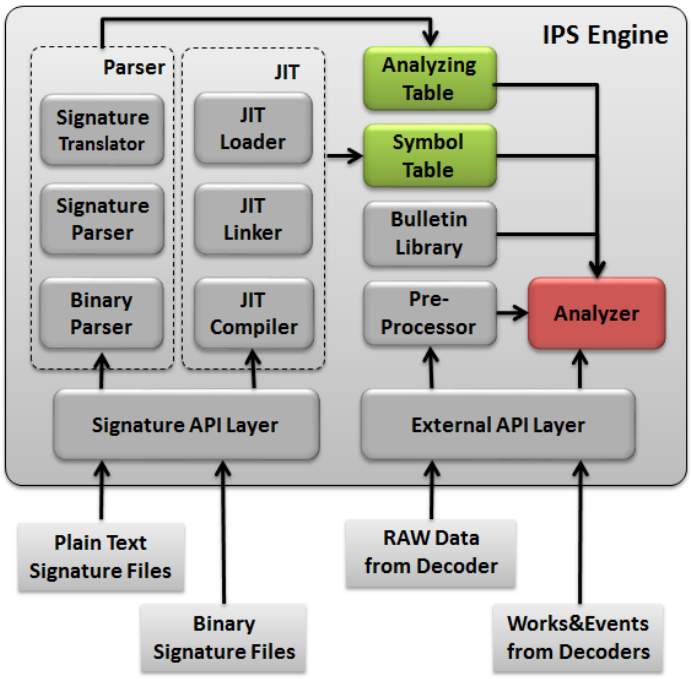
从软件模块的角度看待 eyoung 系统，eyoung 主要分为 IPS 引擎和协议分析器两大功能模块，这两个模块间是严格地单向调用的关系，即协议分析器调用 IPS 引擎的相关功能，它们之间有清晰的 API 函数接口，这个 API 函数集被称为 **External API Layer**。

从运行阶段的角度看待 eyoung 系统，eyoung 运行时主要分为静态加载阶段和动态检测阶段：

- 静态加载阶段，相当于一台计算机加电启动时操作系统的加载，主要是由 eyoung IPS 引擎加载相关的规则文件、二进制规则库等静态文件内容。eyoung IPS 规则支持高度的可编程性，IPS 规则中可以调用 eyoung IPS 引擎的相关功能，这是通过 eyoung IPS 引擎提供的 API 函数接口实现。eyoung IPS 引擎中的起这个作用的 API 函数集被称为 **Signature API Layer**。
- 动态检测阶段，主要进行攻击检测，即协议分析器接受网络数据并进行协议分析，并把协议分析的结果以结构化的方式，使用 External API Layer 相关的 API 提交给 eyoung IPS 引擎进行业务分析和攻击检查。

1.2. IPS 引擎的设计

1.2.1. 模块结构

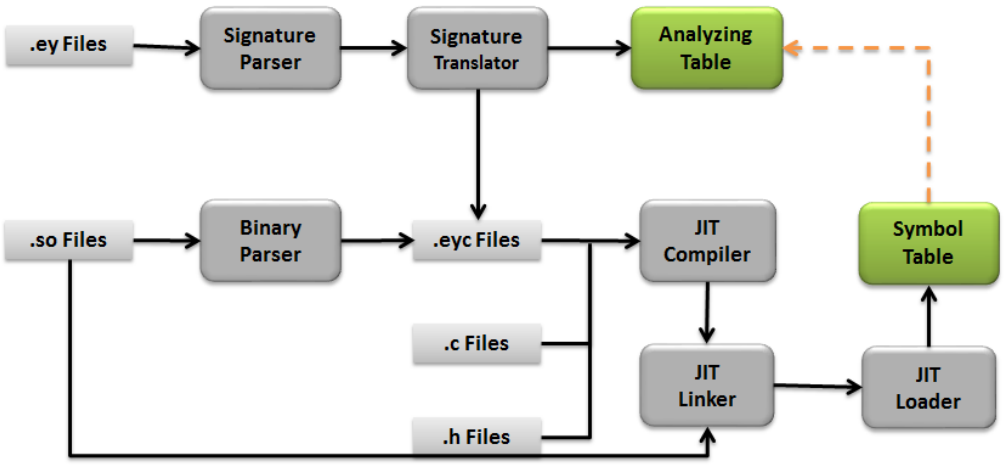


IPS 引擎从结构上主要分为：

1， IPS 规则解析器（Parser），它主要负责解析规则文件（Plain Text 规则、二进制规则库等）。它主要完成：
   a） 规则文件书写的合法性检查，探测并报告语法错误
   b） 对规则文件.ey 进行预处理，转换成.eyc 中间代码文件。eyoung IPS 规则的中间代码文件是 C99 兼容格式的 C 代码源文件。
   c） 对规则文件中表达的元数据，例如事件的定义、注册的函数接口、规则的表示，规则解析器将它们转换成引擎内部的数据格式，保存在符号表（Symbol Table）和分析表（Analyzing Table）中。
2， IPS 规则运行时编译器（JIT），它的作用是把经过 Parser 预处理之后的.eyc 中间代码文件进行运行时刻编译、链接和加载，同时将规则文件中的全局函数入口地址、全局变量地址保存到符号表中。
3， IPS 引擎函数库（Bulletin Library），主要提供基础的处理功能，主要包括内存管理、基础数据结构封装等等。
4， 预处理器（Pre-Processor）。预处理器是一个固定的接口层，通过对接口层不同的实现，可以将异构、异步的 IPS 引擎和规则集成到 eyoung 系统中。预处理器通常不对协议分析器的输出进行处理，它的处理对象通常是未进行应用层协议分析的原始数据（RAW Data）。

5， 分析器（Analyzer）。分析器负责对协议分析结果进行检测和分析，是运行时刻整个 eyoung IPS 引擎的核心。

1.2.2. 规则加载流程



1， 对于 Plain Text 的规则文件：

a) 规则解析器（Signature Parser）进行规则合法性检查

b) 规则翻译器（Signature Translator）进行规则翻译：

- 将规则翻译成.eyc 中间代码文件
- 规则翻译器还把规则文件中的事件定义、规则语义，转换成分析表（Analyzing Table）中的表项。
- 将规则文件中记录的 file、work、event 等元素的构造和析构函数入口，记录到分析表中。相关的详细说明在本文档的 2.3 中介绍。

2， 对于二进制规则.so 动态链接库文件：

a) 二进制文件解析器（Binary Parser）负责读取动态链接库.so 文件中的.eyoung_ident 和.eyoung_type 段（如果存在的话）的内容，将其中的内容也保存在.eyc 中间代码文件中。

b) 二进制文件解析器解析.eyoung_init 段内容，并执行.eyoung_init 段中包含的构造函数入口地址，完成二进制规则加载（Attach）后的初始化工作。

c) 二进制文件解析器还负责解析.eyoung_finit 段的内容，并记录下该段中保存的析构函数入口地址，用于在二进制规则库卸载（Detach）时释放相关资源。

3， 经过 1 和 2，所有的规则文件都转换成.eyc 中间代码文件，而后运行时编译器 JIT 接管规则加载的后续操作：

a) 运行时编译器（JIT Compiler）负责编译.eyc 中间代码文件以及其它附加的.h 头文件和.c 源文件。这个运行时编译操作都是在内存中完成的。

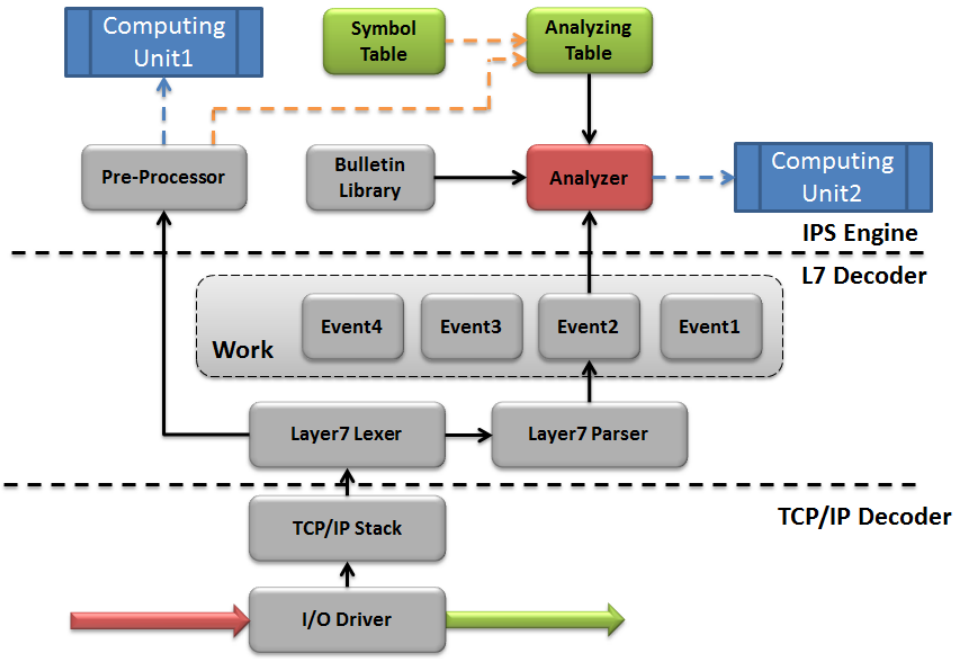
b) 运行时链接器（JIT Linker）负责将 JIT Compiler 编译好的文件单元在内存中进行链接。通过 JIT Linker 的处理，不同规则文件中的全局符号（全局变量、全局函数）完成重定位和相互引用。

c) 运行时加载器（JIT Loader）将链接后的代码加载到当前程序的运行环境中，同时将本进程中的全局符号与中间代码文件编译链接后的代码再次链接。这个过程可以使得规则文件中的代码直接调用当前进程中开放的函数接口和全局变量。这种机制增加 Firmware 和 Signature 的交互，提升规则的描述能力。例如，某些复杂耗时的协议分析功能可能仅仅是在特定条件下需要完成，没有必要无差别地调用。此时，可以把这种协议分析的功能以全局函数的形式实现在协议分析器中，但协议分析器并不调用，而是在 IPS 规则中在一定条件下（例如只针对配置的 URI）调用该函数。

d) 加载后的全局符号及其地址都保存在符号表（Symbol Table）中。

1.3. 协议分析器的设计

1.3.1. 协议分析器及攻击检测



1， TCP/IP Decoder 是低层协议栈，负责链路层、网络层、传输层数据的解析、重组和代理等工作。eyoung 系统不过多地考虑这个问题，它们的通常是操作系统内核的功能，或是类似 Intel DPDK 等用户态的数据转发架构。

2， 应用层协议分析器（Layer7 Decoder），用来分析应用层数据。分两部分：

a) 应用层数据词法分析器（Layer7 Lexer），与低层协议栈交互并维护应用层数据缓冲区。它将应用层数据按照词法规则分解成一系列应用层数据"词

元”，并将这些词元提交给 Layer7 Parser 用于进一步协议分析；同时，应用层数据词法分析器还将低层获得的数据，通过 eyoung IPS 引擎的 External API Layer 提交给 eyoung IPS 引擎的 Pre-Processor 进行第三方规则的预处理。应用层数据词法分析器采用 Stream-Mode 进行解析，具体方法本文 3.1 节详细介绍。

b) 应用层数据语法分析器（Layer7 Parser），将接收到的应用层数据“词元”序列，按照应用层语法进行“自底向上”的语法分析，从逻辑上对应用层数据进行解释，并将语法分析的结果以 work、event 等抽象数据形式，提交给 eyoung IPS 引擎。Work、event 等概念在本文 2.1 节、3.3 节中详细介绍。应用层数据语法分析器采用 Push-Mode 方式，具体方法在本文 3.2 一节介绍。

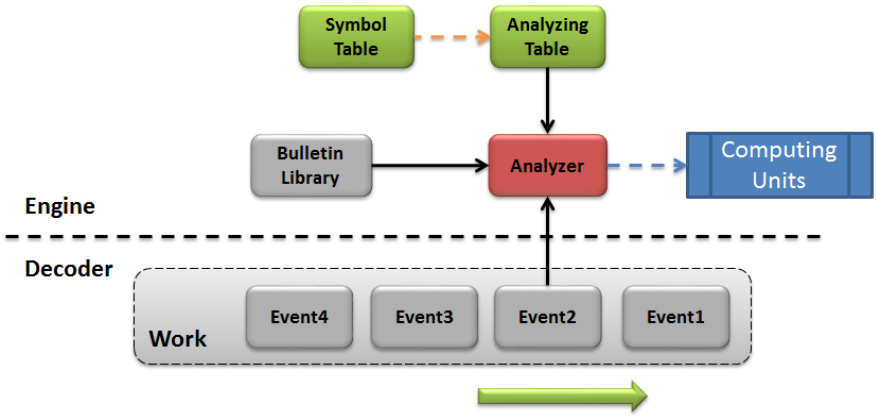
3， 预处理器（Pre-Processor），它可以兼容第三方检测引擎和第三方规则格式。eyoung 期望预处理器可以异步地（目前是同步方式）在其它的计算单元上执行，例如其它 CPU、ASIC 芯片或 GPU 等等。预处理器的处理结果，将被记录在分析表（Analyzing Table）中，并被分析器（Analyzer）利用。

4， 分析器（Analyzer），它是 eyoung IPS 引擎在检测阶段的核心部件。它结合 eyoung IPS 规则、分析表的数据、预处理器等信息，对应用层协议分析器提交的 work、event 对象序列进行分析和检测。

2. IPS 引擎

2.1. 主要数据结构

eyoung IPS 引擎的核心数据结构定义在 include/libengine_type.h 头文件中。在概念上，该文件中定义的数据结构主要有三个：engine_t、engine_work_t 和 engine_work_event_t。其中，engine_t 是一个静态的概念，表示 eyoung IPS 引擎本身，我们简称为 Engine；engine_work_t 和 engine_work_event_t 都属于动态概念，用于表示被检测的对象。engine_work_t 也被简称为 work，engine_work_event_t 也被简称为 event。



2.1.1. 源代码

```
#include "ey_queue.h"

typedef void* engine_t;
typedef int (*file_init_handle)(engine_t eng);
typedef int (*file_finit_handle)(engine_t eng);

typedef struct engine_work
{
    TAILQ_ENTRY(engine_work) link;
    engine_t engine;
    void *priv_data;          /*for libengine itself*/
    void *predefined;         /*for protocol parser*/
    void *user_defined;       /*for signature writer*/
}engine_work_t;
typedef TAILQ_HEAD(engine_work_list, engine_work) engine_work_list_t;
typedef int (*work_init_handle)(engine_work_t *work);
typedef int (*work_finit_handle)(engine_work_t *work);

typedef struct engine_work_event
{
    engine_work_t *work;
    void *event;
    void *predefined;         /*for protocol parser*/
    void *user_defined;       /*for signature writer*/
    engine_action_t *action; /*OUTPUT*/
}engine_work_event_t;
typedef int (*event_init_handle)(engine_work_event_t *work_event);
typedef int (*event_finit_handle)(engine_work_event_t *work_event);
typedef int (*event_preprocess_handle)(engine_work_event_t
        *work_event);
typedef int (*event_condition_handle)(engine_work_t *engine_work,
      engine_work_event_t *work_event);
typedef int (*event_action_handle)(engine_work_t *engine_work,
      engine_work_event_t *work_event);
```

2.1.2. 说明

1，engine_t

eyoung IPS 引擎实例对外的类型抽象，eyoung IPS 引擎之外的模块需要操作 IPS 引擎时，例如加载规则文件、卸载规则文件、注册事件等，相关 API 都需要提供 engine_t 实例。

2，file_init_handle、file_finit_handle

函数指针类型，其对应的函数实例需要使用 2.3.1 中定义的 ey_add_file_init、ey_add_file_finit 两个宏向 engine 中注册，也可以通过规则文件中的%file-init 和%file-finit 两个关键字选项在规则文件中向 engine 注册。

eyoung IPS 引擎可以加载多个.ey 规则文件，每一个规则文件中都可以定义**至多一个** file_init_handle 和**至多一个** file_finit_handle 函数。对于 file_init_handle 函数，其调用的时机是在所有规则文件都加载完毕后，由 eyoung IPS 引擎按照规则文件加载的顺序依次调用，用来完成文件级的初始化工作；file_finit_handle 函数，是在 eyoung IPS 引擎要卸载规则时，按照当初的规则文件加载的顺序调用，用来完成文件级的资源释放工作。

3，engine_work_t

简称为 work，是一个运行时的动态概念。work 是由一系列 event 组成，用来抽象表示一个 IPS 检测过程，例如一条 TCP 连接可以被视为一个 work。

- work 通过 link 成员组成一个链表，该链表记录在 engine_t 对象中
- engine 成员用来指向本 work 隶属的 engine_t 对象
- priv_data 成员，用来指向 work 在 engine 中的数据对象，该成员不允许外部修改其内容
- predefined 成员，指向一个由协议分析器开发者定义的内存对象。因为 work 对象是由协议分析开发者创建，是一个"连接"级概念，所以如果协议开发人员期望记录一部分"连接"级的数据（例如一条 TCP 连接上的计数信息），就可以通过 work 中的 predefined 成员来实现。predefined 具体的格式由协议开发人员定义和解释，engine 对象不能理解其结构。
- user_defined 成员，指向一个由规则开发者定义的内存对象。由于应用层协议分析开发和检测规则开发是分离的,所以在协议解析开发阶段不可能预期所有的检测需求。这就需要有一部分"连接"级的信息需要在规则开发阶段，由规则开发人员自行定义并保存。user_defined 就是为了满足这样的需要，其格式由规则开发人员自己定义和解释，eyoung IPS 引擎不能理解其结构。

4，work_init_handle、work_finit_handle

是两个函数指针类型，其对应的函数实例在 work 创建、销毁的时候被 eyoung IPS 引擎对象调用。其主要用来分配、释放 predefined 和 user_defined 的数据。这些函数的实例由协议分析开发人员和规则开发人员

定义，并通过 2.3.1 中的相关 API 注册到 eyoung IPS 引擎对象中。具体地，work 中 predefined 成员的创建和销毁功能的函数实例，由协议开发人员通过ey_set_predefine_work_init 和ey_set_predefine_work_finit 两个宏注册到 eyoung IPS 引擎对象中；work 中 user_defined 成员的创建和销毁功能的函数实例，由规则开发人员通过两个宏 ey_set_userdefine_work_init、ey_set_userdefine_work_finit 向 eyoung IPS 引擎对象注册，或者在规则文件中通过使用%work-init 和%work-finit 两个关键字选项向 eyoung IPS 引擎对象注册。

5，engine_work_event_t

简称为 event，属于运行时的动态概念，它是对检测元素的数据类型抽象。一个 event 既可以是一个实实在在的应用层协议成员，例如 HTTP URI，也可以是一个抽象的事件，例如一次非法的 DNS 请求，它的设计由协议分析开发人员定义。event 的创建和销毁都由协议分析开发人员决定，目前不支持规则开发人员在规则中创建和销毁 event。

- work 成员，用来指向 event 所隶属的 work 对象。
- event 成员，用来指向 event 定义时的类型对象。该类型对象，属于一个静态的概念，是在规则加载时由规则中的%event 关键字创建的。该成员的解释、使用都由 engine 对象完成，协议分析开发人员和规则开发人员不能修改该成员。
- predefined 成员，指向一个由协议分析器开发者定义的内存对象。如同 work 对象中的 predefined 成员一样，这里 predefined 成员也是由协议开发人员自行定义和解释，eyoung IPS 引擎对象不能理解其结构。
- user_defined 成员，指向一个由规则开发者定义的内存对象。与 work 对象的 user_defined 成员一样，该成员的设立是为了对协议分析的结果进行补充。例如，对于 HTTP URI，在经过协议分析之后被简单解释成字符串之后，但是规则中可能需要对 URI 中某一个特定名称的参数进行过滤。URI 中的参数可以被解释成一个“键-值”对数组，如果一开始在协议分析阶段就完成“键-值”对的分析，很可能会无谓地浪费性能。此时，可以在规则匹配阶段，针对特殊的 URI 单独做参数的解析。此时，原始的 URI 字符串保存在 event 的 predefined 成员中；而进一步分析出来的“键-值”对则被保存在 event 的 user-defined 成员中。

6，event_init_handle、event_finit_handle、event_preprocess_handle

三个函数指针类型，其对应的函数实例在 event 创建、销毁、检测的时候被 eyoung IPS 引擎对象调用。其主要用来分配、释放、处理 event 中的 predefined 和 user_defined 数据。这些函数的实例分别由协议分析开发人员和规则开发人员定义，并通过 2.3.1 中的相关 API 注册到 eyoung IPS 引擎中。具体地，event 对象中 predefined 成员的创建、销毁、处理对应的函数实例，由应用层协议分析器的开发人员通过使用 2.3.1 中定义的 ey_set_predefine_event_init，ey_set_predefine_event_finit，ey_set_predefine_event_preprocess 三个宏注册到 eyoung IPS 引擎对象中；event 中 user_defined 成员的创建、销毁、处理功能的函数实例，

由规则开发人员通过使用三个宏 ey_set_userdefine_work_init、ey_set_userdefine_work_finit、ey_set_userdefine_event_preprocessor 向 eyoung IPS 引擎对象注册，或在规则文件中通过%event-init、%event-finit、%event-preprocessor 三个关键字选项向 eyoung IPS 引擎对象注册。

7，event_condition_handle、event_action_handle

这是两个由 engine 内部使用的函数指针类型，分别对应于规则中 condition 和 action 两部分转换之后的函数类型。可以参考《Signatures Specification》中 3.2 一节的叙述。

2.2. External API

定义在 include/libengine_function.h 头文件中。

2.2.1. 源代码

```
#include "libengine_type.h"
#include "libengine_export.h"
extern engine_t ey_engine_create(const char *name);

extern void ey_engine_destroy(engine_t engine);

extern int ey_engine_load(engine_t engine,
      char *files[], int files_num);

extern int ey_engine_find_event(engine_t engine,
      const char *event_name);

extern engine_work_t *ey_engine_work_create(engine_t engine);

extern void ey_engine_work_destroy(engine_work_t *work);

extern engine_work_event_t *ey_engine_work_create_event(
      engine_work_t *work,
      unsigned long event_id,
      engine_action_t *action);

extern int ey_engine_work_detect_data(engine_work_t *work,
      const char *data,
      size_t data_len,
      int from_client);

extern int ey_engine_work_detect_event(
      engine_work_event_t *event,
      void *predefined);

extern void ey_engine_work_destroy_event(engine_work_event_t *event);
```

```
extern int debug_engine_parser;
extern int debug_engine_lexier;
extern int debug_engine_init;
extern int debug_engine_compiler;
extern int debug_engine_runtime;
```

2.2.2. 说明

1. ey_engine_create

用途：

该函数用于创建一个 eyoung IPS 引擎。

参数：

name：是要创建的 IPS Engine 实例的名称字符串，以'\0'结尾。系统不检查名称是否重名。

返回值：

创建成功则返回 engine_t 类型的指针，否则返回 NULL。

示例：

参见 2.5.1 节

2. ey_engine_destroy

用途：

该函数用于销毁一个 eyoung IPS 引擎实例。

参数：

engine：ey_engine_create 的返回值。

返回值：

该函数没有返回值。

示例：

参见 2.5.1 节

3. ey_engine_load

用途：

eyoung IPS 引擎实例创建之后，必须通过该函数加载相关 IPS 规则文件。

参数：

engine：ey_engine_create 的返回值。

files：字符指针数组，用来记录需要加载的规则文件路径。

file_num：指示 files 数组的长度。

返回值：

成功返回 0；否则返回-1。

示例：

参见 2.5.1 节

4. ey_engine_find_event

用途：

该函数通常由协议分析器调用，用来根据“事件名称”，查找“事件索引值”。在 eyoung 中，“事件”的声明是通过规则文件，在规则加载的时候完成，具体的事件定义格式参照《Signatures Specification》的 2.11 一节。事件有名字，在一个 eyoung IPS 引擎实例中，事件的名称不允许重复。在 eyoung 系统中，规则、IPS 引擎、协议分析器都是通过事件名称的约定来完成数据的传递。为了方便传递，eyoung IPS 引擎使用事件的索引值来代替事件名称进行信息传递。ey_engine_find_event 是通过“事件名称”查找“事件索引值”的 API。

参数：

engine：ey_engine_create 的返回值。

event_name：事件名称字符串，以’\0’结尾。

返回值：

成功则返回非负值，表示对应的事件索引值；失败则返回-1。

示例：

libdecoder/http/http_server.y 中：

```
void http_server_register(http_decoder_t *decoder)
{
    assert(decoder!=NULL);
    assert(decoder->engine!=NULL);

    engine_t engine = decoder->engine;
    int index = 0;
    for(index=0; index<YYNNTS; index++)
    {
        const char *name = yytname[YYNTOKENS + index];
        if(!name || name[0]=='$' || name[0]=='@')
            continue;
        yytid[YYNTOKENS + index] =
                    ey_engine_find_event(engine, name);
        if(yytid[YYNTOKENS + index] >= 0)
            http_debug(debug_http_server_parser,
                "server event id of %s is %d\n",
                name, yytid[YYNTOKENS + index]);
        else
            http_debug(debug_http_server_parser,
                "failed to register server event %s\n", name);
    }
}
/*bison 所生成的语法分析代码中，yytname 数组记录着所有符号的名称，
YYNTOKENS 宏记录着非终结符的起始 ID。在 eyoung 中，协议分析器在“移动-
规约”的语法分析过程中，将归约出来的非终结符依次提交给 IPS 引擎检测。所
以，协议分析器中的非终结符，实际就是 IPS 引擎中的“事件”，所有的非终结
符都需要在 IPS 规则中书写，以使 IPS 引擎感知到“事件”的存在。该函数在规
则加载之后，HTTP 协议分析器初始化的时候被调用，通过依次对所有非终结符
调用 ey_engine_find_event 查找 IPS 引擎中的事件索引值，并保存在 yytid 数组
中。*/
```

5. ey_engine_work_create

用途：

创建一个 work 实例,创建过程中，通过 ey_set_predefine_work_init、ey_set_userdefine_work_init、%work-init 注册的初始化函数，会被 eyoung IPS 引擎调用以完成相关的初始化工作。

参数：

engine：ey_engine_create 的返回值

返回值：

成功则返回一个 engine_work_t 的实例指针；否则返回 NULL。

示例：

参见 2.5.2 节。

6. ey_engine_work_destroy

用途：

销毁一个 work 实例。在 work 实例销毁的过程中，通过 ey_set_predefine_work_finit 、 ey_set_userdefine_work_finit 和%work-finit 注册的析构函数，会被 eyoung IPS 引擎调用以完成相关的资源释放工作。

参数：

work：ey_engine_work_create 的返回值

返回值：

无。

示例：

参见 2.5.2 节。

7. ey_engine_work_create_event

用途：

用来创建一个 event 实例。在 event 实例创建的过程中，通过 ey_set_predefine_event_init、ey_set_userdefine_event_init 和%event-init 注册的初始化函数，会被 eyoung IPS 引擎调用以完成相关的初始化工作。

参数：

work：ey_engine_work_create 的返回值

event_id：由 ey_engine_find_event 返回的合法的事件索引值。

action：输出参数，用来描述事件的检测结果。

返回值：

成功则返回 engine_work_event_t 的实例指针；否则返回 NULL。

示例：

libdecoder/http/decode/http_decode.c

```
int http_element_detect(http_data_t *http_data,
      const char *event_name, int event_id, void *event,
   char *cluster_buffer, size_t cluster_buffer_len)
```

```
{
    ……
    engine_action_t action = {ENGINE_ACTION_PASS};
    engine_work_t *work = http_data->engine_work;
    engine_work_event_t *work_event =
        ey_engine_work_create_event(work, event_id, &action);
    if(!work_event)
    {
        http_debug(debug_http_detect,
            "create event for %s failed\n", event_name);
        return 0;
    }
    ey_engine_work_detect_event(work_event, event);
    ey_engine_work_destroy_event(work_event);
    http_debug(debug_http_detect,
        "detect %s[%d], get actoin %s\n",
        event_name, event_id,
        ey_engine_action_name(action.action));
    if(action.action==ENGINE_ACTION_PASS)
        return 0;
    return -1;
}
```

/ 该函数在 HTTP 协议分析语法分析器中被调用，调用时机是语法分析器在规则加载时被自动调用，相关调用方式参照 3.2*/*

8. ey_engine_work_destroy_event

用途：

用来销毁一个 event 实例。在 event 实例销毁的过程中，通过 ey_set_predefine_event_finit、ey_set_userdefine_event_finit 和%event-finit 注册的析构函数，会被 eyoung IPS 引擎调用以完成相关的资源释放工作。

参数：

event：ey_engine_work_create_event 的返回值

返回值：

无

示例：

参考 ey_engine_work_create_event

9. ey_engine_work_detect_event

用途：

将一个创建的 event 实例，提交给 IPS 引擎进行分析和检测。通过 ey_set_predefine_event_preprocess、%event-preprocessor 和 ey_set_userdefine_event_preprocess 注册的事件预处理函数，会被 eyoung IPS 引擎调用以完成事件数据的处理工作。

参数：

event：ey_engine_work_create_event 的返回值

predefined：event 的初始值。

返回值：

0 表示事件是干净的，-1 表示事件是包含攻击数据的。

示例：

参考 ey_engine_work_create_event

10. ey_engine_work_detect_data

用途：

不同于 ey_engine_work_detect_event，ey_engine_work_detect_data 函数提交的数据参与的是 IPS 引擎预处理器的检查。预处理器可以被视为与 eyoung IPS 引擎平行的检测单元。

参数：

work：ey_engine_work_create 的返回值

data：要检测的缓冲区起始地址

data_len：要检测的缓冲区长度

from_client：缓冲区是否来自客户端一侧的连接

返回值：

目前固定返回 0

示例：

libdecoder/http/decode/http_decode.c

```
int http_decode_data(http_work_t work,
      const char *data, size_t data_len,
      int from_client, int last_frag)
{
    assert(work != NULL);
    http_data_t *http_data = (http_data_t*)work;
    if(http_data->engine_work)
        ey_engine_work_detect_data(http_data->engine_work,
            data, data_len, from_client);
    if(from_client)
        return parse_http_client_stream(http_data,
            data, data_len, last_frag);
    else
        return parse_http_server_stream(http_data,
            data, data_len, last_frag);
}
```

*/*http_decode_data 函数是在 Socket 上收到 HTTP 数据之后首先被调用的。该函数先调用 ey_engine_work_detect_data 函数，用来完成相关 eyoung IPS 引擎预处理器的检查，而后才调用 parse_http_client_stream 和 parse_http_server_stream 完成 HTTP 的协议分析工作和 eyoung IPS 规则匹配工作。*/*

11. debug_engine_init

这是一个 debug 开关，用来控制 eyoung IPS 引擎初始化阶段的信息，包括 ey_engine_create、ey_engine_destroy 和 ey_engine_find_event 的 debug 信息。

12. debug_engine_lexier

这个 debug 开关用来控制 eyoung IPS 引擎加载规则文件时，eyoung IPS

引擎规则词法分析器的 debug 信息。这些 debug 信息位于 API ey_engine_load 函数中。

13. debug_engine_parser

这个 debug 开关用来控制 eyoung IPS 引擎加载规则文件时，eyoung IPS 引擎规则语法分析器的 debug 信息。这些 debug 信息位于 API ey_engine_load 函数中。

14. debug_engine_compiler

这个 debug 开关用来控制 eyoung IPS 引擎加载规则文件时，JIT 编译器、预处理器的 debug 信息。这些 debug 信息位于 API ey_engine_load 函数中。

15. debug_engine_runtime

这个 debug 开关用来控制 eyoung IPS 引擎在进行攻击检测阶段的 debug 信息。包括 ey_engine_work_create、ey_engine_work_create_event、ey_engine_work_detect_data 、 ey_engine_work_detect_event 、ey_engine_work_destroy_event、ey_engine_work_destroy 中的 debug 信息。

2.3. Signature API

eyoung signature 中可以直接调用的 IPS Engine 的功能函数 API，都定义在头文件 include/libengine_export.h 中。Signature API 主要是为了方便规则开发人员，注册 file、work、event 等对象的创建、销毁、处理等动作。

2.3.1. 源代码

```
#include "libengine_type.h"
#define ey_add_file_init(eng, func)                          \
    _ey_add_file_init(eng,#func,func,__FILE__,__LINE__)
#define ey_add_file_finit(eng, func)                         \
    _ey_add_file_finit(eng,#func,func,__FILE__,__LINE__)
extern int _ey_add_file_init(engine_t engine,
    const char *function, file_init_handle address,
    const char *filename, int line);
extern int _ey_add_file_finit(engine_t engine,
    const char *function, file_finit_handle address,
    const char *filename, int line);

#define ey_set_userdefine_work_init(eng,func)                \
    _ey_set_work_init(eng,1,#func,func,__FILE__,__LINE__)
#define ey_set_predefine_work_init(eng,func)                 \
    _ey_set_work_init(eng,0,#func,func,__FILE__,__LINE__)
extern int _ey_set_work_init(engine_t engine, int type,
      const char *function, work_init_handle address,
```

```
        const char *filename, int line);

#define ey_set_userdefine_work_finit(eng,func)          \
    _ey_set_work_finit(eng,1,#func,func,__FILE__,__LINE__)
#define ey_set_predefine_work_finit(eng,func)           \
    _ey_set_work_finit(eng,0,#func,func,__FILE__,__LINE__)
extern int _ey_set_work_finit(engine_t engine, int type,
        const char *function, work_finit_handle address,
        const char *filename, int line);

#define ey_set_userdefine_event_init(eng,ev,func)       \
    _ey_set_event_init(eng,#ev,1,#func,func,__FILE__,__LINE__)
#define ey_set_predefine_event_init(eng,ev,func)        \
    _ey_set_event_init(eng,#ev,0,#func,func,__FILE__,__LINE__)
extern int _ey_set_event_init(engine_t engine,
        const char *event, int type,
    const char *function,
        event_init_handle address,
        const char *filename, int line);

#define ey_set_userdefine_event_finit(eng,ev,func)      \
    _ey_set_event_finit(eng,#ev,1,#func,func,__FILE__,__LINE__)
#define ey_set_predefine_event_finit(eng,ev,func)       \
    _ey_set_event_finit(eng,#ev,0,#func,func,__FILE__,__LINE__)
extern int _ey_set_event_finit(engine_t engine,
        const char *event, int type,
    const char *function,
        event_finit_handle address,
        const char *filename, int line);

#define ey_set_userdefine_event_preprocess(eng,ev,func) \
    _ey_set_event_preprocessor(eng, #ev, 1,             \
    #func, func, __FILE__, __LINE__)
#define ey_set_predefine_event_preprocess(eng,ev,func)  \
    _ey_set_event_preprocessor(eng, #ev, 0,             \
    #func, func, __FILE__, __LINE__)
extern int _ey_set_event_preprocessor(engine_t engine,
        const char *event, int type,
    const char *function,
        event_preprocess_handle address,
        const char *filename, int line);
```

2.3.2. 说明

本节所涉及的 API 函数，在 2.2.2 中都有讲述，这里不再多做叙述。可以从 libdecoder/http/testsuite/http_xss.ey、libdeocder/pop3/testsuite/pop3.ey、libdecoder/html/testsuite/html.ey 等例子中看到相关 API 的应用。

2.4. 二进制规则库

2.4.1. 二进制规则

eyoung 支持规则的高度可编程性，其中重要的一点就是支持由 eyoung IPS 规则文件通过%import 选项导入外部动态链接库.so 文件。这个特性与 Linux 内核的可加载内核模块.ko 的运行方式比较相近。eyoung 在设计这个特性的时候，要解决的核心问题是**功能的可扩展性、可升级性与执行效率之间的平衡**。主要有三个相关的问题：

1) 符号的共享，即当前 eyoung 引擎运行环境要能感知到足够的外部动态链接库中的符号，包括全局变量（名称、加载后地址和变量类型）、全局函数（名称、入口地址、返回类型和参数列表）、类型定义等等信息。对于这个问题，在 eyoung 的解决方案中，既可以通过#include 头文件的方式向外界发布，也可以通过 eyoung 内置的 EY_EXPORT_IDENT 和 EY_EXPORT_TYPE 两个宏，将相关定义以字符串的形式直接写入到外部动态链接库 ELF 文件中。
2) 动态库可以动态加载和卸载，eyoung 规则语法中，支持%import 关键字，用来动态导入外部动态链接库。动态库的卸载（通常是在规则升级的过程中），则通过在重新加载规则的过程中完成。
3) 外部的动态链接库的加载后可以自动初始化，例如动态库中全局变量的资源分配、初始化操作。eyoung 借鉴了 Linux 内核可加载内核模块.ko 文件的设计思想，将相关的模块 init、模块 finit 函数的入口信息使用 EY_EXPORT_INIT 和 EY_EXPORT_FINIT 两个宏，直接写入到动态链接库自身的 ELF 文件中。在执行动态库加载的时候，eyoung 分析 ELF 文件，得到相关的入口地址，并自动执行 init 函数；规则卸载的时候，执行 finit 函数。

2.4.2. Source Code

```
#ifndef EY_EXPORT_H
#define EY_EXPORT_H 1

#define EY_EXPORT_TYPE_IDENT          1
#define EY_EXPORT_TYPE_TYPE           2
#define EY_EXPORT_TYPE_INIT           3
#define EY_EXPORT_TYPE_FINIT          4

#define EY_IDENT_SECTION              ".eyoung_ident"
#define EY_TYPE_SECTION               ".eyoung_type"
#define EY_INIT_SECTION               ".eyoung_init"
#define EY_FINIT_SECTION              ".eyoung_finit"

typedef struct ey_extern_symbol
```

```
{
    char *name;
    void *value;
    char *decl;
    char *file;
    int line;
    int type;
}ey_extern_symbol_t;

#define EY_EXPORT_IDENT(name, decl)                             \
    static const ey_extern_symbol_t __eyoung_ident_##name    \
    __attribute__((section(EY_IDENT_SECTION), unused)) =     \
    {                                                           \
        #name,                                                  \
        &name,                                                  \
        decl,                                                   \
        __FILE__,                                               \
        __LINE__,                                               \
        EY_EXPORT_TYPE_IDENT                                    \
    };

#define EY_EXPORT_TYPE(name,decl)                               \
    static const ey_extern_symbol_t __eyoung_type_##name     \
    __attribute__((section(EY_TYPE_SECTION), unused)) =      \
    {                                                           \
        #name,                                                  \
        (void*)0,                                               \
        decl,                                                   \
        __FILE__,                                               \
        __LINE__,                                               \
        EY_EXPORT_TYPE_IDENT                                    \
    };

#define EY_EXPORT_INIT(name)                                    \
    static const ey_extern_symbol_t __eyoung_type_##name     \
    __attribute__((section(EY_INIT_SECTION), unused)) =      \
    {                                                           \
        #name,                                                  \
        name,                                                   \
        (void*)0,                                               \
        __FILE__,                                               \
        __LINE__,                                               \
        EY_EXPORT_TYPE_INIT                                     \
    };

#define EY_EXPORT_FINIT(name)                                   \
    static const ey_extern_symbol_t __eyoung_type_##name     \
    __attribute__((section(EY_FINIT_SECTION), unused)) =     \
    {                                                           \
        #name,                                                  \
        name,                                                   \
        (void*)0,                                               \
        __FILE__,                                               \
        __LINE__,                                               \
        EY_EXPORT_TYPE_FINIT                                    \
    };
#endif
```

2.4.3. 说明

1，EY_EXPORT_IDENT

这个宏可以将**全局变量、全局函数的声明**，以字符串的形式写入到动态链接库 ELF 文件的.eyoung_ident 段中。eyoung IPS 引擎在加载 IPS 规则时，执行%import 关键字时，会试图读取要加载的动态库中.eyoung_ident 段（如果存在）的内容，并将符号的声明直接写入到转换后的.eyc 中间结果文件中。

2，EY_EXPORT_TYPE

这个宏将动态库源文件中的**原始类型定义**，写入到动态链接库 ELF 文件的.eyoung_type 段中。eyoung IPS 引擎在加载 IPS 规则时，执行%import 关键字时，会试图读取要加载的动态库中.eyoung_type 段（如果存在）的内容，并将符号的声明直接写入到转换后的.eyc 中间结果文件中。

3，EY_EXPORT_INIT

这个宏可以将一个指定的函数入口，写入到动态链接库 ELF 文件的.eyoung_init 段中。eyoung IPS 引擎在加载 IPS 规则时，执行%import 关键字时，会试图读取要加载的动态库中.eyoung_init 段（如果存在）的内容，得到相关入口函数的地址后执行该入口函数，自动完成初始化功能。

4，EY_EXPORT_FINIT

这个宏可以将一个指定的函数入口，写入到动态链接库 ELF 文件的.eyoung_finit 段中。eyoung IPS 引擎在加载 IPS 规则时，执行%import 关键字时，会试图读取要加载的动态库中.eyoung_finit 段（如果存在）的内容，并记录 finit 函数的入口地址，待到规则卸载时执行该入口函数，自动完成终止化功能，以释放相关资源。

上述功能的示例，在本文 2.5.3 给出。

2.5. 示例

2.5.1. demo_http_xss.c

本示例主要展示 ***ey_engine_create***、***ey_engine_load*** 和 ***ey_engine_destroy*** 的使用。

libdecoder/http/testsuite/demo_http_xss.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "libengine.h"
#include "http.h"
#include "html.h"
```

```
int parse_http_file(http_handler_t decoder,
      const char *filename)
{
  char *line = NULL;
  size_t len = 0;
  ssize_t read = 0;
  int ret = -1;
  http_work_t *work = NULL;
  int lines = 0;
  FILE *fp = fopen(filename, "r");
  if(!fp)
  {
      fprintf(stderr,
          "failed to open file %s\n", filename);
      goto failed;
  }

  work = http_work_create(decoder, 0);
  if(!work)
  {
      fprintf(stderr,
          "failed to alloc http private data\n");
      goto failed;
  }

  while ((read = getline(&line, &len, fp)) != -1)
  {
      lines++;
      if(read <=1)
      {
          fprintf(stderr,
              "invalid line(%d): %s\n", lines, line);
          goto failed;
      }
      if(toupper(line[0])=='C' && line[1]==':')
      {
          if(http_decode_data(work, line+2, read-2, 1, 0))
          {
              fprintf(stderr,
                  "parse client failed, line(%d): %s\n",
                  lines, line);
              goto failed;
          }
      }
      else if (toupper(line[0])=='S' && line[1]==':')
      {
          if(http_decode_data(work, line+2, read-2, 0, 0))
          {
              fprintf(stderr,
                  "parse server failed, line(%d): %s\n",
                  lines, line);
              goto failed;
          }
      }
      else
      {
```

```
            fprintf(stderr,
                "invalid line(%d): %s\n", lines, line);
            goto failed;
        }
    }

    /*give end flag to parser*/
    if(http_decode_data(work, "", 0, 1, 1))
    {
        fprintf(stderr,
            "parse client end flag failed");
        goto failed;
    }
    if(http_decode_data(work, "", 0, 0, 1))
    {
        fprintf(stderr,
            "parse server end flag failed");
        goto failed;
    }

    ret = 0;
    /*pass through*/
    fprintf(stderr, "parser OK!\n");
failed:
    if(work)
        http_work_destroy(work);
    if(line)
        free(line);
    if(fp)
        fclose(fp);
    return ret;
}

int main(int argc, char *argv[])
{
    int ret = 0;
    http_handler_t decoder = NULL;
    engine_t engine = NULL;
    if(argc != 3)
    {
        fprintf(stderr,
            "Usage: http_parser <sig_file> <msg_file>\n");
        return -1;
    }
    debug_http_server_lexer = 0;
    debug_http_server_parser = 0;
    debug_http_client_lexer = 0;
    debug_http_client_parser = 0;
    debug_http_mem = 0;
    debug_http_detect = 0;

    debug_html_lexer = 0;
    debug_html_parser = 0;
    debug_html_mem = 0;
    debug_html_detect = 0;

    debug_engine_parser = 0;
```

```
    debug_engine_lexier = 0;
    debug_engine_init = 0;
    debug_engine_compiler = 0;
    debug_engine_runtime = 0;

    engine = ey_engine_create("http");
    if(!engine)
    {
        fprintf(stderr, "create http engine failed\n");
        ret = -1;
        goto failed;
    }

    if(ey_engine_load(engine, &argv[1], 1))
    {
        fprintf(stderr, "load http signature failed\n");
        ret = -1;
        goto failed;
    }

    decoder = http_decoder_init(engine, NULL);
    if(!decoder)
    {
        fprintf(stderr, "create http decoder failed\n");
        ret = -1;
        goto failed;
    }

    ret = parse_http_file(decoder, argv[2]);

failed:
    if(decoder)
        http_decoder_finit(decoder);
    if(engine)
        ey_engine_destroy(engine);
    return ret;
}
```

2.5.2. http_decode.c

本示例主要展示 ***ey_engine_work_create***、***ey_engine_work_detect_data***、***ey_engine_work_create_event***、***ey_engine_work_detect_event***、***ey_engine_work_destroy_event*** 和 ***ey_engine_work_destroy*** 的用法。**libdecoder/http/decode/http_decode.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

#include "http.h"
#include "http_private.h"
#include "libengine.h"

http_work_t http_work_create(http_handler_t handler, int greedy)
```

```
{
  if(!handler)
  {
      http_debug(debug_http_mem,
          "null http decoder handler\n");
      return NULL;
  }

  http_data_t *priv_data = NULL;
  http_decoder_t *decoder = (http_decoder_t*)handler;
  engine_t engine = decoder->engine;
  priv_data = http_alloc_priv_data(decoder, greedy);
  if(!priv_data)
  {
      http_debug(debug_http_mem,
          "failed to alloc http private data\n");
      return NULL;
  }

  priv_data->decoder = handler;
  if(engine)
  {
      engine_work_t *engine_work =
          ey_engine_work_create(engine);
      if(!engine_work)
      {
          http_debug(debug_http_mem,
              "failed to alloc engine work\n");
          http_free_priv_data(decoder, priv_data);
          return NULL;
      }
      priv_data->engine_work = engine_work;
      engine_work->predefined = (void*)priv_data;
  }

  return priv_data;
}

void http_work_destroy(http_work_t work)
{
  if(work)
  {
      http_data_t *priv_data = (http_data_t*)work;
      http_decoder_t *decoder =
          (http_decoder_t*)(priv_data->decoder);
      if(priv_data->engine_work)
          ey_engine_work_destroy(priv_data->engine_work);
      http_free_priv_data(decoder, priv_data);
  }
}

int http_decode_data(http_work_t work,
       const char *data, size_t data_len,
       int from_client, int last_frag)
{
  assert(work != NULL);
  http_data_t *http_data = (http_data_t*)work;
```

```
  if(http_data->engine_work)
      ey_engine_work_detect_data(http_data->engine_work,
      data, data_len, from_client);
  if(from_client)
      return parse_http_client_stream(http_data,
          data, data_len, last_frag);
  else
      return parse_http_server_stream(http_data,
          data, data_len, last_frag);
}

http_handler_t http_decoder_init(engine_t engine,
  void *html_decoder)
{
  http_decoder_t *decoder =
      (http_decoder_t*)http_malloc(sizeof (http_decoder_t));
  if(!decoder)
  {
      http_debug(debug_http_mem,
          "failed to alloc http decoder\n");
      goto failed;
  }
  memset(decoder, 0, sizeof(*decoder));

  if(http_mem_init(decoder))
  {
      http_debug(debug_http_mem,
          "failed to init http decoder mem\n");
      goto failed;
  }

  decoder->html_decoder = html_decoder;
  if(engine)
  {
      decoder->engine = engine;
      http_server_register(decoder);
      http_client_register(decoder);
  }
  return (http_handler_t)decoder;

failed:
  if(decoder)
      http_decoder_finit((http_handler_t)decoder);
  return NULL;
}

void http_decoder_finit(http_handler_t handler)
{
  if(handler)
  {
      http_decoder_t *decoder = (http_decoder_t*)handler;
      http_mem_finit(decoder);
      http_free(decoder);
  }
}

int http_element_detect(http_data_t *http_data,
```

```
    const char *event_name, int event_id, void *event,
    char *cluster_buffer, size_t cluster_buffer_len)
{
  if(!http_data->engine_work)
  {
      http_debug(debug_http_detect,
          "engine work is not created, skip scan\n");
      return 0;
  }

  if(!event_name)
  {
      http_debug(debug_http_detect, "event name is null\n");
      return 0;
  }

  if(event_id < 0)
  {
      http_debug(debug_http_detect,
          "event id %d for event %s is illegal\n",
          event_id, event_name);
      return 0;
  }

  if(!http_data || !event)
  {
      http_debug(debug_http_detect,
          "bad parameter for event %s\n", event_name);
      return 0;
  }

  engine_action_t action = {ENGINE_ACTION_PASS};
  engine_work_t *work = http_data->engine_work;
  engine_work_event_t *work_event =
       ey_engine_work_create_event(work, event_id, &action);
  if(!work_event)
  {
      http_debug(debug_http_detect,
          "create event for %s failed\n", event_name);
      return 0;
  }
  ey_engine_work_detect_event(work_event, event);
  ey_engine_work_destroy_event(work_event);
  http_debug(debug_http_detect,
             "detect %s[%d], get actoin %s\n",
      event_name, event_id,
             ey_engine_action_name(action.action));
  if(action.action==ENGINE_ACTION_PASS)
      return 0;
  return -1;
}
```

2.5.3. export_test.c

本示例主要展示 **EY_EXPORT_IDENT**、**EY_EXPORT_INIT**、**EY_EXPORT_TYPE** 和 **EY_EXPORT_TYPE** 的用法。

test/export_test.c

```c
#include <stdio.h>
#include "ey_export.h"

int a=1;

int foo(void *link, void *event)
{
  printf("call foo, a=%d\n", ++a);
  return 1;
}

int bar(void *link, void *event)
{
  printf("call bar, a=%d\n", ++a);
  return 0;
}

int test_init(void *eng)
{
  printf("call init, a=%d\n", a++);
}

int test_exit(void *eng)
{
  printf("call finit, a=%d\n", a--);
}

struct s
{
  int a;
  int b;
};

EY_EXPORT_IDENT(a, "extern int a;");
EY_EXPORT_IDENT(foo, "int foo(void *link, void *event);");
EY_EXPORT_IDENT(bar, "int bar(void *link, void *event);");
EY_EXPORT_TYPE(s, "struct s{int a; int b;};");

EY_EXPORT_INIT(test_init);
EY_EXPORT_FINIT(test_exit);
```

3. 协议分析器

协议分析器对 Layer7 数据进行协议分析，并且以层次化、结构化的方式将分析之后的数据以协议事件（event）的形式提交给 eyoung IPS 引擎，用以在规则的指引下进行分析和检测。为了达到这个目的，eyoung 协议分析器从结构上大致分为协议词法分析器 Lexer 和协议语法分析器 Parser。

3.1. Stream-mode 协议词法分析器

3.1.1. 设计

应用层协议词法分析器（Lexer）在协议分析器中更贴近底层，即更多地与网络数据直接交互，主要的作用是对获取的网络数据，按照分析规则，将原始报文拆解成一系列的"词元"。这一点同编译器中的词法分析器的作用是一致的。不同的是，编译器是对本地磁盘上的文件进行操作，多采用阻塞式的文件 IO 功能（例如 fread、fscanf 等等），而用于协议分析的词法分析器不可能采用阻塞的方式，必须在网络数据"流"上进行词法分析。为此，eyoung 改造了 GNU flex 的用法，解决了以下问题：

1) 输入缓冲区自动管理。对于上一个输入缓冲区未被匹配的部分，例如当前的输入缓冲区的结尾部分仅部分匹配一个模式，此时 eyoung lexer 不按照 GNU flex 中 default 模式返回，而是将结尾部分未完全匹配的部分"copy and save"。待到下一次送入输入缓冲区后，再次拼接并重新匹配。
2) greedy 模式。在贪婪模式下，如果上一个缓冲区的结尾恰好匹配了某一个模式，此时将 eyoung lexer 直接返回匹配的模式；反之，eyoung lexer 按照 1)的逻辑保存最后匹配的内容，并在下一个输入缓冲区到来时拼接缓冲区并重新匹配。这是因为，要匹配的模式可能存在包含关系，例如"b"和"ba"，如果当前缓冲区以"b"结尾，此时有可能匹配"ba"（只需要下一个输入缓冲区以"a"开头即可）。在 greedy 模式下，eyoung lexer 会返回第一个模式"b"；反之则保存结尾的"b"，待到下一个输入缓冲区到来之后再做判断。注意：eyoung lexer 使用的是 GNU flex 工具，而 GNU flex 使用的是 greedy 模式且不能修改，只是 GNU flex 并没有在 stream 上考虑这个问题。eyoung non-greedy 模式仅仅是针对当前输入报文结尾部分的，并不会导致输入报文非结尾的词法分析使用非 greedy 模式。

3.1.2. 示例

```
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "http.h"
#include "http_server_parser.h"
#include "http_private.h"

#define YY_USER_ACTION                                  \
  if(yyg->yy_hold_char==YY_END_OF_BUFFER_CHAR &&    \
      save_server_stream_context(yyg,0))            \
      return TOKEN_SERVER_CONTINUE;

#ifdef YY_RESTORE_YY_MORE_OFFSET
  #undef YY_RESTORE_YY_MORE_OFFSET
  #define YY_RESTORE_YY_MORE_OFFSET                 \
  {                                                 \
      if(save_server_stream_context(yyg,1))         \
          return TOKEN_SERVER_CONTINUE;             \
  }
#endif
%}

%option header-file="http_server_lex.h"
%option outfile="http_server_lex.c"
%option prefix="http_server_"
%option bison-bridge
%option noyywrap
%option reentrant
%option case-insensitive
%option ansi-definitions
%option noinput
%option nounput
%option noyyalloc
%option noyyfree
%option noyyrealloc
      ……
%%
       ……
%%
       ……
struct yy_buffer_state* http_server_scan_stream(
       const char *new_buf, size_t new_buf_len, http_data_t *priv)
{
  YY_BUFFER_STATE b;
  char *buf;
  yy_size_t n, _yybytes_len;
  char *last_buf = priv->response_parser.saved;
  size_t last_buf_len = priv->response_parser.saved_len;
  yyscan_t scanner = (yyscan_t)priv->response_parser.lexier;
```

```
  _yybytes_len = new_buf_len + last_buf_len;
  n = _yybytes_len + 2;
  buf = (char *)http_server_alloc(n, scanner);
  if (!buf)
  {
      http_debug(debug_http_server_lexer,
          "out of dynamic memory in http_server_scan_stream()\n");
      return NULL;
  }

  if(last_buf)
      memcpy(buf, last_buf, last_buf_len);

  if(new_buf)
      memcpy(buf+last_buf_len, new_buf, new_buf_len);

  buf[_yybytes_len] = buf[_yybytes_len+1] = YY_END_OF_BUFFER_CHAR;
  http_debug(debug_http_server_lexer, "[HTTP SERVER]: %s\n", buf);

  //b = http_server_scan_buffer(buf, n, scanner);
  b = (YY_BUFFER_STATE)http_server_alloc(sizeof(struct
      yy_buffer_state), scanner);
  if ( ! b )
  {
      http_debug(debug_http_server_lexer,
          "failed to alloc server buffer state\n");
      http_server_free(buf, scanner);
      return NULL;
  }

  b->yy_buf_size = n - 2;  /* "- 2" to take care of EOB's */
  b->yy_buf_pos = b->yy_ch_buf = buf;
  b->yy_is_our_buffer = 0;
  b->yy_input_file = 0;
  b->yy_n_chars = b->yy_buf_size;
  b->yy_is_interactive = 0;
  b->yy_at_bol = 1;
  b->yy_fill_buffer = 0;
  b->yy_buffer_status = YY_BUFFER_NEW;
  http_server_switch_to_buffer(b, scanner);
  b->yy_is_our_buffer = 1;

  if(priv->response_parser.saved)
  {
      http_server_free(priv->response_parser.saved, scanner);
      priv->response_parser.saved = NULL;
      priv->response_parser.saved_len = 0;
  }
  return b;
}

static int save_server_stream_context(
       yyscan_t yyscanner, int from_default)
{
  struct yyguts_t * yyg = (struct yyguts_t*)yyscanner;
  http_data_t *priv = (http_data_t*)http_server_get_extra(yyg);
  int len = 0;
```

```
    if(priv->response_parser.saved)
    {
        http_server_free(priv->response_parser.saved, yyg);
        priv->response_parser.saved = NULL;
        priv->response_parser.saved_len = 0;
    }

    if(!priv || priv->response_parser.last_frag ||
               (!from_default && !priv->response_parser.greedy))
    {
        http_debug(debug_http_server_lexer,
            "No need to save stream context\n");
        return 0;
    }

    len = from_default?yyleng-1:yyleng;
    if(!len)
    {
        http_debug(debug_http_server_lexer,
            "Exit save stream context for ZERO length yytext\n");
        return 1;
    }

    priv->response_parser.saved = http_server_alloc(len, yyg);
    if(!priv->response_parser.saved)
    {
        http_debug(debug_http_server_lexer,
            "out of memory while saving context\n");
        return 0;
    }
    memcpy(priv->response_parser.saved, yytext, len);
    priv->response_parser.saved_len = len;
    http_debug(debug_http_server_lexer,
        "Save stream context, string: %s, len: %d\n", yytext, len);
    return 1;
}
```

说明：

http_server_scan_stream 函数用来组装当前的扫描缓冲区，如果 eyoung lexer 之前没有保存待确定的内容，则直接用当前的输入缓冲区做本次扫描缓冲区；否则，则分配一块缓冲区，其长度为保存的待确定内容长度+当前输入缓冲区长度，并把两段缓冲区 copy 到新分配的缓冲区中。**这里存在优化空间留待以后完成**，即可以不保存第一次匹配的结尾数据，而只保存结尾的匹配状态，这样可以避免一次内存分配和数据拷贝！

save_server_stream_context 函数被调用的地方有两处：YY_USER_ACTION 宏和 YY_RESTORE_YY_MORE_OFFSET 宏，这两个宏都定义在 GNU flex 的代码模板中。

- YY_USER_ACTION 是当某一个模式被匹配时调用，按照 3.1.1 中 2)的介绍，需要判断当前匹配位置是否到了输入缓冲区的结尾同时判断是否开启了 greedy 模式，如果到了结束位置且开启了 greedy 模式，则保存结尾的匹

配部分，并返回 TOKEN_CONTINUE 通知 parser。

- YY_RESTORE_YY_MORE_OFFSET，是在 lexer 到达当前输入缓冲区结尾后，且并没有找到任何可匹配模式（实际上进入 GNU flex 的 default 处理）时被调用。由于 eyoung lexer 是 stream-based，这种情况可能不代表不匹配任何模式。此时如果当前输入缓冲区不是 stream 上的最后一个缓冲区，则保存最后不匹配任何模式的部分，并返回 TOKEN_CONTINUE 通知 parser 等待下一个输入缓冲区。

3.2. Push-Mode 协议语法分析器

3.2.1. 设计

eyoung 协议分析器的核心是一个协议语法分析器（Parser），这个 Parser 在协议语法规则（通常是一个上下文无关文法）的指引下，进行语法分析。语法分析器一方面检查协议数据在语法和语义上是否符合协议规范，另一方面自动地把分析后的相关数据传递到 eyoung IPS 引擎进行攻击检测。

传统的语法分析器，例如 bison 生成的语法分析器，其工作模式通常采纳“拉”（PULL）的方式，即语法分析器内部通过默认的接口调用词法分析器获取下一个词元，词法分析器再进行文件 I/O 和缓冲区操作。这种方式对于阻塞式 I/O 的词法分析器是合适的，但是对于网络数据检测，这种方式的阻塞式特性会对性能造成非常严重的影响。为了解决这个问题，eyoung 利用了 bison 的新特性，那就是 Push-Mode Parser。这种模式下，词法分析器的调用实际是先于语法分析器被调用，然后将词法分析器得到的词元，以参数的形式“压”（PUSH）入语法分析器。

在 eyoung 中，由于词法分析器当遇到数据流结尾位置时，可能会进行缓冲区”copy and save”，并返回 TOKEN_CONTINUE，控制流将不调用语法分析器的功能。

3.2.2. 示例

libdecoder/http/decode/http_server.y

```
int parse_http_server_stream(http_data_t *priv,
      const char *buf, size_t buf_len, int last_frag)
{
    http_server_pstate *parser =
        (http_server_pstate*)priv->response_parser.parser;
    yyscan_t lexier = (yyscan_t)priv->response_parser.lexier;
    YY_BUFFER_STATE input = NULL;
    int token = 0, parser_ret = 0;
    HTTP_SERVER_STYPE value;

    yydebug = debug_http_server_parser;
```

```
    priv->response_parser.last_frag = last_frag;
    input = http_server_scan_stream(buf, buf_len, priv);
    if(!input)
    {
        http_debug(debug_http_server_parser,
            "create http server stream buffer failed\n");
        return 1;
    }

    while(1)
    {
        memset(&value, 0, sizeof(value));
        if(http_server_lex_body_mode(lexier))
            token = http_server_body_lex(&value, lexier);
        else
            token = http_server_lex(&value, lexier);
        if(token == TOKEN_SERVER_CONTINUE)
            break;
        parser_ret = http_server_push_parse(parser, token,
                    &value, (void*)priv);
        if(parser_ret != YYPUSH_MORE)
            break;
    }
    http_server_delete_buffer(input, lexier);

    if(parser_ret != YYPUSH_MORE && parser_ret != 0)
    {
        http_debug(debug_http_server_parser,
            "find error while parsing http server stream\n");
        return 2;
    }
    return 0;
}
```

这个函数在底层 I/O 收到收到一段网络报文<buf, buf_len>之后，以 PUSH 的方式调用 parse_http_server_stream。其中：

- http_server_scan_stream 负责组装词法分析器使用的 buffer 结构，细节在 3.1.2 中已经做了介绍。
- http_server_delete_buffer 负责销毁创建的 buffer 结构，这个是 GNU flex 提供的 API 函数，详情参见 GNU flex manual。
- http_server_lex_body_mode 是一个返回布尔值的函数，用来判断当前词法分析器是否分析到 HTTP 报文的 BODY 部分。由于 HTTP BODY 不管是使用 CONTENT-LENGTH 定义还是使用 CHUNK TRANSFER-ENCODING，BODY 内容的解析都是依赖长度信息的。这种计数的解析并不适合使用基于正则表达式的 FLEX 完成。所以，eyoung 在解析此类数据时，将 BODY 部分和非 BODY 部分分别解析。http_server_lex_body_mode 返回 TRUE 时，eyoung HTTP 协议分析其使用 http_server_body_lex 对 BODY 部分进行词法分析；反之使用 http_server_lex 进行非 BODY 部分的解析，http_server_lex 是由 FLEX 生成的标准函数。
- http_server_body_lex 或 http_server_lex 返回 SYM_CONTINUE 时，意味着当前缓冲区已经全部读取完毕，并可能把当前缓冲区的最后一

段”copy and save”。此时，整个解析过程应该跳出 while 循环。

- http_server_push_parse 函数是 Push-Mode Parser 的入口，该函数由 GNU bison 自动生成，函数的参数是词法分析器的返回结果。http_server_push_parse 返回 0 时，表示输入的词元序列已经完全被语法分析器所接受；返回 YYMORE_PUSH，表示当前输入词元符合语法，但是需要后续词元序列推进语法分析器的状态，直到返回 0 或发现错误；其它返回值意味着发现语法错误。

3.3. 事件

3.3.1. 事件的类型

事件的声明，是使用规则文件完成的，详细的语法通过《Signatures Specification》描述。另外，在本文档的 2.2.2 节的 ey_engine_find_event 的介绍中，已经讲述了如何在 eyoung 协议分析器中获取已经创建事件的索引值的方法。本节讲述的是事件的类型定义机制以及事件的发射机制。

在上下文无关语言的语法分析中，通常采用“自底向上”的 LR 分析法（或 LR 改进算法，如 LALR），即“移动—规约”分析法。在这种分析法中，直接由词法分析器得到的词元被称为“终结符”（Terminal Token），而在规约过程中产生的符号被称为“非终结符”（Non-Terminal Token）。**eyoung 的事件本质上就是由 LR 分析中产生的“非终结符”**。随着协议分析，协议数据语法分析器会自底向上地产生一系列“非终结符”，这些“非终结符”从概念上也必然是“自底向上”的，即从微观的概念到宏观的概念。这样 IPS 规则就获得了在不同概念层面描述协议内容细节的能力。

事件的名称就是“非终结符”的名称，事件的类型就是“非终结符”的类型。在 GNU bison 语法定义中，可以通过使用***%union*** 和***%type*** 关键字对“非终结符”定义类型。例如，协议语法规则文件 http_server.y 中：

```
%union
{
    ……
    http_response_header_t *header;
    ……
}

……
%type <header> response_header_host
……
```

这段代码意味着“非终结符” response_header_host 在语法分析器中的类型是 http_response_header_t 的指针。如果要让 eyoung IPS 引擎感知事件 response_header_host 的类型，必须通过规则文件。所以上述例子在 HTTP 规则文件中有如下的描述：

Libdecoder/http/testsuite/http_xss.ey

……

```
%event "response_header_host"       "http_response_header_t *"
```

……

3.3.2. 事件的提交

协议数据语法分析器在“自底向上”的语法分析过程中，每规约出来一个“非终结符”，就将此“非终结符”以对应的事件格式提交给 eyoung IPS 引擎进行扫描和检查。为了达到这个目的，eyoung 修改了 GNU bison 生成代码的模板 yacc.c，patch 文件是 tool/yacc.c.diff。同时，在协议分析语法文件中需要做出配合修改：

Libdecoder/http/decode/http_server.y 文件片段：

```
#ifdef YY_REDUCTION_CALLBACK
#undef YY_REDUCTION_CALLBACK
#endif
#define YY_REDUCTION_CALLBACK(data,name,id,val)             \
    do                                                      \
    {                                                       \
        if(http_element_detect(data,name,id,val)<0)         \
        {                                                   \
            http_debug(debug_http_detect, "find attack!\n"); \
            return -1;                                      \
        }                                                   \
    }while(0)
```

YY_REDUCTION_CALLBACK 这个宏是 eyoung 对 GNU bison 的扩展，该宏是在 LR 语法分析器在规约动作发生时被自动调用。http_element_detect 函数已经在 2.5.2 中给出了实现。经过 Patch 之后的 GNU bison 模板文件 yacc.c 会显示 YY_REDUCTION_CALLBACK 是如何被调用的，读者可以自己查看。