Signatures Specification

eyoung.father@gmail.com

2014/3/21

目录

1. 总体结构

1.1. 语法定义

```
eyoung_file:
    prologue_opt  TOKEN_DPERCENT
    signature_opt TOKEN_DPERCENT
    epilogue_opt
    ;
```

1.2. 语法说明

eyoung IPS 规则文件从总体结构上大致分为三个部分：规则前言、规则体和规则后记，它们之间以“%%”分隔。这三个部分都是可选的，但是分隔符“%%”是不能省略的。这一点，eyoung IPS 规则文件同 bison 的语法文件类似。

示例：

```
<prologue>
%%
<signature>
%%
<epilogue>
```

2. 规则前言

规则前言主要用来定义规则中关键的元数据信息，包括事件、相关函数入口、外部库文件引用、外部函数声明等等。

2.1. 语法定义

```
prologue_opt:
    empty
    | prologue_list
    ;

prologue_list:
    prologue
```

```
    | prologue_list prologue
    ;

prologue:
    TOKEN_PROLOGUE_CODE
    | TOKEN_OUTPUT TOKEN_STRING
    | TOKEN_IMPORT TOKEN_STRING
    | TOKEN_FILE_INIT TOKEN_STRING
    | TOKEN_FILE_FINIT TOKEN_STRING
    | TOKEN_WORK_INIT TOKEN_STRING
    | TOKEN_WORK_FINIT TOKEN_STRING
    | TOKEN_EVENT_INIT TOKEN_STRING TOKEN_STRING
    | TOKEN_EVENT_PREPROCESSOR TOKEN_STRING TOKEN_STRING
    | TOKEN_EVENT_FINIT TOKEN_STRING TOKEN_STRING
    | TOKEN_EVENT TOKEN_STRING TOKEN_STRING
    ;

empty:
    ;
```

这里，prologue_opt 部分要么是 empty，表示当前规则文件不存在规则前言；要么是一个 prologue_list 链。prologue_list 由不少于一个 prologue 组成，每一个 prologue 都是一个独立的前言定义。

2.2. Prologue Code

Prologue Code 是使用成对的"%{"和"}%"包含的 C 语言代码，代码的格式与 C99 标准兼容。Prologue Code 不支持嵌套，其中包含的代码会被完整地拷贝到.eyc 中间代码文件中，用于规则加载后中间代码的实时编译。例如：

```
%{
    #include <stdio.h>
    #include "myheader.h"

    #ifdef SOMETHING
    #undef SOMETHING
    #endif
    #define SOMETHING xxx

    /*
     * c style comments
    */
    //c++ style comments
    typedef struct my_struct
    {
        int a;
    }my_struct_t;
    extern int my_external_func(void);

    static int aaaa;
    static int my_static_func(void);
}%
```

2.3. %output 选项

格式：

%**output** “*file-name*”

说明：

%output 选项用来定义当前规则文件经过预处理之后的文件名称，使用 C99 风格字符串的形式给出——双引号包含的字符序列。不使用该选项时，中间文件的名称默认在当前规则文件名称之后添加“.c”后缀。例如，如果当前处理的规则文件是 http.ey 文件，默认会被预处理为 http.ey.c 文件。如果使用%output “http.eyc”选项，则会将 http.ey 处理后保存为 http.eyc。**推荐使用该选项，并将.eyc 作为中间文件的扩展名。**

示例：

```
%output “http.eyc”
```

2.4. %file-init 选项

格式：

%**file-init** “*function-name*”

说明：

%file-init 选项用来向 eyoung IPS 引擎注册一个文件级构造函数，该函数类型定义在 include/libengine_type.h 中：

```
typedef int (*file_init_handle)(engine_t eng);
```

该构造函数在所有规则文件被解析完成后，且 JIT 实时编译器完成动态编译和链接后，由 eyoung IPS 引擎按照规则文件解析的顺序依次调用。**构造函数返回 0 表示执行成功，否则表示执行失败。eyoung IPS 引擎发现执行失败的初始化函数后，就会停止加载规则。**

注意：由于这个初始化函数是在所有规则文件编译、链接之后才被执行，所以要求这个函数具有**“外部”属性**，它可以是规则文件内定义的全局函数，也可以是通过%import 选项加载的动态链接库中的全局函数，但是无法使用 static 函数！

示例：

```
%{
#include “libengine.h”
extern int my_file_init(engine_t eng);
}%

%file-init “my_file_init”
```

2.5. %file-finit 选项

格式：

%**file-finit** “*function-name*”

说明：

%file-finit 选项用来向 eyoung IPS 引擎注册一个文件级析构函数，该函数类型定义在 include/libengine_type.h 中：

```
typedef int (*file_finit_handle)(engine_t eng);
```

该析构函数在规则卸载时，由 eyoung IPS 引擎按照当初规则文件被加载的顺序依次调用，用以释放规则文件内分配的资源。函数返回 0 表示执行成功，否则表示执行失败。**执行失败意味着可能会有资源没有被释放，造成资源的泄露，需要由规则开发人员仔细检查失败的原因。**

注意：与%file-init 选项类似，%file-finit 也要求注册的函数具有**“外部”属性**，无法使用 static 静态函数！

另外，还可以使用头文件 include/libengine_export.h 中定义的 **ey_add_file_finit** 宏，在%file-init 注册的函数中调用该宏完成相关功能。此时允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include “libengine.h”
extern int my_file_finit(engine_t eng);
}%

%file-finit “my_file_finit”
```

(2)：**推荐**

```
%{
#include “libengine.h”
static int my_file_finit(engine_t eng);
%}
%file-init “my_file_init”
%%
%%
static int my_file_finit(engine_t eng)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_add_file_finit(eng, my_file_finit);
    return 0;
}
```

2.6. %work-init 选项

格式：

%**work-init** "*function-name*"

说明：

%work-init 选项用来向 eyoung IPS 引擎注册**最多一个**构造函数，该函数类型定义在 include/libengine_type.h 中：

```
typedef int (*work_init_handle)(engine_work_t *work);
```

当一个 engine_work_t 对象被创建时，eyoung IPS 引擎自动调用这个注册的构造函数。该构造函数用来负责 work 级的资源的分配和初始化，work 的具体概念请参考《Programming Guide》。该构造函数返回 0，表示初始化成功，否则表示失败。执行失败时，会造成 work 对象的创建失败。

注意：与%file-init 选项类似，%work-init 也要求注册的函数具有**"外部"属性**，无法使用 static 静态函数！

此外，还可以使用头文件 include/libengine_export.h 中定义的宏 **ey_set_userdefine_work_init** 完成相同功能。此时需要在%file-init 注册的函数中调用该宏完成相关工作。这种方式时允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include "libengine.h"
extern int my_work_init(engine_work_t *work);
}%

%work-init "my_work_init"
```

(2)：**推荐**

```
%{
#include "libengine.h"
static int my_work_init(engine_work_t *work);
%}
%file-init "my_file_init"
%%
%%
static int my_work_init(engine_work_t *work)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_set_userdefine_work_init(eng, my_work_init);
    return 0;
}
```

2.7. %work-finit 选项

格式：

%**work-finit** "*function-name*"

说明：

%work-finit 选项用来向 eyoung IPS 引擎注册**最多一个**析构函数，函数类型定义在 include/libengine_type.h 中：

typedef int (*work_finit_handle)(engine_work_t *work);

当一个 work 对象被销毁时，eyoung IPS 引擎自动调用这个注册的析构函数。该析构函数用来释放和清理 work 级别的资源，work 的具体的概念参考《Programming Guide》。该函数返回 0，表示清理成功，否则表示失败。**函数执行失败会造成 work 级的资源泄露，规则开发人员必须要仔细检查失败原因。**

注意：与%file-init 选项类似，%work-finit 也要求注册的函数具有**"外部"属性**，无法使用 static 静态函数！

此外，定义在头文件 include/libengine_export.h 中的宏 **ey_set_userdefine_work_finit** 也可以完成相同功能。此时，需要在%file-init 注册的函数中调用该宏完成相关工作。这种用法允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include "libengine.h"
extern int my_work_finit(engine_work_t *work);
}%

%work-init "my_work_finit"
```

(2)：**推荐**

```
%{
#include "libengine.h"
static int my_work_finit(engine_work_t *work);
%}
%file-init "my_file_init"
%%
%%
static int my_work_finit(engine_work_t *work)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_set_userdefine_work_finit(eng, my_work_finit);
    return 0;
}
```

2.8. %event-init 选项

格式：

%**event-init** *"event-name" "function-name"*

说明：

%event-init 选项用来为名称为"event-name"的事件向 eyoung IPS 引擎注册**最多一个**构造函数，函数类型定义在 include/libengine_type.h 中：

```
typedef int (*event_init_handle)(engine_work_event_t *event);
```

当一个 event 对象被创建时，eyoung IPS 引擎自动调用注册的构造函数。该函数用来负责 event 级别资源的分配和初始化，event 的具体概念请参考《Programming Guide》。构造函数返回 0，表示初始化成功，否则表示失败。执行失败时，会造成 event 对象创建失败。

注意：与%file-init 选项类似，%event-init 也要求注册的函数具有**"外部"属性**，无法使用 static 静态函数！

此外，定义在头文件 include/libengine_export.h 中的宏 **ey_set_userdefine_event_init** 也可以完成相同功能。此时，需要在%file-init 注册的函数中调用该宏完成相关工作。这种做法允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include "libengine.h"
extern int my_event_init(engine_work_event_t *event);
}%
%event "my_ev" "void"
%event-init "my_ev" "my_event_init"
```

(2)：**推荐**

```
%{
#include "libengine.h"
static int my_event_init(engine_work_event_t *event);
%}
%event "my_ev" "void"
%file-init "my_file_init"
%%
%%
static int my_event_init(engine_work_event_t *event)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_set_userdefine_event_init(eng, my_ev, my_event_init);
    return 0;
}
```

2.9. %event-preprocessor 选项

格式：

%**event-preprocessor** *"event-name" "function-name"*

说明：

event 对象被创建后，可能被分多次提交到 eyoung IPS 引擎进行攻击检测。%event-preprocessor 选项用来为名称为"event-name"的事件向 eyoung IPS 引擎注册**最多一个**预处理函数，函数类型定义在 include/libengine_type.h 中：

```
typedef int (*event_preprocess_handle)(engine_work_event_t *event);
```

当一个 event 对象实例被提交到 eyoung IPS 引擎检测时，eyoung IPS 引擎自动调用这个注册的函数完成诸如数据格式转换等操作，event 的具体概念请参考《Programming Guide》。函数返回 0，表示预处理成功，否则表示失败。**函数执行失败时，会造成 event 对象不能被 eyoung IPS 引擎检测。**

注意：与%file-init 选项类似，%event-preprocessor 也要求注册的函数具有**"外部"属性**，无法使用 static 静态函数！

此外，定义在头文件 include/libengine_export.h 中的宏 **ey_set_userdefine_event_preprocessor** 也可以完成相同功能。此时，需要在%file-init 注册的函数中调用该宏完成相关工作。这种做法允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include "libengine.h"
extern int my_event_preprocessor(engine_work_event_t *event);
}%
%event "my_ev" "void"
%event-preprocessor "my_ev" "my_event_preprocessor"
```

(2)：**推荐**

```
%{
#include "libengine.h"
static int my_event_preprocessor(engine_work_event_t *event);
%}
%event "my_ev" "void"
%file-init "my_file_init"
%%
%%
static int my_event_preprocessor(engine_work_event_t *event)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_set_userdefine_event_preprocessor(eng,
        my_ev, my_event_preprocessor);
    return 0;
```

```
}
```

2.10. %event-finit 选项

格式：

%event-finit *"event-name" "function-name"*

说明：

%event-finit 选项用来为名称为"event-name"的事件向 eyoung IPS 引擎注册**最多一个**析构函数，函数类型定义在 include/libengine_type.h 中：

```
typedef int (*event_finit_handle)(engine_work_event_t *event);
```

当一个 event 对象被销毁时，eyoung IPS 引擎自动调用这个注册的析构函数。该函数用来释放 event 级别的资源，event 的具体概念请参考《Programming Guide》。该析构函数返回 0，表示资源释放成功，否则表示失败。**析构函数执行失败时，会造成 event 级的资源泄露，规则开发人员必须检查失败的原因。**

注意：与%file-init 选项类似，%event-finit 也要求注册的函数具有**"外部"属性**，无法使用 static 静态函数！

此外，定义在头文件 include/libengine_export.h 中的宏 **ey_set_userdefine_event_finit** 也可以完成相同功能。此时，需要在%file-init 注册的函数中调用该宏完成相关工作。这种用法允许注册规则文件内定义的 static 函数。**实际应用中，推荐使用本用法。**

示例：

(1)：

```
%{
#include "libengine.h"
extern int my_event_finit(engine_work_event_t *event);
}%
%event "my_ev" "void"
%event-finit "my_ev" "my_event_finit"
```

(2)：**推荐**

```
%{
#include "libengine.h"
static int my_event_finit(engine_work_event_t *event);
%}
%event "my_ev" "void"
%file-init "my_file_init"
%%
%%
static int my_event_finit(engine_work_event_t *event)
{
    return 0;
}

int my_file_init(engine_t eng)
{
    ey_set_userdefine_event_finit(eng, my_ev, my_event_finit);
```

```
    return 0;
}
```

2.11. %event 选项

格式：

%**event** *"event-name" "event-type"*

说明：

《Programming Guide》中谈到：**"事件本质上是由 LR 语法分析过程中产生的非终结符，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"**。所以，事件的定义本质上是在协议分析器的设计过程中完成的，此处只是通过%event 选项告知 eyoung IPS 引擎，主要包括事件名称和事件类型——**这些两个属性必须与协议分析器的设计相匹配**。

%event 选项执行后有两种效果，首先是在 eyoung IPS 引擎内部分配并初始化了一个 ey_event_t 的对象，用来记录与该事件相关的信息；其次，会在规则转换后的.eyc 中间代码文件中添加如下代码：

```
typedef event-type *event-name;
```

定义一个在规则中可以直接使用的、与事件名称同名的**指针类型。**

示例：

```
%event "response_list"          "void"
%event "response_header_server" "http_response_header_t *"
```

在.eyc 中间代码文件中，会有如下定义：

```
typedef void *response_list;
typedef http_response_header_t *response_header_server;
```

2.12. %import 选项

格式：

%import "*dynamic-library-name*"

说明：

《Programming Guide》的"二进制规则库"一节，介绍了 eyoung IPS 引擎提供的一种 IPS 规则中加载外部动态链接库的机制。%import 选项是二进制规则的入口，它负责将外部动态链接库引入到当前程序的运行镜像中。

%import 的执行会导致四个作用：（1）eyoung IPS 引擎使用 GNU libdl 的功能将 dynamic-library-name 指示的库动态加载到当前进程运行镜像中；（2）eyoung IPS 引擎使用 libelf 的功能，将动态库中的 eyoung 扩展段.eyoung_type 和.eyoung_ident 的内容读取到预处理之后的.eyc 文件中；（3）如果动态链接库文件中包含.eyoung_init 段，则读取此段中包含的函数入口地址并执行，完成动态链接库文件加载之后的第一次初始化操作；（4）如果动态链接库文件中包含.eyoung_finit 段，则读取并记录其中的函数入

口地址，当动态库卸载的时候由 eyoung IPS 引擎执行该函数，完成动态链接库文件卸载时的资源清理工作。

对于第（2）点，以 test/export_test.c 为例

```
#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);
```

加载之后，将会在预处理之后的.eyc 中间文件中有如下内容：

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

同时，初始化函数 test_init 会被执行。

3. 规则体

3.1. 语法定义

```
signature_opt:
    empty
    | signatures
    ;

signatures:
    signature
    | signatures signature
    ;

signature:
    signature_lhs TOKEN_COLON signature_pipe_list TOKEN_SEMICOLON
    ;

signature_lhs:
    TOKEN_INT
    ;

signature_pipe_list:
    signature_rhs_list
    | signature_pipe_list TOKEN_PIPE signature_rhs_list
    ;

signature_rhs_list:
    signature_rhs
    | signature_rhs_list signature_rhs
    ;

signature_rhs:
    rhs_name rhs_condition_opt rhs_cluster_opt rhs_action_opt
    ;

rhs_name:
    TOKEN_ID
    ;

rhs_condition_opt:
    empty
    | TOKEN_RHS_CONDITION
    ;

rhs_cluster_opt:
    empty
    | TOKEN_SLASH TOKEN_ID TOKEN_COLON TOKEN_STRING
    ;
```

```
rhs_action_opt:
    empty
    | TOKEN_RHS_ACTION
    ;
```

3.2. 语法说明

1，signature

规则体由若干条规则（signature）组成。每条规则的基本结构是：

<id> : signature_pipe_list ;

- 标点符号冒号“:”和分号“;”不能省略。
- id 是一个十进制整数，是规则在 eyoung IPS 引擎中唯一的索引值，在同一个 engine 对象中，id 不允许重复，否则会被视为规则解析阶段的语法错误。
- signature_pipe_list 的定义见 2。

2，signature_pipe_list

signature_pipe_list 是由竖线“|”分隔的子规则链（每一条子规则称为 signature_rhs_list，定义见下文），子规则间是“OR”的关系，即竖线两侧的子规则有一条命中即意味着整个 signature 的命中。例如：

1 : pipe0 **|** *pipe1* **|** *pipe2 ;*

表示 id 为 1 的规则由三个子规则组成，分别是 pipe0、pipe1 和 pipe2，语义上是 pipe0 ***OR*** pipe1 ***OR*** pipe2

3，signature_rhs_list

由不少于一个 signature_rhs 组成

4，signature_rhs

signature_rhs 是基本的检测单元，其格式是：

<ev-name> [(<C-Format expr>)] [/<pp-name>:"<pp-signature>"] [{C-Format Code}]

- 方括号[]表示可选内容
- 尖括号<>表示规则开发者填写的可变化内容
- 加大加粗的标点符号不能省略。
- <ev-name>是一个经过%event 选项定义过的事件名称。
- <C-Format expr>被称为 Condition，是一个 C99 兼容的标量表达式。Condition 可选，没有 Condition 时表示 True。
- <pp-name>是注册的预处理器的名称，关于预处理器请参考《Programming Guide》的介绍。
- <pp-signature>被称为 Pre-Condition，是由<pp-name>预处理器负责定义、执行的预处理规则（第三方规则）。Pre-Condition 可选，不写表示 True
- <C-Format Code>被称为 Action，是一个 C99 兼容的语句段。这段代

码需要返回一个 Flase/True 型的返回值。Action 可选，不写 Action 暗含返回值是 True。

- signature_rhs 的求值方式可以用下边的伪代码表示：

```
if(Pre-Condition.Calc_Value() == False)
    return RHS-NOT-MATCH;

if(Condition. Calc_Value() == False)
    return RHS-NOT-MATCH;

if(Action.Run() == False)
    return RHS-NOT-MATCH;

return RHS-MATCH
```

- Condition 的表达式最终会被翻译成一个隐含的函数调用，例如

```
1: my_event(β);
```

其中的 Condition 经过 eyoung IPS 引擎的解析后，会在.eyc 中间代码文件中添加一个名为_condition_1_0_0 的全局函数：

```
int _condition_1_0_0(engine_work_t *_WORK_,
    engine_work_event_t *_THIS_)
{
    return (β);
}
```

该函数名称中"1_0_0"中的 1 是规则 ID,; 中间的 0 表示规则右侧 signature_rhs_list 出现的顺序，从 0 开始记；最后一个 0 表示同一个 signature_rhs_list 内 signature_rhs 出现的顺序，从 0 开始记。翻译后的 Condition 函数被添加了两个形参， _WORK_和_THIS_，分别用来指代当前被检测的 work 对象和 event 对象。所以，在 β 中可以像使用关键字一样直接使用_WORK_和_THIS_。

- 类似地，Action 最终也会被翻译成一个函数，例如对以下规则：

```
1: my_event1{β1} my_event2{δ1}
    | my_event1{β2} my_event2{δ2}
    ;
```

其中的 Action 经过 eyoung IPS 引擎的解析后，会在.eyc 中间代码文件中添加以下函数：

```
int _action_1_0_0(engine_work_t *_WORK_,
    engine_work_event_t *_THIS_)
{
    β1
}

int _action_1_0_1(engine_work_t *_WORK_,
    engine_work_event_t *_THIS_)
{
    δ1
}

int _action_1_1_0(engine_work_t *_WORK_,
    engine_work_event_t *_THIS_)
{
```

```
    β2
}

int _action_1_1_1(engine_work_t *_WORK_,
    engine_work_event_t *_THIS_)
{
    δ2
}
```

这里编号的同 Condition 函数的编号规则一致。同样地，在 action 中也允许像使用关键字一样直接使用_WORK_和_THIS_。

- 所有的 Action 和 Condition 函数，经过上述的转换后，都会被 JIT 实时编译器编译成机器指令并添加到当前进程的运行镜像中。eyoung IPS 引擎在检测过程中，会找到编译之后的 action 和 condition 函数的入口地址并执行它们。
- Action 和 Condition 可以有效提升 eyoung IPS 规则对攻击特征的描述能力。

4. 规则后记

4.1. 语法定义

```
epilogue_opt:
    empty
    | TOKEN_EPILOGUE_CODE
    ;
```

4.2. 语法说明

规则后记就是一段 C99 兼容的代码段，它是可选的。所有的规则后记代码，都会在规则解析阶段被 eyoung IPS 引擎拷贝到.eyc 中间代码文件中。