Signatures Specification

eyoung.father@gmail.com

2014/3/21

directory

1. Signature Framework

1.1. Syntax definition

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

1.2. Syntax Description

In general, eyoung IPS signature file can be divided into three parts: `signature prologue`、`signature body and signature epilogue`, separated by "%%". The three parts are optional, but the separator "%%" cannot be omitted. In general, eyoung IPS signature file is similar to the GNU yacc&bison grammar file.
For example:

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

2. Signature Prologue

The signature `prologue` is used to define important metadata information of the signature, including events, function entry, external libraries reference, external function declaration, etc.

2.1. Syntax definition

```
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:
    ;
```

Here, the prologue_opt may be nothing, which means the current signature file does not contain any prologue part, or may be a prologue_list. A prologue_list consists of at least one prologue, which is an independent prologue definition.

2.2. Prologue Code

`Prologue Code` is C format code in pairs of "%{" and "}%", and the code format is compatible with C99 standard. The prologue Code cannot be nested and all codes between "%{ and %}" will be copied to the intermediate code file for real-time compiling after loading all signature files.

For example:

```
%{
    #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 option

Format:

```
%output "file-name"
```

Introduction:

The `%output` option is used to define the name of intermediate code file, which is generated after pre-processing the signature file. The parameter of this option is a string following the C99 standard, quoted with a pair of double quotation marks. If the option is not used, the intermediate file is by default named by adding ".c" suffix behind the current signature file name. For example, if the processing signature file is "http.ey", the name of the intermediate code file is "http.ey.c" by default. If the `%output` "http.eyc" option is used, the name of the intermediate code file is "http.eyc". We recommended using this option, and setting ".eyc" as expanded-name of the intermediate code file.

Example:

```
%output "http.eyc"
```

2.4. %file-init option

Format:

```
%file-init "function-name"
```

Introduction:

The `%file-init` option is used to register a file-level constructor into the eyoung IPS engine, and the prototype of the constructor is defined in header file libengine_type.h:

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

After all the signature files are parsed, compiled and linked by the JIT compiler and linker, the constructor is called by eyoung IPS engine according to the signature files parsing order. The returning value ZERO by the constructor means successful execution otherwise means failure. When eyoung IPS engine finds initializing failure, it will stop loading signature files.

Notice: The registered initializing function will be executed after all signature files are

compiled and linked, so the registered function needs “external” attribute, which may be global function defined in a signature file, or a global function defined in the dynamic link binary library which is loaded by the `%import` option, but CANNOT be static function!

Example:

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

%file-init “my_file_init”
```

2.5. %file-finit option

Format:

```
%file-finit “function-name”
```

Introduction:

The `%file-finit` option is used to register a file-level destructor into the eyoung IPS engine, and the prototype of the destructor is defined in header file libengine_type.h:

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

When the eyoung IPS engine unloads loaded signatures, the engine will call the destructor according to the parsing order. The destructor is for releasing the resources allocated in the signatures files. The returning value ZERO by the destructor means successful execution otherwise means failure. Failure means that some resources are not released correctly, and it will lead resource leaks. The signature developers MUST examine the root reason.

Notice: Similar to the `%file-init` option, the `%file-finit` option also requires registered function with “external” attributes. Functions with “static” attributes CANNOT be used!

In addition, in the function registered by the “`%file-init`” option, we can also use **`ey_add_file_finit`** macro defined in header file libengine_export.h to register the file destructor. Functions with “static” attributes can be accepted in this usage. In practice, **`ey_add_file_finit`** is recommended.

Example:

(1):

```
%{
```

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

    %file-finit "my_file_finit"
(2): RECOMMENDED
    %{
    #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 option

Format:

```
%work-init "function-name"
```

Introduction:

The `%work-init` option is used to register a work-level constructor into the eyoung IPS engine, and the prototype of the constructor is defined in the header file libengine_type.h:

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

When an engine_work_t object is created, the eyoung IPS engine will automatically call the registered constructor. The constructor is used to allocate work-level resource and initialize the work. To get more information of the "work", you can refer to ***Programming Guide***. The returning value ZERO by the constructor means successful execution otherwise means failure. When the eyoung IPS engine finds initializing failure, the engine will failed to create the work object.

Notice: Similar to the `%file-init` option, the `%work-init` option also requires registered function with "external" attributes. Functions with "static" attributes CANNOT be used!

In addition, in the function registered by the "%file-init" option, we can also use **ey_set_userdefined_work_init** macro defined in header file libengine_export.h to register the work constructor. Functions with "static" attributes can be accepted in this usage. In practice, **ey_set_userdefined_work_init** is recommended.

Example:

(1):

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

%work-init "my_work_init"
```

(2): **RECOMMENDED**

```
%{
#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 option

Format:

```
%work-finit "function-name"
```

Introduction:

The %work-finit option is used to register a work-level destructor into the eyoung IPS engine, and the prototype of the destructor is defined in header file libengine_type.h:

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

When an engine_work_t object is destroyed, the eyoung IPS engine will call the registered destructor automatically. The destructor is used to release and clean the

work-level resources. To get more information of the "work", you can refer to ***Programming Guide***. The returning value ZERO by the destructor means successful execution otherwise means failure. Failure means that some resources are not released correctly, and it will lead resource leaks. The signature developers MUST examine the root reason.

Notice: Similar to the `%file-init` option, the `%work-finit` option also requires registered function with "external" attributes. Functions with "static" attributes CANNOT be used!

In addition, in the function registered by the "`%file-init`" option, we can also use **`ey_set_userdefined_work_finit`** macro defined in header file libengine_export.h to register the work destructor. Functions with "static" attributes can be accepted in this usage. In practice, **`ey_set_userdefined_work_finit`** is recommended.

Example:

(1):

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

%work-init "my_work_finit"
```

(2): **RECOMMENDED**

```
%{
#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 option

Format:

```
%event-init "event-name" "function-name"
```

Introduction:

The `%event-init` option is used to register an event-level constructor into the eyoung IPS engine, and the prototype of the constructor is defined in the header file libengine_type.h:

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

When an engine_work_event_t object is created, the eyoung IPS engine will call the registered constructor for the event named with "*event-name*" automatically. The constructor is used to allocate event-level resource and initialize. To get more information of the "event", you can refer to ***Programming Guide***. The returning value ZERO by the constructor means successful execution otherwise means failure. When the eyoung IPS engine finds initializing failure, the engine will failed to create the event object.

Notice: Similar to the `%file-init` option, the `%event-init` option also requires registered function with "external" attributes. Functions with "static" attributes CANNOT be used!

In addition, in the function registered by the "`%file-init`" option, we can also use **`ey_set_userdefined_event_init`** macro which is defined in header file libengine_export.h to register the constructor. Functions with "static" attributes can be accepted in this usage. In practice, **`ey_set_userdefined_event_init`** is recommended.

Example:

(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): **RECOMMENDED**

```
%{
#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 option

Format:

```
%event-preprocessor "event-name" "function-name"
```

Introduction:

After an engine_work_event_t object is created, it may be submitted to the eyoung IPS engine for attack detection for several times. The `%event-preprocessor` option is used to register an event-level pre-processing function for the event called "*event-name*" into the eyoung IPS engine, and the prototype of the pre-processor is defined in header file libengine_type.h:

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

When the event is submitted to the eyoung IPS engine, the eyoung IPS engine will call the registered function automatically to do something such as data format conversion etc. To get more information of the "event", you can refer to ***Programming Guide***. The returning value ZERO by the destructor means successful execution otherwise means failure. Pre-processing failure leads that the event submitted cannot be detected correctly by the eyoung IPS engine.

Notice: Similar to the `%file-init` option, the `%event`-preprocessor option also requires registered function with "external" attributes. Functions with "static" attributes CANNOT be used!

In addition, in the function registered by the "`%file-init`" option, we can also use **`ey_set_userdefined_event_preprocessor`** macro defined in header file libengine_export.h to do the same thing. Functions with "static" attributes can be accepted in this usage. In practice, **`ey_set_userdefined_event_preprocessor`** is recommended.

Example:

(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): **RECOMMENDED**

```
%{
#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 option

Format:

```
%event-finit "event-name" "function-name"
```

Introduction:

The `%event-finit` option is used to register an event-level destructor into the eyoung IPS engine, and the prototype of the destructor is defined in header file libengine_type.h:

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

When an engine_work_event_t object is destroyed, the eyoung IPS engine will call the registered destructor automatically. The destructor is used to release and clean the event-level resources. To get more information of the "event", you can refer to ***Programming Guide***. The returning value ZERO by the destructor means successful execution otherwise means failure. **Failure means that some resources are not released correctly, and it will lead resource leaks. The signature developers MUST examine the root reason**.

Notice: Similar to the `%file-init` option, the `%event-finit` option also requires registered function with "external" attributes. Functions with "static" attributes CANNOT be used!

In addition, in the function registered by the "%file-init" option, we can also use **ey_set_userdefined_event_finit** macro defined in header file libengine_export.h to register the event destructor. Functions with "static" attributes can be accepted in this usage. In practice, **ey_set_userdefined_event_finit** is recommended.

Example:

(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): **RECOMMENDED**

```
%{
#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 option

Format:

```
%event "event-name" "event-type"
```

Introduction:

Programming Guide shows: "An event is a non-terminal symbol generated in the LR parsing process. The name of an event is just the name of the non-terminal symbol. The type of an event is just the type of the non-terminal symbol". In fact, definition of an event is done during the design of the LR protocol analyzer. In signature files, the %event option is just used to notify the eyoung IPS engine what type and name an event is.

There are two results after executing the %event option, firstly allocating and initializing an ey_event_t object to record related information of the event in the eyoung IPS engine; secondly adding the following codes into the intermediate file after loading the signature file:

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

Example:

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

There will be definition in intermediate code file as follow:

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

2.12. %import option

Format:

```
%import "dynamic-library-name"
```

Introduction:

The section "Binary Library Signature" in ***Programming Guide***, introduces a mechanism provided by the eyoung IPS engine, with which the signature can load an external dynamic link library as binary signatures. The %import option is the entry of binary library signature. The execution of the %import option will lead to four results:

(1) The eyoung IPS engine uses the GNU library libdl to load indicated dynamic library into the current running program context;

(2) The eyoung IPS engine uses library libelf to read records in ".eyoung_type" and ".eyoung_ident" ELF sections of the loaded dynamic link library, and write them into the intermediate file;

(3) If the ".eyoung_init" ELF section exists in the loaded dynamic link library, the eyoung IPS engine will read the entry address and execute it to complete the first initialization after loading dynamic link library file;

(4) If the ".eyoung_finit" ELF section exists in the loaded dynamic link library, the eyoung IPS engine will read and save the entry address. When the eyoung IPS engine unloads the dynamic library, the eyoung IPS engine will execute the saved function and complete resource cleanup.

Example:

```
#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);
```

After loading, there will be following content in the intermediate file:

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

At the same time, the initialization function test_init will be executed.

3. Signature Body

3.1. Syntax definition

```
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. Syntax Description

1， signature

The “signatures” is made up of some “signature”s. The form of each signature is:

```
<id> : signature_pipe_list ;
```

- punctuations “:” and ”;” CANNOT be omitted.
- *Id* is a decimal integer, which is the only index of signature in the eyoung IPS engine.

2， signature_pipe_list

The signature_pipe_list is sub-signature list separated by “|” (each sub-signature is called signature_rhs_list). The relationship between each sub-signature is "OR", that any one sub-signature matching means the whole signature matching. For example:

```
1: pipe0 | pipe1 | pipe2 ;
```

The signature whose id is 1 is composed of three sub-signatures, pipe0、pipe1 and pipe2, equivalent to “pipe0 ***OR*** pipe1 ***OR*** pipe2”.

3， signature_rhs_list

A signature_rhs_list is composed of at least one signature_rhs.

4， signature_rhs

The signature_rhs is based detection element, whose format is:

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

- [...] means the content is optional
- <...> means the content is written by the developer

- red punctuations cannot be omitted if used.
- <ev-name> is an event name defined by the %event option.
- <C-Format expr> is called Condition, which is a scalar expression compatible with C99 standard. Condition is optional, and no Condition is equivalent to True.
- <pp-name> is the name of a registered preprocessor. To get more information of the "preprocessor", you can refer to ***Programming Guide***.
- <pp-signature> is called Pre-Condition, which is defined and executed by <pp-name> preprocessor. Pre-Condition is optional, and no Pre-Condition is equivalent to True.
- <C-Format Code> is called Action, which is compatible with C99 standard. This code needs return False/True. Action is optional, and no Action is equivalent to True.
- The evaluation of signature_rhs can be shown by the following pseudo-code:

```
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
```

- The expression of Condition will be translated into an implicit global function, for example:

  1: my_event(β);

  The eyoung IPS engine will add a global function named _condition_1_0_0 into the intermediate code file after parsing signature file:

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

  In the name "_condition_1_0_0", "1" is the signature ID; the first "0" is the sequence number in the signature_rhs_list, starting with 0; the second "0" is the sequence number in the signature_rhs in the signature_rhs_list, starting with 0.
  The translated Condition function contains two parameters, _WORK_ and

THIS. The parameter _WORK_ is for the work object and the parameter _THIS_ is for the event object. So, signature writers can use _WORK_ and _THIS_ directly in the Condition part like using keywords.

- Similarly, the Action part is also translated into a function, for example:

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

The eyoung IPS engine will add global functions into the intermediate code file after parsing signature file:

```
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
}
```

Signature writers can also use _WORK_ and _THIS_ directly in the Action part like using keywords.

- All the Action and Condition functions will be compiled to the target machine instructions by the JIT compiler and loaded into the current running process context by the JIT loader. In the process of attacking detection, the eyoung IPS engine will find the entry address of compiled Action and Condition functions and execute them.

4. Signature Epilogue

4.1. Syntax definition

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

4.2. Syntax Description

The signature `epilogue` is a code segment whose format is compatible with C99 standard. This part is optional. All the codes in `epilogue` will be copied into the intermediate code file in the signature parsing stage.