

Programming Guide

`eyoung.father@gmail.com`

2014/4/5

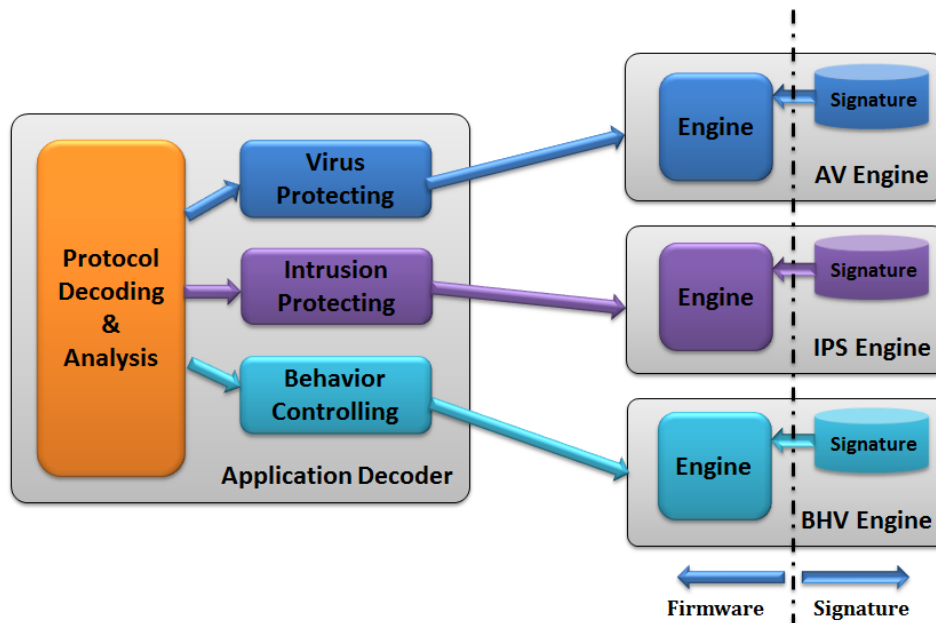
Directory

1.	Summary & Concept.....	2
1.1	Why write the eyoung	2
1.2	Design of the eyoung IPS Engine	5
1.2.1	Modules.....	5
1.2.2	Loading Signatures.....	6
1.3	The Design of Protocol Decoder	7
1.3.1	Protocol Decoder and Attack Detection.....	7
2.	The IPS Engine	8
2.1	Main Data Structure	8
2.1.1	Source Code.....	8
2.1.2	Directions	9
2.2	External API	11
2.2.1	Source Code.....	11
2.2.2	Directions	12
2.3	Signature API	18
2.3.1	Source Code.....	18
2.3.2	Directions	20
2.4	Binary Signature Package.....	20
2.4.1	Binary Signature.....	20
2.4.2	Source Code.....	21
2.4.3	Directions	22
2.5	Examples.....	23
2.5.1	demo_http_xss.c	23
2.5.2	http_decode.c.....	27
2.5.3	export_test.c.....	31
3.	The L7 Decoder.....	32
3.1	Stream-mode Lexer	32
3.1.1	Design	32
3.1.2	Examples:.....	33
3.2	Push-Mode Parser	37
3.2.1	Design	37
3.2.2	Examples.....	37
3.3	Event.....	39
3.3.1	Type of Event	39
3.3.2	Submitting Event.....	40

1. Summary & Concept

1.1 Why write the eyoung

The chart below illustrates the common L7 (Application Layer) architecture. Here taking the open source IPS, the Snort, as an example.



The dash line in this chart marks the boundary between the Firmware and the Signature. The Firmware is featured as relatively stable without frequent upgrade and its upgrade period may be counted per month(s). The Signatures are usually those frequently updated contents such as IPS signatures and virus database etc. whose upgrade period are usually counted per day(s) or hour(s). In such architecture, there usually is a general-purposed L7 protocol analyzer (the Application Decoder) for carrying out the analysis of L7 data. Besides, this framework usually contains a tailored and upgradable signature package. The Engine in the Firmware and external upgradable IPS signatures constitute the entire IPS Engine. The widely-used signature in open source IPS is currently represented by the Snort, which takes the string matching and the regular expression matching as its main detection method. Such description method is featured as easy and suitable for hardware acceleration whereas its disadvantage is also remarkable, i.e. very poor description capability:

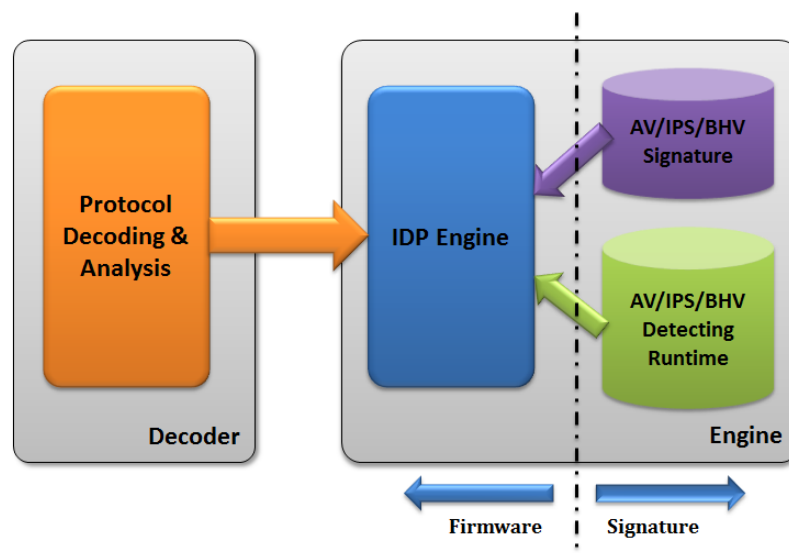
First, many kinds of attacks, among which the buffer overflow attack may be deemed as a most typical one, cannot be described by regular expression. It is obviously that regular expression cannot contain all the combinations of machine instructions, leading to both false-positive alarm and false-negative alarm very seriously.

Second, regular expression does not have an accurate description capability, e.g. SQL injection attack. Regular expression can only describe limited partial SQL

segments. However, concerning the SQL language which is a typical Context Free Language, regular expression is theoretically unable to describe, which therefore will definitely cause false-positive alarm and false-negative alarm. The DDoS attack is another typical intrusion that cannot be described by regular expression.

Last, regular expression is unable to describe the work flow. Due to the very fast changes of L7 data, e.g. webpage framework and form structure in a webpage, the general-purposed IPS signatures are unable to catch the changes in the work level. For purpose of describing the work under such IPS architecture, parsing the data in a given work flow may only be implemented in a tailored L7 decoder, which as a result makes the firmware design redundant and non-universal and also a negative effect on the overall performance. Because of the never-ending changes and improvement of internet industry, the firmware upgrade thus is likely to be left far behind of the work-flow changes, which makes the R&D of firmware much more complicated and difficult.

For purpose of rendering intrusion and work flow describable, the traditional L7 processing architecture is modified by the eyoung as below:



The eyoung system contains two basic modules: protocol analyzer (Decoder) and IPS engine (Engine). The Decoder is merely a general-purposed L7 protocol analyzer and it does not paraphrase the semantics of work flow, but only finish parsing the L7 data according to the RFC documents. The Engine is aimed at attack detection, responsible for loading signatures and detecting attacks, whose input comes from the result of the Decoder. The distinguishing feature of the eyoung IPS signature is the excellent programmability:

First, the eyoung IPS signatures are completely programmable. The eyoung supports C99-compatible grammar, namely that eyoung makes it possible for signature developers, by using C99-compatible grammar, to embed complicated work-flow description codes or attack detection codes into signatures.

Second, the eyoung IPS Engine is embedded with a JIT compiler which is able to compile the embedded codes into machine instructions directly and make dynamic



linking with current process context, thus remarkably improving the performance of execution and detection.

Last, the eyoung supports importing binary signatures dynamically. For example, it is able for a complicated work-flow processing module by the form of the Dynamic Linkage Library (.so file in Linux), such as webmail parsing, to be dynamically loaded into the eyoung IPS Engine as well as to be upgraded singly. This feature simplifies the implementation of L7 Decoder and significantly realizes the work-flow separation, while on the other hand, also extend the description capabilities and increase the developing response speed concerning the frequent changes of L7 work flow.

The design concept of the eyoung is to stabilize the Firmware side as much as possible while leaving all the quickly-changed part to the Signature side so as to simplify and extend the IPS architecture.

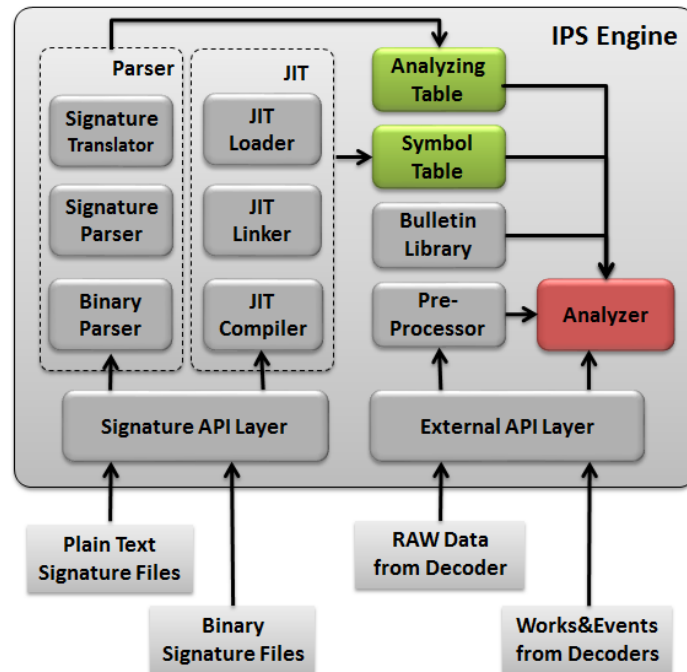
The eyoung, from the perspective of software module, is divided into two basic modules: the IPS engine and the Protocol Analyzer, which strictly keep a relationship of single-way-calling. The Protocol Analyzer calls for relevant functions of the IPS engine. There are clear APIs between them. The APIs between the Protocol Analyzer and the IPS Engine is called the **External API Layer**.

From the perspective of executing, the eyoung is divided into loading stage and detection stage:

-  Loading stage mainly involves the loading of static contents, such as IPS signature files and binary signature packages. This stage is similar as the boot stage of an operation system. The eyoung IPS signature supports high programmability. The relevant functions in the eyoung IPS Engine called by the signatures are realized by APIs, whose name is the **Signature API Layer**.
-  Detection stage mainly involves attack detection. The protocol analyzer receives L7 data, carries out protocol analysis and submits the analysis result in a systematic way, by using the External API Layer, to the eyoung IPS Engine for attack detection.

1.2 Design of the eyoung IPS Engine

1.2.1 Modules

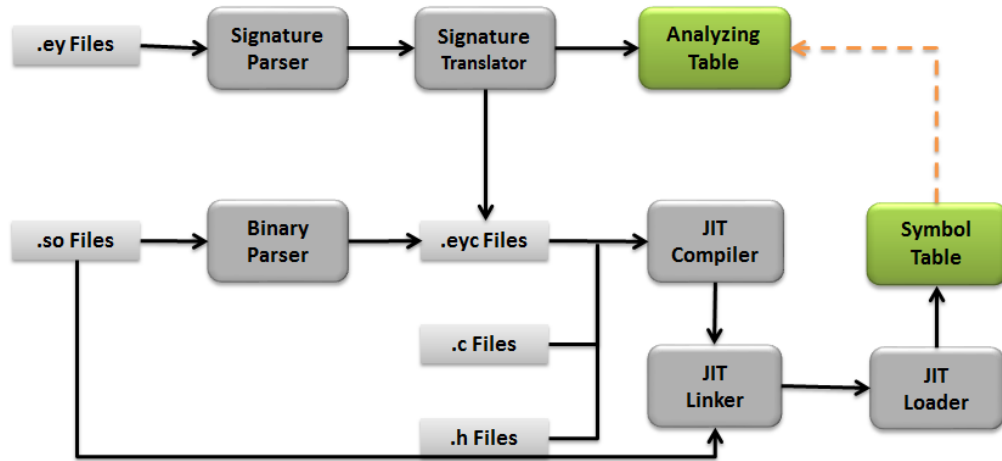


The eyoung IPS Engine mainly includes the following modules:

1. IPS signature parser (Parser), whose work is to parse the signature file, not only including the signature in Plain Text form, but also the binary signature package. Its main purpose is to:
 - a) Check the validity of signature and report grammar mistakes.
 - b) Preprocess the signature files (usually .ey files) and translate them to intermediate code file (usually .eyc files). The intermediate code is the C code source file which is C99 compatible.
 - c) Extracting the information in signature files, such as the definition of event, function entry, into certain data structures and store them into the Symbol Table and Analyzing Table.
2. JIT compiler (JIT), whose work includes: to in-time compile, link and load the intermediate code files preprocessed by the Parser; to save the global function entry address and global variable address into the Symbol Table.
3. The function in Bulletin Library, whose work is to provide basic operation, including memory management, basic data structure encapsulation, etc.
4. Pre-processor, an abstract interface layer, by which we are able to integrate third-party heterogeneous and asynchronous IPS engine into the eyoung system. Usually, the pre-processor does not process the output from the Protocol Decoder whereas its main processing object is the raw data which has not been protocol-analyzed.
5. Analyzer, the CORE of the eyoung IPS Engine during execution, responsible for the

attack detection regarding to the analyzed L7 data and detection signatures.

1.2.2 Loading Signatures



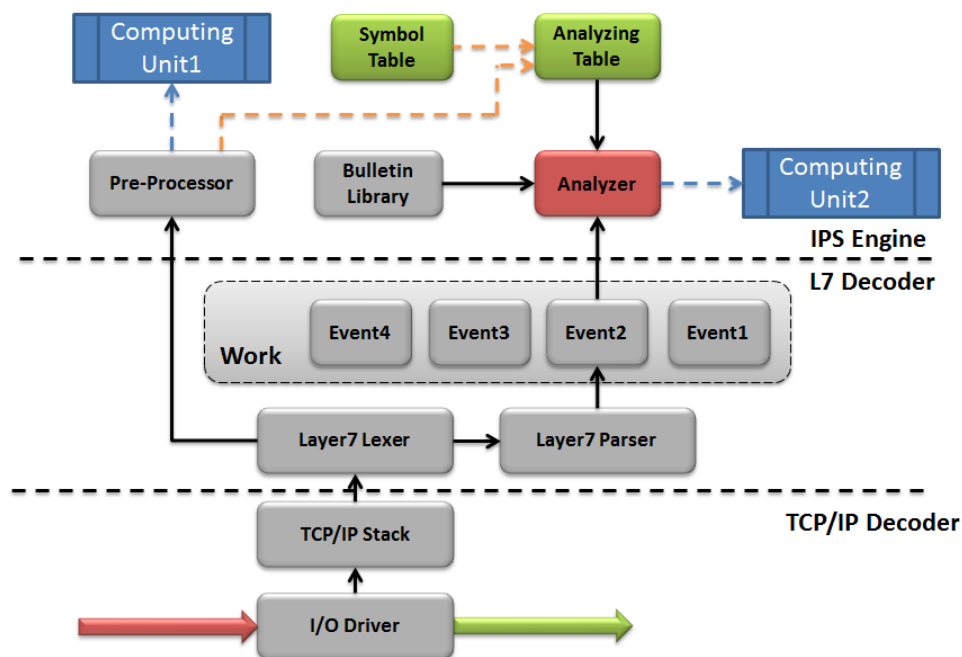
1. For the plain-text signature file:
 - a) The Signature Parser is to carry out signature validity checking.
 - b) The Signature Translator is to: translate the signature into intermediate code file (the intermediate code is C99 compatible, as aforementioned); extract the signature semantics information in signature files into the Analysis Table.
 - c) The Signature Translator is also to record the *init/finit* function entries into the Analysis Table. The detailed is to be seen in section 2.3 of this document.
2. For the binary signature file:
 - a) The Binary Parser is to read the contents of two sections in the ELF file if existing, *.eyoung_ident* and *.eyoung_type* and save these readable contents into intermediate code file.
 - b) The Binary Parser also reads the contents of *.eyoung_init* section in the ELF file, and runs the initialized entry function to complete the initialization of binary signature package.
 - c) The Binary Parser is also reads the contents of *.eyoung_finit* section in the ELF file, and saves the destructor function entry address for cleaning up resources when unloading the binary signature package.
3. All signature files are to be translated into intermediate code file after the 1 and 2 aforementioned. Then the JIT Compiler is to take charge of the left work of signature loading:
 - a) The JIT Compiler is to compile the intermediate code file as well as other header files (.h files) and source file (.c files), and to complete the file unit compiling in the memory.
 - b) The JIT Linker is to link the compiled file units. By the processing of the JIT Linker, global symbols in different signature files can be relocated and mutually referred.
 - c) The JIT Loader is to load the linked code into current program context and dynamically link the global symbol in signature codes, which enables the

signature to directly call global functions and global variable, which may be in other binary library. This feature strengthens the interaction between the Firmware and the Signature, and substantially enhances the description capacity of the eyoung IPS signature. For example, some complicated and time-consuming protocol analyzing functions may only need to be finished under certain conditions. Therefore, these analyzing functions are suitable to implement as the form of global functions in the Protocol Decoder, but not be called by the Decoder. We can write signatures to call these functions only under the needed conditions.

- d) The loaded global symbols and their addresses are saved in the Symbol Table.

1.3 The Design of Protocol Decoder

1.3.1 Protocol Decoder and Attack Detection



1. The TCP/IP Decoder, a low-layer protocol stack, is responsible for parsing the L3/L4 (network and transport layer) data, which is **NOT** included in the eyoung system. It is of network function of the operation system kernels or similar with the Intel DPDK, a kind of user-mode data forwarding framework.
2. The L7 Decoder, for analyzing the application layer data, is composed of two parts:
 - a) The L7 Lexer, which is to interact with low-layer protocol stack and maintain the raw L7 data buffers. It de-composes the L7 data into a series of atomic tokens and submits these tokens into the L7 Parser. Meanwhile, it submits the data acquired in the low-layer, via the External API Layer into the Pre-processor. The L7 Lexer is in Stream-Mode, which is to be detailed in part 3.1.
 - b) The L7 Parser, which is to check the syntax in a bottom-to-top way for the

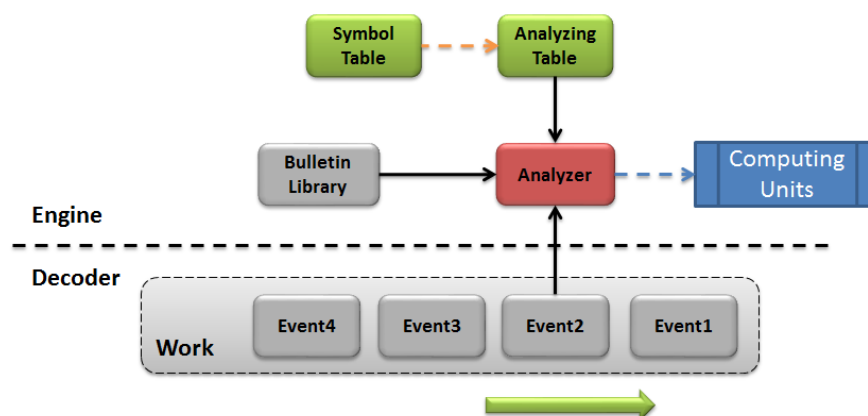
received tokens in order to: explain the L7 data from the logic perspective, submit the parsing results to the eyoung IPS Engine in the form of abstract data type, such as work, event, etc. The definition of “work” and “event” is to be detailed in section 2.1 and 3.3. The L7 Parser, which is of Push-mode, is to be detailed in section 3.2.

3. The Pre-processor, which is compatible with third-party IPS engines and signature formats. The Pre-processor is expected to be executed asynchronously on other computing units, such as other CPU, ASIC chip or GPU, etc. The results by the Pre-processor are to be recorded in the Analyzing Table and used by the Analyzer.
4. The Analyzer, the core component of the eyoung IPS Engine during detection stage, is to detect attack from objects such as “work” and “event” submitted by L7 Parser and the Pre-processor.

2. The IPS Engine

2.1 Main Data Structure

The most important data structures of the eyoung IPS Engine are all defined in header file *include/libengine_type.h*. There are mainly three defined data structures in this file: *engine_t*, *engine_work_t* and *engine_work_event_t*. The *engine_t* type is for a static concept, representing the eyoung IPS Engine itself. The *engine_work_t* type and the *engine_work_event_t* type are for dynamic concepts, representing the run-time detecting objects.



2.1.1 Source Code

```
#include "ey_queue.h"

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

```

typedef int (*file_init_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 Directions

1. engine_t

This type abstracts of the eyoung IPS Engine itself. When any external modules need to operate the eyoung IPS Engine, such as loading/unloading signature files, the relevant APIs always need the *engine_t* instance.

2. file_init_handle, file_finit_handle

They are of function pointer types. The corresponding function needs to be registered into the eyoung IPS Engine using two macros, *ey_add_file_init* and *ey_add_file_finit* as defined in section 2.3.1. There are still two keywords, *%file-init* and *%file-finit* could do the same thing.

The eyoung IPS Engine is able to load multiple signature files. At most one

file_init_handle function and one *file_finit_handle* function could be defined in each signature file. The call time of *file_init_handle* function is after **ALL** the signature files finish being loaded. The eyoung IPS Engine calls the registered init function in turn according to loading sequence to finish the file-level initialization. The *file_finit_handle* function is called when the eyoung IPS Engine is going to unload the signatures. The finit function will be called automatically also according to the loading sequence to finish the file-level resource cleaning-up.

3. engine_work_t

The *engine_work_t*, abbreviated as “work”, is a runtime concept. The work is composed of a series of events and is to indicate an IPS detection process. For example, a TCP connection could be taken as a work.

- ✚ A link list is formed by work through the member “*link*”. The header of the list is recorded in *engine_t* object.
- ✚ The member “*engine*” points at the *engine_t* object of this work.
- ✚ The member “*priv_data*”, is not allowed to be modified by the external except for the engine. This member points at the data work in the engine for private usage.
- ✚ The member “*predefined*” points at an object defined by the L7 Decoder developer. Since the work object is created as “link” concept, if the developer wants to record some information of the “link” (e.g. the linked counting information), it could be implemented by this member. The detailed format definition is at the disposal of protocol developer, which could not be comprehended by the eyoung IPS engine.
- ✚ The member “*user_defined*” points at an object defined by signature developers. When developing the L7 Decoder, it is impossible to predict all later detection requirement. If signature developers want to save something into the work, this member is just for such need. Its definition of format, operation of data object is all to be finished by the signature developers. The eyoung IPS Engine does not comprehend its structure (only concerned with the entry addresses for creating and destroying).

4. work_init_handle, work_finit_handle

They are pointer types at functions. The corresponding function instance is called by the eyoung IPS Engine when a work is created or destroyed. The instance of *work_init_handle* type is called to initialize data in work, while the instance of *work_finit_handle* type is called to clean up resource in work. The instances of these functions are all defined by protocol analyzer developers and signature developers and are registered into the eyoung IPS Engine by relevant API of section 2.3.1. Specifically, there are two macros for the L7 Decoder developer to register the function instance of the construction and destruction of work into the eyoung IPS Engine: *ey_set_predefine_work_init* and *ey_set_predefine_work_finit*. There still are two macros for signature developers to register the function instance of the creation and destruction of work into the eyoung IPS Engine: *ey_set_userdefine_work_init* and *ey_set_userdefine_work_finit*, or two keywords: *%work-init* and *%work-finit*.

5. engine_work_event_t

Abbreviated as “event”, it is the data type abstracting atomic detection element. Similar with work, event is also a runtime concept. A work is composed of a series of events. An event can be an HTTP URI, or an invalid DNS request etc. The creation or destruction of event can only be done in L7 Decoders now.

- ✚ The member *work*, pointing at the work object to which the event belongs.
- ✚ The member *event*, pointing at the type definition object. This object is of a static concept and is created by the keyword %event during signature loading. This member cannot be modified out of the event.
- ✚ The member *predefined*, pointing at an object defined by the L7 Decoder developers. Like the member *predefined* of the work object, the definition here is completely interpreted by L7 Decoder developers whereas the eyoung IPS Engine cannot recognize and operate its structure.
- ✚ The member *user_defined* is a pointer at an object defined by signature developers. Like the member *user_defined* of the work object, this member is for supplementing the result of the L7 Decoder. For example, regarding HTTP URI, after being parsed simply as string through the L7 Decoder, a certain parameter in URI may be matched and filtered in a signature. Parameters in URI can be considered as “key-value” pairs. If the “key-value” pair is always parsed in the stage of protocol analyzing, it might be a waste of performance. By then the wanted parameter parsing can be carried out just for special URI. Tailored parsing function is provided by this member. The parsing result can be saved in this member.

6. **event_init_handle, event_finit_handle, event_preprocess_handle**

They are pointer types at functions. The corresponding function instances are called by the eyoung IPS Engine while event is created, destroyed and detected. They are mainly used to allocate, release and pre-process. The instances of these functions are respectively defined by L7 Decoder developers and signature developers and are registered into the eyoung IPS Engine through relevant APIs. Please refer to the section 2.8, section 2.9 and section 2.10 in [*Signatures Specification*](#).

7. **event_condition_handle, event_action_handle**

They are pointer types at function. They are internally used by the engine while loading signature files. Refer to [*Signatures Specification*](#).

2.2 External API

External APIs are for the L7 Decoder developers. They are defined in the header file *include/libengine_function.h*.

2.2.1 Source Code

```
#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 Directions

1. *ey_engine_create*

Purpose:

This function is for creating an instance of the eyoung IPS Engine.

Parameter:

The parameter *Name* is the name string, ended with “\0”. Duplication of name is not to be checked in the eyoung system.

Return Value:

Pointer at `engine_t` type will be returned if successfully created, otherwise "NULL".

Examples:

See 2.5.1

2. `ey_engine_destroy`**Purpose:**

This function is for destroying an instance of the eyoung IPS Engine

Parameter:

engine: the return value of function `ey_engine_create`.

Return value:

None.

Examples:

See 2.5.1.

3. `ey_engine_load`**Purpose:**

Load signature files after creating an instance of the eyoung IPS Engine.

Parameter:

engine: the return value of function `ey_engine_create`

files: char pointer array, for passing the paths of signature files to be loaded.

file_num: number of members in array *files*.

Return Value:

"0" for success, otherwise "-1".

Examples:

See 2.5.1.

4. `ey_engine_find_event`**Purpose:**

Find event index value according to given event name. This function is usually called by L7 Decoders. The defining of an "event" is completed in signature files. For the detailed format of event, see [Signatures Specification](#). An "event" has its name, and duplicated names are not allowed within one instance of the eyoung IPS Engine. Data-flow from L7 Decoders to the eyoung IPS Engine is all realized by sharing the event name. For convenience, the name of event is replaced by the index of event in the eyoung IPS Engine. Function `ey_engine_find_event` is to look for the event index value for the given event name.

Parameter:

engine: the return value of function `ey_engine_create`

event_name: event name string, ended with "\0".

Return Value:

">=0" for success, indicating the event index value, otherwise "-1".

Example:

“libdecoder/http/http_server.y”:

```
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++)
    {
        constchar*name = yytnam[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);
    }
}
```

*/*Among the code generated by the bison, yytnam array saves all symbol's names while the macro YYNTOKENS saves all indexes of nonterminal symbols. In parsing process of “shifting-reducing”, the reduced nonterminal symbols are to be submitted in turn to the eyoung IPS Engine for further detection. The nonterminal symbols therefore are the “events”. All nonterminal symbols shall be written in IPS signatures so as to enable the IPS engine’s awareness of the existence of the “event”. This function is to be called when HTTP L7 Decoder initializes after signature loading. Event index is searched through calling ey_engine_find_event and is saved in array yytid. */*

5. ey_engine_work_create

Purpose:

Create an instance of the type *engine_work_t*. While creating the instance of the type *engine_work_t*, the initializing functions which are always registered through the function *ey_set_predefine_work_init*, the function *ey_set_userdefine_work_init* and the keyword *%work-init* are called by the eyoung IPS Engine to complete the initializing work.

Parameter:

engine: the return value of *ey_engine_create*

Return Value:

A pointer at the type *engine_work_t* for success, otherwise “NULL”.

Example:

See 2.5.2.

6. *ey_engine_work_destroy***Purpose:**

Destroy an instance of *engine_work_t*. While destroying the instance of the type *engine_work_t*, finalization functions which are always registered through the function *ey_set_predefine_work_init*, the function *ey_set_userdefine_work_init* and the keyword *%work-init* are called by the eyoung IPS Engine to release resources.

Parameter:

work: the return value of function *ey_engine_work_create*

Return Value:

None.

Example:

See 2.5.2.

7. *ey_engine_work_create_event***Purpose:**

This function is usually called by L7 Decoders for creating an event instance to be submitted to the eyoung IPS Engine. In this process, initial functions registered through *ey_set_predefine_event_init*, *ey_set_userdefine_event_init* and *%event-init* are called to complete initializations of event.

Parameter:

work: the return value of function *ey_engine_work_create*

event_id: event index value returned by function *ey_engine_find_event*

action: output parameter, for describing the detection result of event

Return Value:

A pointer at the type *engine_work_event_t* for success, otherwise “NULL”.

Example:

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;
}

```

8. *ey_engine_work_destroy_event*

Purpose:

Destroy an event instance and release relevant resources. While destroying the instance of type *engine_work_event_t*, functions which are registered through *ey_set_predefine_event_finit*, *ey_set_userdefine_event_finit* and *%event-finit* are called by the eyoung IPS Engine.

Parameter:

event: the return value of function *ey_engine_work_create_event*

Return Value:

None

Example:

Refer to *ey_engine_work_create_event*.

9. *ey_engine_work_detect_event*

Purpose:

Submit a created event instance to the eyoung IPS Engine for detection. The preprocessing functions which are registered through the function *ey_set_predefine_event_preprocessor*, the keyword *%event-preprocessor* and the function *ey_set_userdefine_event_preprocessor* are called by the eyoung IPS Engine to complete the preprocessing of event data.

Parameter:

event: the return value of function *ey_engine_work_create_event*

predefined: initial value of event

Return Value:

0 for clean event , -1 indicating attack data in event.

Example:

Refer to *ey_engine_work_create_event*.

10. *ey_engine_work_detect_data*

Purpose:

Differing from the function *ey_engine_work_detect_event*, data submitted by this function is for detection by the preprocessor in the eyoung IPS Engine. The preprocessor may be seen as third-party paralleled detection unit, whose detection result may be taken as the premise of the eyoung IPS Engine's event analysis. It can be finished asynchronously (now is synchronously).

Parameter:

work: the return value of *ey_engine_work_create*
data: raw data to be detected
data_len: the length of buffer area to be detected
from_client: whether raw data comes from the client side

Return Value:

It always returns value 0 now.

Example:

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);
}
```

11. *debug_engine_init*

This global variable is a debug switch for the debug information in the initialization stage of the eyoung IPS Engine. This switch may affect functions: *ey_engine_create*, *ey_engine_destroy* and *ey_engine_find_event*.

12. *debug_engine_lexer*

This global variable is a debug switch is for the debug information of the lexer in the eyoung IPS Engine when loading signature files. This switch may affect function *ey_engine_load*.

13. *debug_engine_parser*

This global variable is a debug switch is for the debug information of the parser in the eyoung IPS Engine when loading signature files. This switch may affect function *ey_engine_load*.

14. *debug_engine_compiler*

This global variable is a debug switch is for the debug information of the JIT compiler and pre-processor in the eyoung IPS Engine when loading signature files. This switch may affect function *ey_engine_load*.

15. *debug_engine_runtime*

This global variable is a debug switch is for the debug information in the detecting stage of the eyoung IPS Engine. This switch may affect functions:

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.

2.3 Signature API

The set of Signature APIs can be directly called in the eyoung signatures, and they are all defined in the header file *include/libengine_export.h*. Signature APIs are mainly to help signature developers to register some user-defined function (how to initialize, finalize and preprocess for file, work and event, etc.).

2.3.1 Source Code

```
#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 Directions




As mentioned in section 2.2.2, these APIs are not to be detailed here. Readers can refer to examples:

-  libdecoder/http/testsuite/http_xss.ey
-  libdeocder/pop3/testsuite/pop3.ey
-  libdecoder/html/testsuite/html.ey

2.4 Binary Signature Package

2.4.1 Binary Signature

The eyoung IPS Engine supports high signature-level programmability, of which an important feature is that the engine supports importing third-party dynamic linking library file in the eyoung IPS signatures. This feature is similar with the loadable kernel module (.ko module files) in the Linux kernel. The purpose of such feature is to keep a balance among extensibility, upgradability and performance, which involves three problems to solve:

1. Sharing symbols. The eyoung IPS Engine has to be able to perceive enough symbols of external dynamic linking library, including:
 -  Global variables: name, type and relocated virtual address
 -  Global functions: name, entry address, returning type and parameter list information
 -  Raw type definition.

There are two solutions available for this problem: using header file directly, or writing the raw readable definition into ELF dynamic linking library ELF file. Both the two solutions are supported by the eyoung, and two built-in macros, *EY_EXPORT_IDENT* and *EY_EXPORT_TYPE*, are for the 2nd solution.

2. Dynamically loading and unloading the external dynamic linking library files. The keyword %import is implemented in the eyoung signature grammar for the work.
Note: The unloading work is usually automatically done in process of signature upgrading by the eyoung IPS Engine.
3. Automatically initializing, e.g. the resource allocated, global variables initialization. Inspired by the Linux kernel design of loadable kernel module, the eyoung writes the entrance information of relevant *init/finit* functions into the ELF dynamic linking library file through two macros: *EY_EXPORT_INIT* and *EY_EXPORT_FINIT*. In process of loading the binary signature package, the eyoung IPS Engine is to analyze the ELF file, get the entry address and automatically execute the init function. The finit function is executed when unloading signatures.

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 Directions

1. *EY_EXPORT_IDENT*

This macro can help the developers write the declarations of global variables and global functions, in the form of raw readable string, into the *.eyoung_ident* section in the ELF dynamic linking library file. When executing the keyword *%import* in the eyoung IPS signature files, the eyoung IPS Engine will try to read the contents saved in the *.eyoung_ident* section (if exists) and write the saved declarations of symbols directly into the intermediate code files.

2. *EY_EXPORT_TYPE*

This macro is to write the raw c-format type definitions into *.eyoung_type* section in the ELF dynamic linking library file. When executing the keyword *%import*, the eyoung IPS Engine will try to read the saved contents in the *.eyoung_type* section (if exists) and write the type declarations directly into the intermediate code files.

3. *EY_EXPORT_INIT*

This macro is to write a user-defined function entry into the *.eyoung_init* section in the ELF dynamic linking library file. When executing the keyword *%import*, the eyoung IPS Engine will try to read the saved contents in the *.eyoung_init* section (if exists), and execute the function and automatically complete the initialization work.

4. *EY_EXPORT_FINIT*




This macro is to write a user-defined function entry into the *.eyoung_finit* section in the ELF dynamic linking library file. When executing the keyword *%import*, the eyoung IPS Engine will try to read the saved contents in the *.eyoung_finit* section (if exists), and save the entry address of the finit function into the IPS engine. The eyoung IPS Engine will execute the function when it unloads signatures and automatically complete the finalization work to release resources allocated in the dynamic linking library.

See 2.5.3 for examples of above functions.

2.5 Examples

2.5.1 demo_http_xss.c

This section is to show the usage of:

-  `ey_engine_create`
-  `ey_engine_load`
-  `ey_engine_destroy`.

`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

This example is to show the usage of:

- + ey_engine_work_create
- + ey_engine_work_detect_data
- + ey_engine_work_create_event
- + ey_engine_work_detect_event
- + ey_engine_work_destroy_event

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

This example is to show the usage of:

- ✚ EY_EXPORT_IDENT
- ✚ EY_EXPORT_INIT
- ✚ EY_EXPORT_TYPE
- ✚ EY_EXPORT_FINISH.

test/export_test.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_FINISH(test_exit);

```

3. The L7 Decoder

The L7 Decoder analyzes L7 data and submits the hierarchical and structured results to the eyoung IPS Engine in the form of protocol event for further analysis and detection according to the signatures. For such purpose, the L7 Decoder can be divided into lexer and parser, which is very similar with a common compiler.

3.1 Stream-mode Lexer

3.1.1 Design

The lexer is much closer to the bottom layer, and always interactions with network data directly. Its main function is, to save and buffer the network data, and split the raw message into a series of tokens according to analysis rules like a common

compiler. The difference is that the compiler's operations always on the local disk files are through the "blocking" I/O function (e.g. fread and fscanf, etc.). However, the "blocking" method is inapplicable for high-performance lexer and the analysis must be carried out in the "stream". By amending the usage of flex, GNU software, the eyoung provides a non-blocking stream-mode solution below:

- 1) Modification on flex's automatic management of input buffer. Regarding the unmatched part of input buffer, e.g. the ending position of the current input buffer only partially matching with a mode, the eyoung lexer is to copy and save the unmatched partial part instead of reporting a defaults result in standard flex. Buffer reassembling and re-matching will be carried out for next input buffer.
- 2) Greedy mode matching. In greedy mode, if the ending part of an input buffer just matches with a pattern, the eyoung lexer is to return the matched mode directly; otherwise, the eyoung lexer is to save the last matched contents as per the 1), and buffer reassembling and re-matching will be carried out for next input buffer. This design is due to that there may be some pattern-including relationship to be matched. For example, pattern "b" and pattern "ba" are pattern-including relationship. For an input segment ending with "b", pattern "ba" may be matched (the next buffer begins with "a"). In greedy mode, the eyoung lexer is to return the first pattern "b". Otherwise, the ending part "b" is to be saved and buffer reassembling and re-matching will be carried out for next input buffer.

Note: The tool flex is used by the eyoung lexer. In flex, the greedy mode is used by flex cannot be closed except for modifying its source code, because the flex is not considered for the stream input. The non-greedy mode of the eyoung just regards with the ending part of current input buffer and doesn't applied for none-ending part of the input buffer.

3.1.2 Examples:

```
%{
#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?yylen-1:yylen;
    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;
}

```

The function *http_server_scan_stream* is used to reassemble the input buffer. If

no any content is saved for last input buffer, the current input buffer is taken as the scan buffer. Otherwise a new buffer is allocated, whose length is equal to the length of saved contents plus the length of current input buffer. And the two buffers are to be merged into the newly allocated buffer. An optimization is available here that it only needs to save the matching state of the last input buffer rather than the ending data, thus avoiding data copy.

The function *save_server_stream_context* is called by *YY_UUSER_ACTION* and *YY_RESTORE_YY_MORE_OFFSET*. The macro *YY_USER_ACTION* will be called when a pattern is matched. As introduced in section 3.3.1, it shall be further checked whether the current matching position is at the ending of current input buffer and whether the greedy mode is activated. If so, the matched ending part is to be saved and the lexer will return *TOKEN_CONTINUE*. Its return value *TOKEN_CONTINUE* is passed to the upper parser. The macro *YY_RESTORE_YY_MORE_OFFSET* will be called when the lexer reaches to the end of current input buffer without finding any matched pattern. In this situation, the flex turns to do default operation. As the eyoung lexer is stream-based, it may not lead to a conclusion that no pattern is matched. And if the current input buffer is not the last segment in the stream, the ending unmatched part will be saved and the lexer also returns *TOKEN_CONTINUE* to notify the upper parser.

3.2 Push-Mode Parser

3.2.1 Design

The core of the eyoung L7 Decoder is the parser, which carries out the decoding work under the guidance of protocol grammar rules (usually a Context Free Language). The parser is to check whether the protocol data complies with the protocol specification on both grammar and semantics. On the other hand, the parser is also to automatically submit the parsed syntax information into the eyoung IPS Engine for further attack detection.

Traditional parser generated by the GNU bison, usually works in “PULL” mode, i.e. the parser atomically calls the lexer to get the next token, and the lexer is to do the file and buffer operations. This is suitable for the blocking-mode lexer whereas the blocking feature will lead to very poor performance for network data parsing. By applying new feature of the tool bison, the eyoung provides Push-Mode parser as a solution for this problem. In Push-Mode parser, the lexer is called before and outside the parser. The token acquired by lexer is pushed into the parser.

As the lexer may copy and save some ending content of current input buffer and return *TOKEN_CONTINUE*, the parser must handle *TOKEN_CONTINUE*.

3.2.2 Examples

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;
}

```

The function *parse_http_server_stream* is called after receiving a raw network buffer <buf, buf_len> in bottom layer.

- ✚ The function *http_server_scan_stream* is for reassembling the buffer used by lexer, as detailed in 3.1.2.
- ✚ The function *http_server_delete_buffer* is for destroying the buffer, which is the API generated by the flex. Please see manual of the flex for details.
- ✚ The function *http_server_lex_body_mode* returns a Boolean-type value and is for judging whether there is an HTTP body being parsed in the lexer. Since the HTTP body parsing relays on the length information, either defined in the header *CONTENT-LENGTH* or defined as *CHUNKED* in the header *TRANSFER-ENCODING*. Such counting-based parsing is not suitable for the regular expression based tool flex. Therefore, the eyoung HTTP lexer carries out the work separately and respectively for BODY part and Non-BODY part. The eyoung HTTP lexer parses the BODY part by using the function *http_server_body_lex* when the function *http_server_lex_body_mode* returns TRUE. Otherwise, the function *http_server_lex* is used for parsing the Non-BODY part. The function *http_server_lex* is atomically generated by the tool flex.
- ✚ In the section 3.1.1, both the function *http_server_body_lex* and the function *http_server_lex* may return *SYM_CONTINUE*. By then, the parsing process shall run out of while loop.
- ✚ Generated by bison automatically, the function *http_server_push_parse* is the entry of the Push-Mode Parser, whose parameter is the token returned by the lexer. If returning ZERO, it means the input token has been totally accepted by parser. If returning YYMORE_PUSH, it means that the current input token complies with the grammar rules but needs more coming tokens to forward the parser's state until it returns 0 or finds some mistake. Any other return values mean errors, grammar error, memory exception etc.

3.3 Event

3.3.1 Type of Event

The concept of event is finished in *Signatures Specification*. This section is mainly about the type and submitting mechanism of the eyoung event.

In bottom-to-top LR analysis, which is a shifting-reducing method for the Context Free Language, tokens acquired by the lexer is called "Terminal Symbol" whereas symbols generated in reduction process is called "Non-Terminal Symbol". In fact, an event in the eyoung is just the Non-Terminal Symbol generated in LR analysis. Along with the analysis process, the parser will generate a series of Non-Terminal Symbols, which in the concept shall surely be "bottom-to-top", i.e. from micro concept to macro concept. Thus the eyoung IPS signatures acquire the capability of describing the protocol content in different concept layers.

The name of event is just the name of the Non-Terminal Symbol. The type of the

Non-Terminal Symbol can be defined by applying *%union* and *%type* in bison. For example, the below definition from `http_server.y`:

```
%union
{
    .....
    http_response_header_t *header;
    .....
}
.....
%type <header> response_header_host
.....
```

These codes mean that the type of the Non-Terminal Symbol `response_header_host` in the grammar definition is a pointer at the type of `http_response_header_t`. In eyoung signature files, we have to add following statements to notify the eyoung IPS Engine:

```
libdecoder/http/testsuite/http_xss.ey
.....
%event "response_header_host"      "http_response_header_t *"
.....
```

3.3.2 Submitting Event

Every Non-Terminal Symbol generated in bottom-to-top parsing is submitted to the eyoung IPS Engine for further scanning and detection. For this purpose, the eyoung modifies the code-generation template file `yacc.c` of the tool bison. A patch located in `tool/yacc.c.diff` must be done to the source code of bison. Meanwhile, corresponding statements shall be added in grammar files to use the patch:

libdecoder/http/decode/http_server.y file:

```
#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)
```

The macro `YY_REDUCTION_CALLBACK` is an extension for bison and is called when the LALR parser does a reduction. The function `http_element_detect` has been detailed in 2.5.2. After being patched, the patched `yacc.c` in bison is to show how `YY_REDUCTION_CALLBACK` is called. Readers can read the patched file.