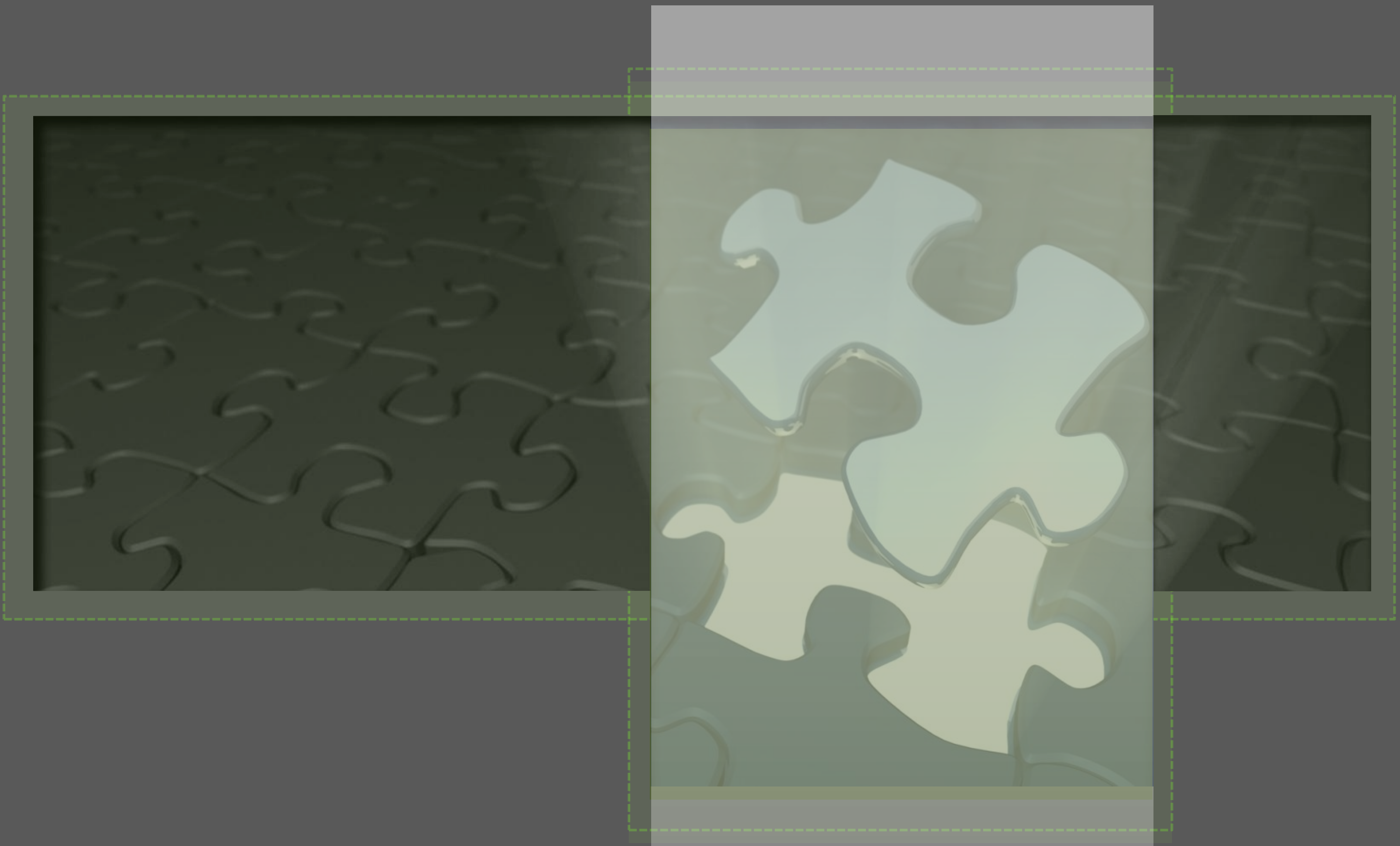


How to implement a plugin in Wirebrus4SPAM

Table of contents

Basic concepts 1

I. Introduction

Wirebrush4SPAM is an extreme efficient open source spam filtering framework and middleware. It design and functionalities are initially inspired in the SpamAssassin framework, but it has been written from scratch in C language including a wide variety of improvements in order to reduce the spam filtering time and to enhance the filtering framework personalization.

Figure 1. Wirebrush4SPAM operation processFigure 1 shows Wirebrush4SPAM main operation process. As described, the process is divided in four stages: (*i*) e-mail parsing, (*ii*) rule execution, (*iii*) learning issues and, finally (*iv*) report generation.

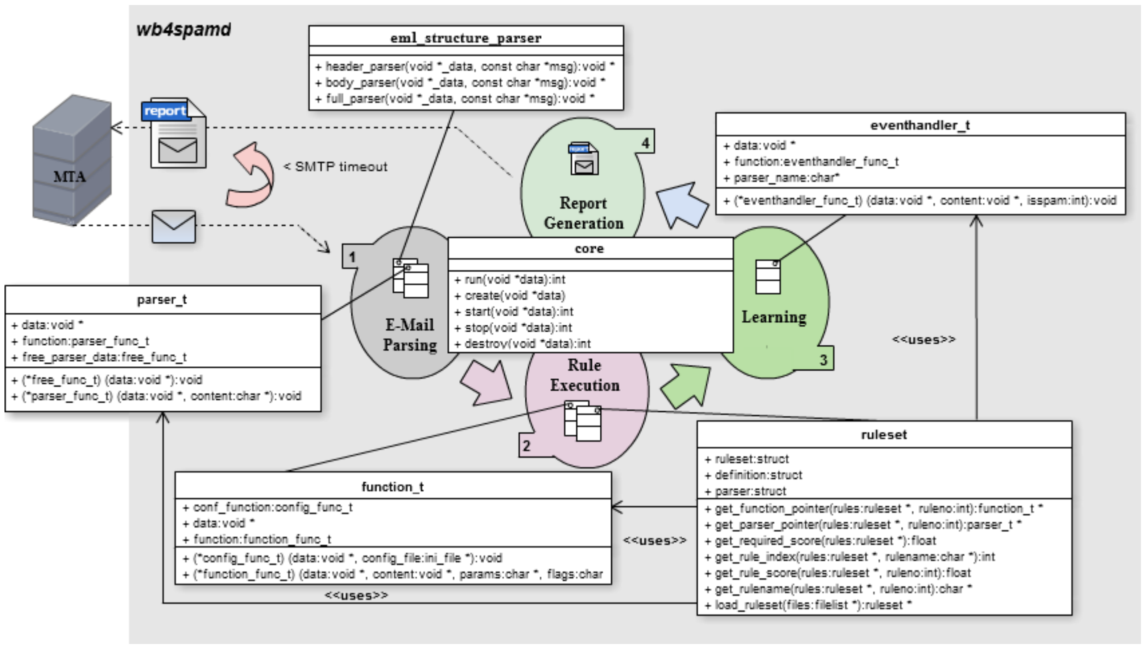

wb4spamd
eml_structure_parser
+ header_parser(void *_data, const char *msg):void *
+ body_parser(void *_data, const char *msg):void *
+ full_parser(void *_data, const char *msg):void *
MTA
report
< SMTP timeout
4
Report Generation
eventhandler_t
+ data:void *
+ function:eventhandler_func_t
+ parser_name:char*
+ (*eventhandler_func_t) (data:void *, content:void *, isspam:int):void
1
E-Mail Parsing
core
+ run(void *data):int
+ create(void *data)
+ start(void *data):int
+ stop(void *data):int
+ destroy(void *data):int
Learning
3
<<uses>>
parser_t
+ data:void *
+ function:parser_func_t
+ free_parser_data:free_func_t
+ (*free_func_t) (data:void *):void
+ (*parser_func_t) (data:void *, content:char *):void
Rule Execution
2
ruleset
+ ruleset:struct
+ definition:struct
+ parser:struct
+ get_function_pointer(rules:ruleset *, ruleno:int):function_t *
+ get_parser_pointer(rules:ruleset *, ruleno:int):parser_t *
+ get_required_score(rules:ruleset *):float
+ get_rule_index(rules:ruleset *, rulename:char *):int
+ get_rule_score(rules:ruleset *, ruleno:int):float
+ get_rulename(rules:ruleset *, ruleno:int):char *
+ load_ruleset(files:filelist *):ruleset *
function_t
+ conf_function:config_func_t
+ data:void *
+ function:function_func_t
+ (*config_func_t) (data:void *, config_file:ini_file *):void
+ (*function_func_t) (data:void *, content:void *, params:char *, flags:char
<<uses>>
<<uses>>

Figure 1. Wirebrush4SPAM operation process

Whenever an email is received, the MTA (*Mail Transfer Agent*) sends it to the Wirebrush4SPAM daemon (called wb4spamd). Below starts the filtering process. In the first stage the middleware executes from all the EML parsers available only those required by the rules. In the next stage, the application runs (concurrently if it is possible) all the filtering rules defined. The three stage is optional and, it will be activated only when it is necessary to store the email. Finally, the last stage adds additional info to the email to indicate the classification info belonging to the email.

II. Wirebrush4SPAM architecture

Wirebrush4SPAM has been written in C programing language. Nevertheless its architecture has been inspired in an object-oriented design where classes have been substituted by modules implementing abstract data types.

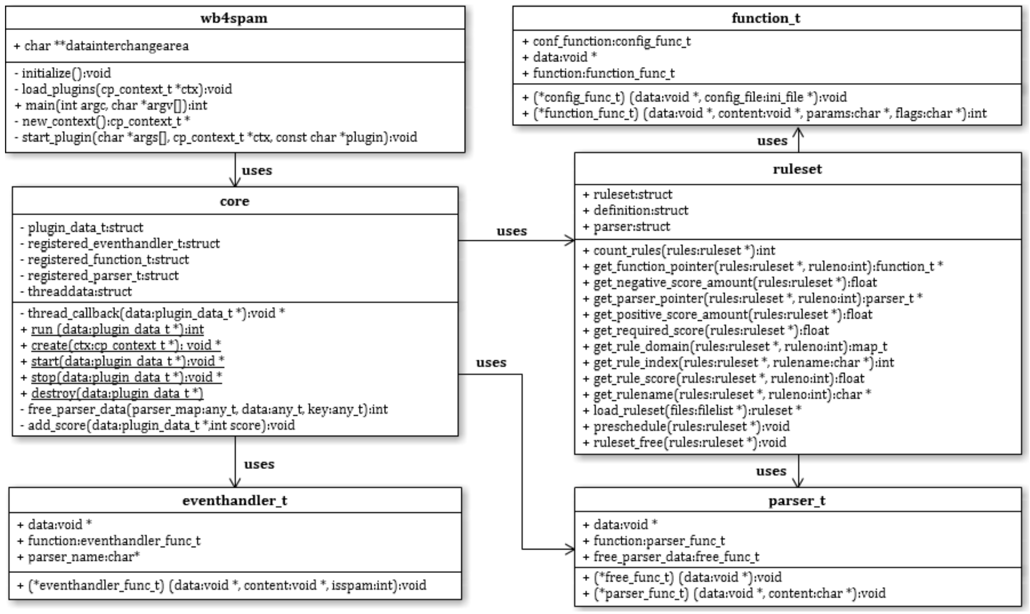

wb4spam
+ char **datainterchangearea
- initialize():void
- load_plugins(cp_context_t *ctx):void
+ main(int argc, char *argv[]):int
- new_context():cp_context_t *
- start_plugin(char *args[], cp_context_t *ctx, const char *plugin):void
uses
core
- plugin_data_t:struct
- registered_eventhandler_t:struct
- registered_function_t:struct
- registered_parser_t:struct
- threaddata:struct
- thread_callback(data:plugin_data_t *):void *
+ run (data:plugin data t *):int
+ create(ctx:cp context t *): void *
+ start(data:plugin data t *):void *
+ stop(data:plugin data t *):void *
+ destroy(data:plugin data t *)
- free_parser_data(parser_map:any_t, data:any_t, key:any_t):int
- add_score(data:plugin_data_t *,int score):void
uses
eventhandler_t
+ data:void *
+ function:eventhandler_func_t
+ parser_name:char*
+ (*eventhandler_func_t) (data:void *, content:void *, isspam:int):void
function_t
+ conf_function:config_func_t
+ data:void *
+ function:function_func_t
+ (*config_func_t) (data:void *, config_file:ini_file *):void
+ (*function_func_t) (data:void *, content:void *, params:char *, flags:char *):int
uses
uses
ruleset
+ ruleset:struct
+ definition:struct
+ parser:struct
+ count_rules(rules:ruleset *):int
+ get_function_pointer(rules:ruleset *, ruleno:int):function_t *
+ get_negative_score_amount(rules:ruleset *):float
+ get_parser_pointer(rules:ruleset *, ruleno:int):parser_t *
+ get_positive_score_amount(rules:ruleset *):float
+ get_required_score(rules:ruleset *):float
+ get_rule_domain(rules:ruleset *, ruleno:int):map_t
+ get_rule_index(rules:ruleset *, rulename:char *):int
+ get_rule_score(rules:ruleset *, ruleno:int):float
+ get_rulename(rules:ruleset *, ruleno:int):char *
+ load_ruleset(files:filelist *):ruleset *
+ preschedule(rules:ruleset *):void
+ ruleset_free(rules:ruleset *):void
uses
uses
parser_t
+ data:void *
+ function:parser_func_t
+ free_parser_data:free_func_t
+ (*free_func_t) (data:void *):void
+ (*parser_func_t) (data:void *, content:char *):void

Figure 2. Wirebrush4SPAM architecture

As we can see from Figure 2, key characteristics from the object-oriented paradigm (including encapsulation and information hiding) have been ported to the Wirebrush4SPAM architecture. As we can see *wb4spam* is the main program being able to load the C-Pluff plugin architecture, initialize the core plugin and subsequently forward messages to it. The core plugin is able to classify e-mails by evaluating rules included in a ruleset. For the execution of each rule, the core plugin first obtains the message content by using a *parser_t* data type and then execute a *function_t* to check the matching with the target e-mail. In order to increase filtering speed, each *parser_t* is launched only when the filter contains a rule that requires its usage, being executed only one time per message by using a parsed contents caching scheme. Once Wirebrush4SPAM has classified the new incoming message, the core plugin calls all registered *eventhandler_t* to notify the final decision about the e-mail. In the proposed scheme, *event handlers* are the Wirebrush4SPAM mechanism to support automatic learning processes.

III. Wirebrush4SPAM main concepts

Wirebrush4SPAM contains three main concepts: *parsers, filtering functions and event listeners.* They are modelled as extensions that can be connected with the core plugin through the corresponding extension points. Therefore, a Wirebrush4SPAM plugin (except the core) is composed by a set of *parsers*, *filtering functions* and *event listeners* sharing some semantic or functional relationships.

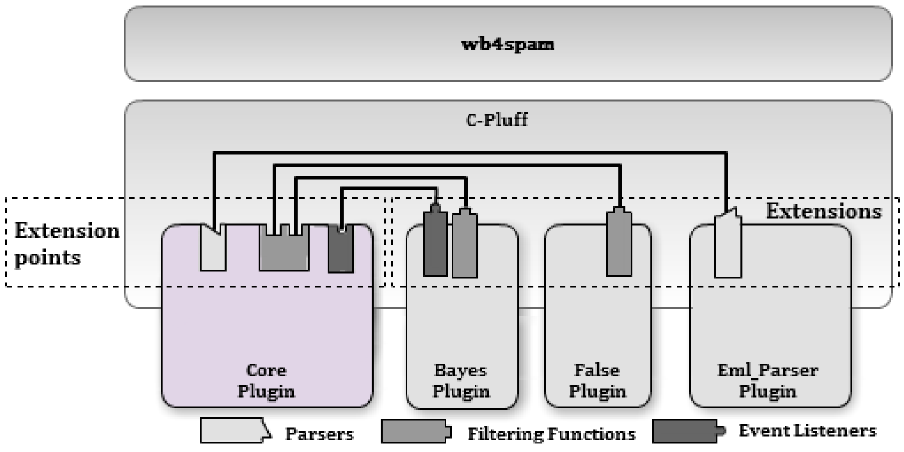


Figure 3. Wirebrush4SPAM plugin architecture.

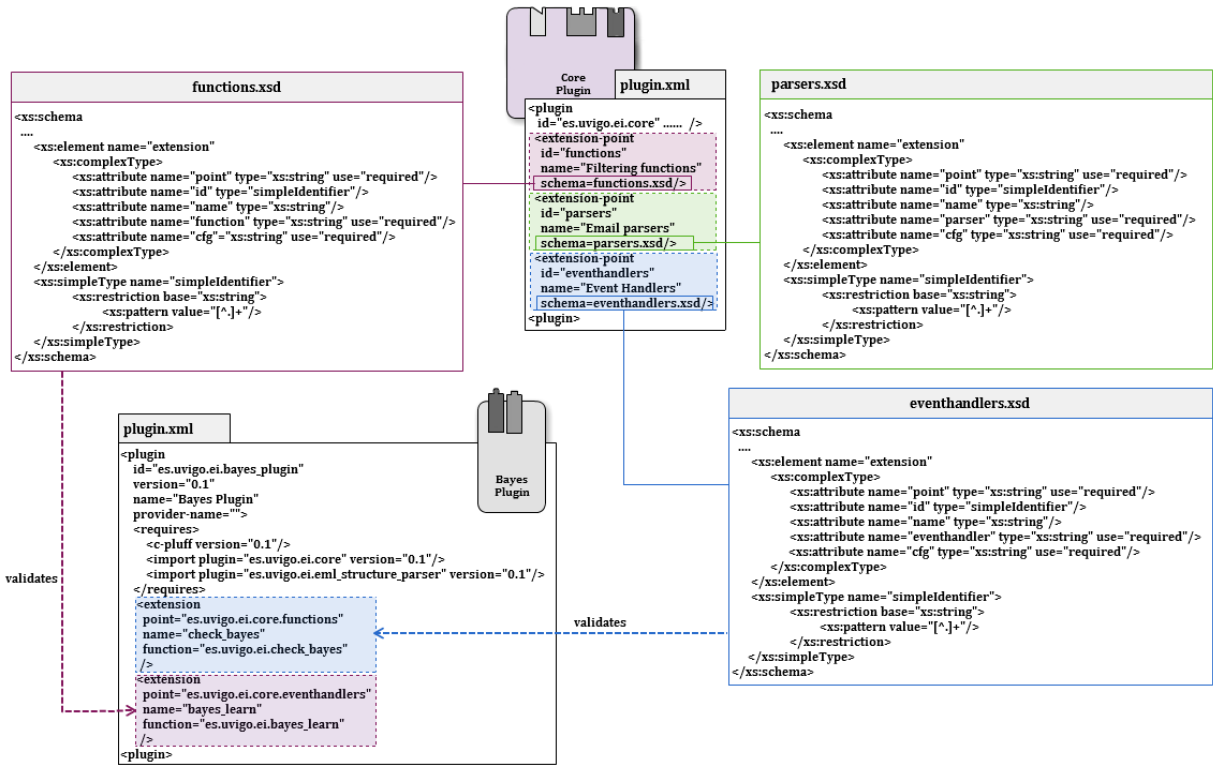
Figure 3 exemplifies the Wirebrush4SPAM plugin architecture by representing some available plugins. In this context, Bayes plugin registers a *filtering function* and an *event listener* used to support the learning requirements. Moreover, Wirebrush4SPAM also includes an EML parser plugin that contains *header*, *full* and *body* parsers used to extract information and tokenize the corresponding parts of any e-mail represented in RFC2822 format.

Plugin implementation 2

I. Plugin description

Each spam detection and filtering technique available for Wirebrush4SPAM must be implemented as a plugin. There are different types of plugins (*i*) static plugins and (*ii*) dynamic plugins.

In this context static plugins are those that not need to manage and manipulate dynamic data structures. In contrast, dynamic plugins are those that handlers all king of dynamic structures.



functions.xsd
```
<xs:schema
 ....
   <xs:element name="extension"
      <xs:complexType>
         <xs:attribute name="point" type="xs:string" use="required"/>
         <xs:attribute name="id" type="simpleIdentifier"/>
         <xs:attribute name="name" type="xs:string"/>
         <xs:attribute name="function" type="xs:string" use="required"/>
         <xs:attribute name="cfg"="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:simpleType name="simpleIdentifier">
         <xs:restriction base="xs:string">
               <xs:pattern value="[^.]+"/>
         </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

plugin.xml
```
<plugin
  id="es.uvigo.ei.core" ...... />
 <extension-point
  id="functions"
  name="Filtering functions"
  schema=functions.xsd/>
 <extension-point
  id="parsers"
  name="Email parsers"
  schema=parsers.xsd/>
 <extension-point
  id="eventhandlers"
  name="Event Handlers"
  schema=eventhandlers.xsd/>
<plugin>
```

parsers.xsd
```
<xs:schema
 ....
   <xs:element name="extension"
      <xs:complexType>
         <xs:attribute name="point" type="xs:string" use="required"/>
         <xs:attribute name="id" type="simpleIdentifier"/>
         <xs:attribute name="name" type="xs:string"/>
         <xs:attribute name="parser" type="xs:string" use="required"/>
         <xs:attribute name="cfg" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:simpleType name="simpleIdentifier">
         <xs:restriction base="xs:string">
               <xs:pattern value="[^.]+"/>
         </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

plugin.xml
```
<plugin
  id="es.uvigo.ei.bayes_plugin"
  version="0.1"
  name="Bayes Plugin"
  provider-name="">
  <requires>
    <c-pluff version="0.1"/>
    <import plugin="es.uvigo.ei.core" version="0.1"/>
    <import plugin="es.uvigo.ei.eml_structure_parser" version="0.1"/>
  </requires>
 <extension
  point="es.uvigo.ei.core.functions"
  name="check_bayes"
  function="es.uvigo.ei.check_bayes"
 />
 <extension
  point="es.uvigo.ei.core.eventhandlers"
  name="bayes_learn"
  function="es.uvigo.ei.bayes_learn"
 />
<plugin>
```

eventhandlers.xsd
```
<xs:schema
 ....
   <xs:element name="extension"
      <xs:complexType>
         <xs:attribute name="point" type="xs:string" use="required"/>
         <xs:attribute name="id" type="simpleIdentifier"/>
         <xs:attribute name="name" type="xs:string"/>
         <xs:attribute name="eventhandler" type="xs:string" use="required"/>
         <xs:attribute name="cfg" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:simpleType name="simpleIdentifier">
         <xs:restriction base="xs:string">
               <xs:pattern value="[^.]+"/>
         </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Figure 4. Plugin definition scheme.

Figure 4 includes a detailed schema of the Bayes plugin implementation. As we can see from this figure, every plugin contains a descriptor file named plugin.xml that specifies runtime features including (*i*) the plugin id, (*ii*) available extension points, (*iii*) implemented extensions, (*iv*) a dynamic library file, (*v*) existing plugin dependences and (*vi*) different user defined information. These extensions defined by a plugin are connected with their corresponding extension points by some sentences included in the plugin descriptor. As showed in Figure 4, Bayes plugin implements *check_bayes* filtering function and *bayes_learn* event handler by extending the corresponding extension points. The core plugin defines three extension points matching with the main concepts of Wirebrush4SPAM platform: *parsers*, *filtering functions* and *event listeners*. For each extension point, an XML schema file (.xsd) should be created containing the parameters to be included in a plugin descriptor (plugin.xml). Also, Figure 4 indicates Bayes plugin dependences. Those dependences avoid plugin execution when at least one of the dependencies does not exist.

II. Plugin implementation API

In order to facilitate the implementation, developing and deploying of new techniques and plugins, Wirebrush4SPAM provides an API facility.

As shown in Figure 5, the API facility is composed by four basic data structures: (*i*) cache (*ii*) linked list (*iii*) hashmap and finally (*iv*) the stack structure.

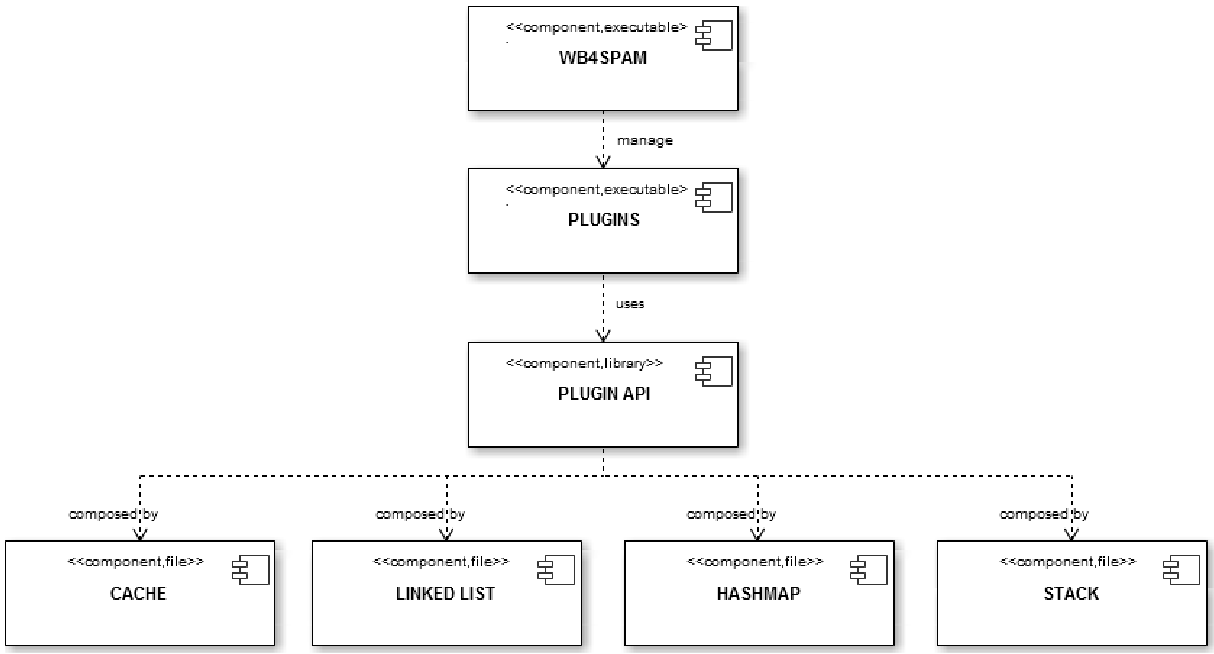


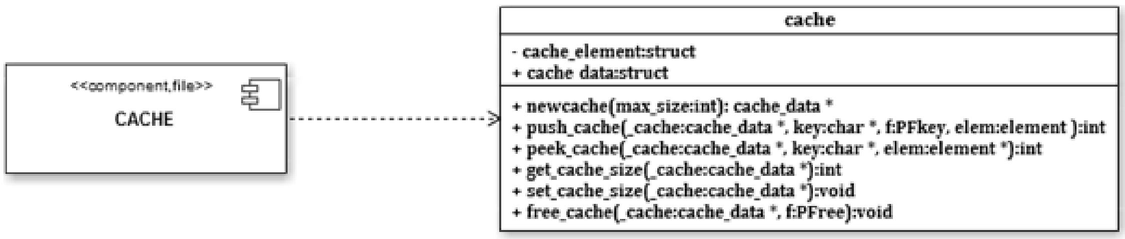
Figure 5. Wirebrush4SPAM API component diagram.

Next subsections describe thoroughly each API component.

II.I. Cache component

Cache is a component that transparently stores data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the greater the number of requests that can be served from the cache, the faster the overall system performance becomes.

To be cost efficient and to enable an efficient use of data, caches are relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications have locality of reference. References exhibit temporal locality if data is requested again that has been recently requested already. References exhibit spatial locality if data is requested that is physically stored close to data that has been requested already.



```
- cache_element:struct
+ cache data:struct
```

```
+ newcache(max_size:int): cache_data *
+ push_cache(_cache:cache_data *, key:char *, f:PFkey, elem:element ):int
+ peek_cache(_cache:cache_data *, key:char *, elem:element *):int
+ get_cache_size(_cache:cache_data *):int
+ set_cache_size(_cache:cache_data *):void
+ free_cache(_cache:cache_data *, f:PFree):void
```

Figure 6. Cache API diagram.

Figure 6 describes the functions and data types provided by the cache structure. The description of each method is described below.

Functions	**Description**
newcache	Initialize the cache data structure.
push_cache	Insert a new element in the cache. If the cache is full, automatically removes the last inserted element.
peek_cache	Obtain the element from the cache. If the element exists returns CACHE_ELEMENT_FOUND, otherwise returns CACHE_ELEMENT_MISSING
get_cache_size	Returns the maximum cache size.
set_cache_size	Modifies the cache size
free_cache	Free al the cache data structures created and all the stored data.

Table 1. Cache methods description

II.II. Linked list component

¡Error! No se encuentra el origen de la referencia. shows the description of *linked list* methods. *Linked list* is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a datum and a reference (in other words, a link) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data structures, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

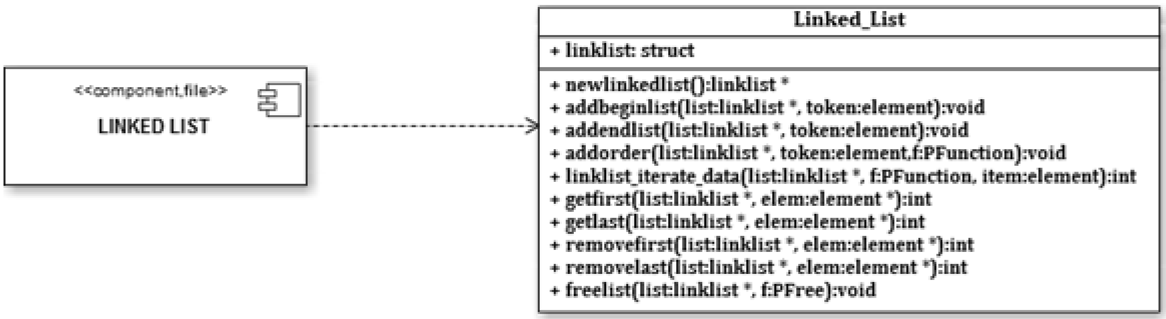

<<component,file>>
LINKED LIST
Linked_List
+ linklist: struct
+ newlinkedlist():linklist *
+ addbeginlist(list:linklist *, token:element):void
+ addendlist(list:linklist *, token:element):void
+ addorder(list:linklist *, token:element,f:PFunction):void
+ linklist_iterate_data(list:linklist *, f:PFunction, item:element):int
+ getfirst(list:linklist *, elem:element *):int
+ getlast(list:linklist *, elem:element *):int
+ removefirst(list:linklist *, elem:element *):int
+ removelast(list:linklist *, elem:element *):int
+ freelist(list:linklist *, f:PFree):void

Figure 7. Linked list API diagram

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require scanning most or all of the list elements.

Below Table 2 shows the description of all methods included in the *linked list* API.

Functions	Description
newlinkedlist	Initialize the *linked list* data structure.
addbeginlist	Insert the element at the beginning of the list.
addendlist	Inserts the elements at the end of the list.
addorder	Insert the element ordered in the list. This method needs a *comparison function* for indicating the correct position of the element in the list.
linklist_iterate_data	Iterator for transverse all the elements stored in the linked list
get_first	Gets the element located at the first position in the linked list.
get_last	Gets the element located at the last position in the linked list.
removefirst	Removes the element located at the first position in the linked list.
removelast	Removes the last element in the linked list.
freelist	Free al the linked list data structure.

Table 2. Linked list API methods description

II.III. Stack component

Another component provided by the Plugin Implementation API is the stack. Stack is a last in, first out (LIFO) abstract data type and linear data structure. A stack can have any abstract data type as an element, but is characterized by only three fundamental operations: push, pop and peek. The *push* operation adds a new item to the top of the stack, or initializes the stack if it is empty. The *pop* operation removes an item from the top of the stack. A *pop* either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it returns a NULL element. The *peek* operation gets the data from the top-most position and returns it to the user without deleting it. The same NULL element return state can also occur in *peek* operation if stack is empty.

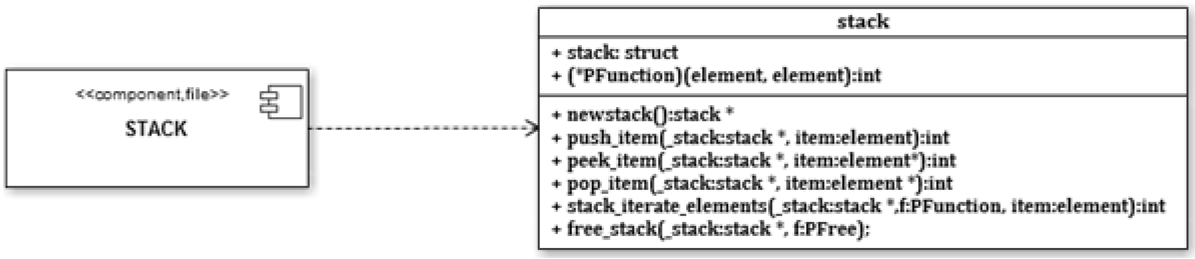

<<component,file>>
STACK
stack
+ stack: struct
+ (*PFunction)(element, element):int
+ newstack():stack *
+ push_item(_stack:stack *, item:element):int
+ peek_item(_stack:stack *, item:element*):int
+ pop_item(_stack:stack *, item:element *):int
+ stack_iterate_elements(_stack:stack *,f:PFunction, item:element):int
+ free_stack(_stack:stack *, f:PFree):

Figure 8. Stack structure diagram

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the *pop* and *push* operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest.

The description of all the functions provided in the stack library are shown in Table 3.

Functions	Description
newstack	Initialize the *stack* data structure.
push_item	Insert the element at the top of the stack.
peek_item	Returns and deletes the element positioned at the top of the stack (returns the last inserted element).
pop_item	Returns (but not deletes) the element at the top of the stack.
stack_iterate_elements	Loops through all the elements stored in the stack structure.
free_stack	Liberates all the elements stored in the stack structure.

Table 3. Stack structure diagram

II.IV. Hashmap component

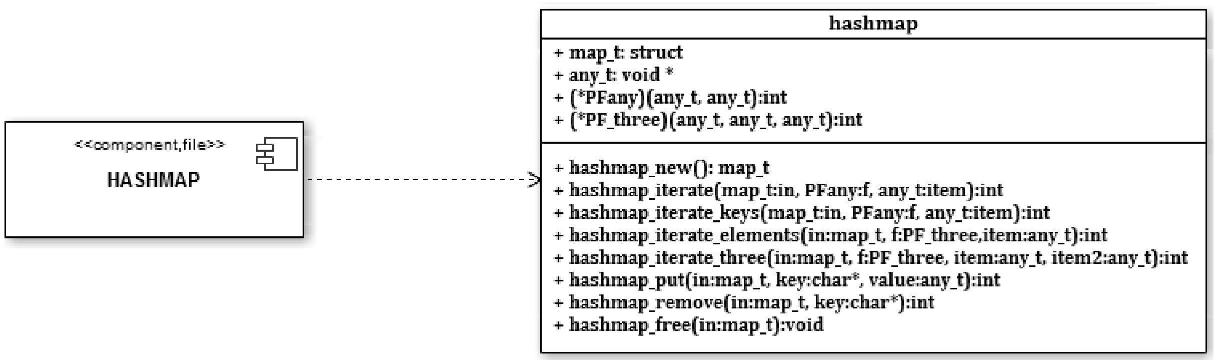
Hashmap is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). Thus, a hash table implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that hash collisions—different keys that map to the same hash value—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

Figure 9 describes thoroughly the methods included in the *hashmap* component.



<<component,file>>
HASHMAP
hashmap
+ map_t: struct
+ any_t: void *
+ (*PFany)(any_t, any_t):int
+ (*PF_three)(any_t, any_t, any_t):int
+ hashmap_new(): map_t
+ hashmap_iterate(map_t:in, PFany:f, any_t:item):int
+ hashmap_iterate_keys(map_t:in, PFany:f, any_t:item):int
+ hashmap_iterate_elements(in:map_t, f:PF_three,item:any_t):int
+ hashmap_iterate_three(in:map_t, f:PF_three, item:any_t, item2:any_t):int
+ hashmap_put(in:map_t, key:char*, value:any_t):int
+ hashmap_remove(in:map_t, key:char*):int
+ hashmap_free(in:map_t):void

Figure 9. Hashmap structure diagram

The description of each implemented method is described as shown in Table 4.

Functions	Description
hashmap_new	Initialize the *hashmap* data structure.
hashmap_iterate	Iterates through all the data items stored in the *hashmap*. The function passed as parameter must be *f(item,data)* .
hashmap_iterate_elements	Iteration function to transverse thoroughly all the elements (both item and key) stored in the *hashmap*. The function received as parameter must be *f(item1, data, key)*
hashmap_iterate_three	Function that allows iterating through all the key elements stored in the *hashmap*. The function received as parameter must be *f(item1,item2,key)*
hashmap_put	Insert the element into the *hashmap*.
hashmap_remove	Deletes the *hashmap* the specified element from
hashmap_free	Deletes the hashmap structure, but does not delete the elements stored in the hashmap.

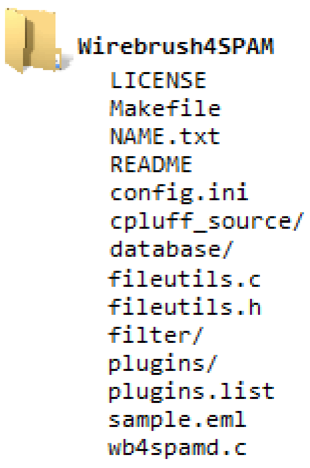
Table 4. Hashmap functions description

III. Plugin implementation example

The sections before, describes theoretically the plugin architecture and the API provided by the plugin platform. In this section we will explain how to implement a new plugin from scratch.

We divided for clarification the implementation of a plugin for Wirebrush4SPAM into 5 stages.

For this example, the reference path will be Wirebrush4SPAM main path. Figure 10 shows the directory tree membership.



```
Wirebrush4SPAM
  LICENSE
  Makefile
  NAME.txt
  README
  config.ini
  cpluff_source/
  database/
  fileutils.c
  fileutils.h
  filter/
  plugins/
  plugins.list
  sample.eml
  wb4spamd.c
```

Figure 10. Wirebrush4SPAM directory tree

III.I. Stage 1: Plugin implementation

First of all, it is necessary to implement the new plugin. As commented above Wirebrush4SPAM is able to manage two types of plugins (*i*) static plugins and (*ii*) dynamic plugins. Therefore, in this tutorial, we will implement both plugin types.

In this example *static_plugin* will be the folder name of *the static plugin,* and *dynamic_plugin* will be the name of the dynamic plugin folder.

Static plugin implementation

The figure below shows the implementation of a plugin that prints "Hello world" in the command prompt.

```
#include <stdio.h>
#include <cpluff.h>
#include "core.h"

/**
 * This plugin always returns 0 and prints hello world in the command prompt.
 */
```

```
08
09 static int static_hello (void *_data, void *msg, char *params, char *flags) {
10       printf ("Hello world.\n");
11       return 0;
12 }
13
14 /* -----------------------------------------------------------------------
15  * Exported classifier information
16  * ---------------------------------------------------------------------*/
17
18 CP_EXPORT function_t es_uvigo_ei_static_hello = { NULL, static_hello, NULL };
19
```

Figure 11. Static plugin implementation example

As shown in line 10 the function must return a value. The possible values are: (*i*) zero if the function does not match or (*ii*) one if the function matches and therefore the score associated to the triggered rule will be added. In this example we consider to return a zero.

Dynamic plugin implementation

This plugin has the same behaviour as the previous plugin with the exception that the printed string must be assigned dynamically.

```
#include <string.h>
#include <stdio.h>
#include <cpluff.h>
#include "core.h"

struct plugin_data{
    function_t *funcs;
    cp_context_t *ctx;
    eventhandler_t *events;
    char *plugin_string;
};

static void *create(cp_context_t *ctx){
    struct plugin_data *retval;
    retval=malloc(sizeof(struct plugin_data));
    retval->ctx=ctx;
    return retval;
}

static int start(void *d) {
    struct plugin_data *data=(struct plugin_data *)d;
    cp_context_t *ctx;
    if (data!=NULL)
        data->plugin_string = malloc(sizeof(char)*12);
25         sprintf(data->plugin_string,"%s","hello_world");
26     else return CP_ERR_RESOURCE;
27
28     data->funcs=(function_t *)malloc(sizeof(function_t));
29     data->funcs->function=&dynamic_hello;
30     data->funcs->conf_function=NULL;
31     data->funcs->data=data;
32
33     data->events=(eventhandler_t *)malloc(sizeof(eventhandler_t));
34     data->events->function=&autolearn_hello_world;
35     data->events->data=data;
36     data->events->parser_name="body";
37
38     if ((cp_define_symbol(ctx, "es_uvigo_ei_dynamic_hello", data->funcs)==CP_OK) &&
39         (cp_define_symbol(ctx, "es_uvigo_ei_autolearn_hello_world", data->events)==CP_OK))
40         return CP_OK;
41     else return CP_ERR_RESOURCE;
42 }
43
44 static void stop(void *d) {
45    struct plugin_data *data=(struct plugin_data *)d;
46    if(data->plugin_string!=NULL) free(data->plugin_string);
47 }
48
49 static void destroy(void *d) {
50    struct plugin_data *data=(struct plugin_data *)d;
51    free(data->funcs);
52    free(data->events);
53    free(data);
54 }
55
56 static int dynamic_hello(void *_data, void *content, char *params, const char *flags){
57     printf("%s\n", data->plugin_string);
58 }
59
60 static void autolearn_spam_hunting(void *_data, void *_content, const int isspam){
61     printf("Executing autolearn hello world...\n");
62 }
63
64 CP_EXPORT cp_plugin_runtime_t dynamic_plugin_runtime_functions={create,start,stop,destroy};
65
```

Figure 12. Dynamic plugin implementation example

Figure 12 shows an example of a dynamic plugin implementation. As we can observe, plugins using internal data structures are more complex to define and require a *cp_plugin_runtime* export sentence to identify the source to execute, create, start, stop and destroy data structures. Moreover, required exports for *function_t*, *parser_t* or *eventhandler_t* variables should be defined inside the start function as showed in lines 28 to 36.

Green areas means the code is common (an also required) in all plugins. Moreover blue areas means optional code and can only be implemented depending on plugin needs.

Also, Wirebrush4SPAM platform allows defining one function per plugin that should be executed at the end of each email classification. These functions called eventhandlers and usually are used to implement some learning techniques.

The main difference between static plugin is the need of defining four new functions (*i*) create, (*ii*) start (*iii*) stop and (*iv*) destroy. The following explains the operation of each one.

i. Create function: the initialization function called to create a new plug-in runtime instance. The initialization function initializes and returns an opaque plug-in instance data pointer which is then passed on to other control functions. This data pointer should be used to access plug-in instance specific data. For example, the context reference must be stored as part of plug-in instance if the plug-in runtime needs it. On failure, the function must return NULL.

ii. Start function: these start function called to start a plug-in instance. The start function must return zero (CP_OK) on success and non-zero on failure. If the start fails then the stop function (if any) is called to clean up plug-in state.

iii. Stop function: the stop function is called to stop a plugin instance. This function must cease all plug-in runtime activities. The stop function should release any external resources hold by the plug-in. Dynamically resolved symbols are automatically released and dynamically defined symbols and registered run functions are automatically unregistered after the call to stop function.

iv. Destroy function: these destroy function is called to destroy a plug-in instance. This function should release any plug-in instance data. The plug-in is stopped before this function is called.

III.II. Stage 2: Plugin descriptor

A plug-in descriptor is an XML document describing a plug-in. It includes information about the contents of the plug-in, the features provided by the plug-in, plug-in version information and static dependencies of the plug-in. Most of the elements are optional. Most of the descriptor information described here is available to software via *cp_plugin_info_t* structure. The plugin descriptor must be located in the plugin directory as plugin.xml.

Static plugin descriptor

Figure 13 contains the source code belonging to the descriptor of *static_plugin*. As previously mentioned, the descriptor file is required to link the exported extensions to the *core* extension points.

```
<plugin
  id="es.uvigo.ei.static_plugin"
  version="0.1"
  name="Static Plugin"
  provider-name="HOW-TO FOR STATIC PLUGIN">
  <requires>
    <c-pluff version="0.1"/>
    <import plugin="es.uvigo.ei.core" version="0.1"/>
  </requires>
  <runtime library="libstatic_plugin"/>
  <extension
     point="es.uvigo.ei.core.functions"
     name="static_hello"
     cfg=""
     function="es_uvigo_ei_static_hello"
  />
</plugin>
```

Figure 13. Plugin descriptor for static_plugin

It is important to use this descriptor as a template for all the static plugins implemented. As commented in previous sections, all descriptors are validated by a plugin scheme (xsd).

Dynamic plugin descriptor

Figure 14 contains the source code belonging to the descriptor of *dynamic_plugin*. As we can see, plugin descriptors for dynamic plugins are slightly more complicated than plugin descriptor for static plugins.

```
<plugin
  id="es.uvigo.ei.dynamic_plugin"
  version="0.1"
  name="Dynamic Plugin"
  provider-name="HOW-TO DYNAMIC PLUGIN">
  <requires>
    <c-pluff version="0.1"/>
    <import plugin="es.uvigo.ei.core" version="0.1"/>
  </requires>
  <runtime library="libdynamic_plugin"
funcs="dynamic_plugin_runtime_functions"/>
  <extension
     point="es.uvigo.ei.core.functions"
     name="dynamic_hello"
     cfg=""
     function="es_uvigo_ei_dynamic_hello"
   />
   <extension
     point="es.uvigo.ei.core.eventhandlers"
     name="autolearn_hello_world"
     cfg=""
     eventhandler="es_uvigo_ei_autolearn_hello_world"
   />
</plugin>
```

Figure 14. Plugin descriptor for dynamic plugin

III.III. Stage 3: Plugin compiling issues

In this subsection we present the Makefile for both plugins. The Makefile should define a dynamic library. This library it is necessary by the plugin framework for linking.

This Makefiles are only an indicative and cannot be used as a template for other plugins. Makefile are a complex rule-compiling file and is only valid for the application witch was build.

Figure 15 shows the Makefile structure for the static plugin implemented. As we can observe, lines 7 to 10 defines the dynamic library needed for the plugin platform.

```
01 CC=cc
02 OPTS=-Wall -O2 -g -fPIC -I../../cpluff/include -I/usr/include -I../core
03 OPTSLIB=-shared -W1,-soname,libfalse_plugin.so.0 -I../../cpluff/include -
04 I/usr/include -I../core -L../../cpluff/lib -L/usr/lib -L/lib -L../core
05 LIBS=-lcpluff -lexpat -lpthread -ldl -lc
06
07 libstatic_plugin.so : static_plugin.o
08         gcc $(OPTSLIB) -o libstatic_plugin.so.0 static_plugin.o $(LIBS)
09         ln -sf libstatic_plugin.so.0 libstatic_plugin.so
10         ln -sf libstatic_plugin.so.0 libstatic_plugin.so.1
11
12 static_plugin.o :
13         $(CC) $(OPTS) $(LIBS) -c static_plugin.c
14
15 clean : rm *.o *.so *.so.?
16
```

Figure 15. Makefile for static_plugin

Figure 16, describes the structure of the Makefile used to compile the dynamic_plugin implantation. As we can see the structure is similar to the static one.

```
01 all: dynamic_plugin
02
03 dynamic_plugin:
04         $(CC) $(CFLAGS) -shared -W1,-soname,libdynamic_plugin.so.0 dyna-
05 mic_plugin.o -o libdynamic_plugin.so.0 $(OPTSLIB) $(LIBS)
06         ln -sf libdynamic_plugin.so.0 libdynamic_plugin.so
07         ln -sf libdynamic_plugin.so.0 libdynamic_plugin.so.1
08
09 dynamic_plugin.o:
10         $(CC) $(CFLAGS) -c dynamic_plugin.c $(OPTSLIB) $(LIBS) -o dyna-
11 mic_plugin.o
12
13 clean : rm *.o *.so *.so.?
14
```

Figure 16. Makefile for dynamic_plugin

III.IV. Stage 4: Registering plugin path

Finally, it is necessary to instruct Wirebrush4SPAM plugin platform the plugin path. To accomplish this task, simply modify the *plugin.list* file adding at the end, the path for the new plugins.

```
plugins/core
plugins/axl_plugin
plugins/regex_plugin
plugins/spf_plugin
plugins/pcre_regex_plugin
plugins/bayes_plugin
plugins/false_plugin
plugins/eml_structure_parser
plugins/rxl_plugin
plugins/static_plugin
plugins/dynamic_plugin
```

Figure 17. Default *plugin.list* content

Filter manipulation 3

I. Filter description

As we comment in previous sections, Wirebrush4SPAM is not a filter, is an extreme efficient middleware and framework for execution and developing of spam filters.

Any Wirebrush4SPAM filter is defined by set of scored rules and a global threshold called ***required_score***. Each rule is composed by a boolean expression (used as trigger) and its associated individual score. Following this simple structure, an e-mail is classified as spam when the sum of individual scores from triggered rules is greater or equal than the value of ***required_score***.

As shown in next figure, Wirebrush4SPAM rule definition format is closer than SpamAssassin one. Under this scenario, it is intended that the migration of Wirebrush4SPAM to SpamAssassin is as simple as possible.

```
<parser_type> <rulename> <ruledefinition>
score <rulename> <rulescore>
[describe <rulename> <ruledescription>]
```

Wirebrush4SPAM filters are defined in *.cf files located in the filter directory. In order to build a filter, these files should contain all rules and the ***required_score*** threshold. Figure 18 shows an example of a Wirebrush4SPAM filter.

```
body BAYES_00 check_bayes(0.00, 0.01)
describe BAYES_00 Bayes between 0 and 0.01
score BAYES_00 -2

body BAYES_05 check_bayes(0.01, 0.05)
describe BAYES_05 Bayes between 0.01 and 0.05
score BAYES_05 -1

body BAYES_20 check_bayes(0.05, 0.20)
describe BAYES_20 Bayes between 0.05 and 0.20
score BAYES_20 -0.5

body BAYES_40 check_bayes(0.20, 0.40)
```

```
14 describe BAYES_40 Bayes between 0.20 and 0.40
15 score BAYES_40 -0.25
16
17 body BAYES_50 check_bayes(0.40, 0.60)
18 describe BAYES_50 Bayes between 0.40 and 0.60
19 score BAYES_50 0
20
21 body BAYES_60 check_bayes(0.60, 0.80)
22 describe BAYES_60 Bayes between 0.60 and 0.80
23 score BAYES_60 0.25
24
25 body BAYES_80 check_bayes(0.80, 0.95)
26 describe BAYES_80 Bayes between 0.80 and 0.95
27 score BAYES_80 1
28
29 body BAYES_95 check_bayes(0.95, 0.99)
30 describe BAYES_95 Bayes between 0.95 and 0.99
31 score BAYES_95 2
32
33 body BAYES_99 check_bayes(0.99, 1.00)
34 describe BAYES_99 Bayes between 0.99 and 1.00
35 score BAYES_99 3
36
37 header HAS_VIAGRA_ON_BODY eval("[vV][iI?1!][aA][gG][rR][aA]")
38 describe HAS_VIAGRA_ON_BODY Contains references to viagra in content
39 score HAS_VIAGRA_ON_BODY 1
40
41 body Levitra_ON_SUBJECT_PCRE pcre_eval_header("Subject","(?i:levitra)")
42 describe Levitra_ON_SUBJECT_PCRE Contains references to levitra in Subject
43 score HAS_LEVITRA_ON_SUBJECT 1
44
45 body SPF_PASS_3 spf_pass(3)
46 describe SP_PASS_3 If third header of e-mail pass the SPF
47 score SPF_PASS_3 -4
48
49 body RWL_DNSWL rxl_check("list.dnswl.org")
50 describe RWL_DNSWL If the third header pass the RWL
51 domain RWL_DNSWL @udc.es @uvigo.es @usc.es
52 score RWL_DNSWL -2
53
54 body RWL_DNSWL_OCTECT rxl_check("list.dnswl.org",3,10)
55 describe RWL_DNSWL_OCTECT If third octet of first header has value 10.
56 score RWL_DNSWL_OCTECT -3
57
58 #Required score to classify a message as spam
59 required_score 3
60
61 #Activate SFE
62 lazy_evaluation -1;
```

Figure 18. Example of Wirebrush4SPAM bayes filter

As showed in Figure 18, the filter involves the execution of a bayes scheme to compute the probability of a message being spam. The proposed filter uses some intervals for the bayes probability and assigns a score for each interval (lines 02, 06, 10, …, to 34). Moreover, it also adds some scores to the target message when it contains the word viagra with different varia-tions (line 38) or when the subject of the e-mail contains the word

Levitra (line 42). These two rules use a different regular expression API to test the specified conditions. The example filter also checks SPF records and a RBL/RWL searching scheme.

II. Filter example

Finally, it is necessary to add the filter those rules that activate the implemented plugin.

```
01 body SIMPLE_PLUGIN_EXAMPLE static_hello()
02 describe SIMPLE_PLUGIN_EXAMPLE Executes static plugin implementation example
03 score SIMPLE_PLUGIN_EXAMPLE 5
04
05 body DYNAMIC_PLUGIN_EXAMPLE dynamic_hello()
06 describe DYNAMIC_PLUGIN_EXAMPLE Executes dynamic plugin implementation
07 score DYNAMIC_PLUGIN_EXAMPLE 3
```

As we can see from the figure above in the lines 02 and 05, the function name must be the same as the defined in the plugin.xml definition scheme.