

LFI WITH SEGMENTATION FAULT

THE PROBLEM OF STRING.STRIP_TAGS FILTER IN PHP

INTRODUCTION

One day when I was trying following code in PHP, however, it came out with a segmentation fault:

```
1. <?php
2. file("php://filter/string.strip_tags/resource=/etc/passwd");
```

A segmentation fault would cause to a serious problem here. I assumed that a hacker includes his webshell in the POST request. Undoubtedly, PHP would store it under the /tmp directory. Originally, it was difficult for the hacker to include his webshell because the temporary file would be deleted soon. But the segmentation fault happening above could change the whole story.

```
1. ?include=php://filter/string.strip_tags/resource=shell.php
```

The temporary file, which was shell.php, would not be deleted because of the segmentation fault. In this case, the hacker can use brute force to start local file inclusion attack with this temporary file.

There were almost no any explanation or analysis for the reason why the segmentation fault would happen in this code. Therefore, I will give my detail of working on analysis in the following sections. In the second section, I will use gdb and static code analysis to expose the secret of the bug. In the third part, I will give one example of exploit. In the fourth part, I will take a conclusion of such vulnerability. In the last part, there will be some reflection on myself about the process of analysis.

ANALYSIS AND DETAILS

The first version of PHP which I ran into the segmentation fault was PHP-7.1.13.

FIRST, COMPILE THE SOURCE OF 7.1.13

In order to find out the cause of the problem accurately, I downloaded the source from the github/php-src. After checking out to the branch of PHP-7.1.13, I used following commands to help me compile the source.

```
1. ./configure --prefix=/install_dir
2. make
3. sudo make install
```

The `/install_dir` in the first command would define the path of your compiled executables. Then, the second command and the third command are responsible for build and compilation. The whole process of compilation would take some time, and I also need to be care about the possible conflict with default package in system. After the installation is completed, there would be an executable php in `/install_dir` or `php-src/sapi/cli/`.

```
shimao@ubuntu:~/php-7.1.13/sapi/cli$ ./php -v
PHP 7.1.13 (cli) (built: Dec 10 2018 01:02:25) ( NTS )
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.1.0, Copyright (c) 1998-2017 Zend Technologies

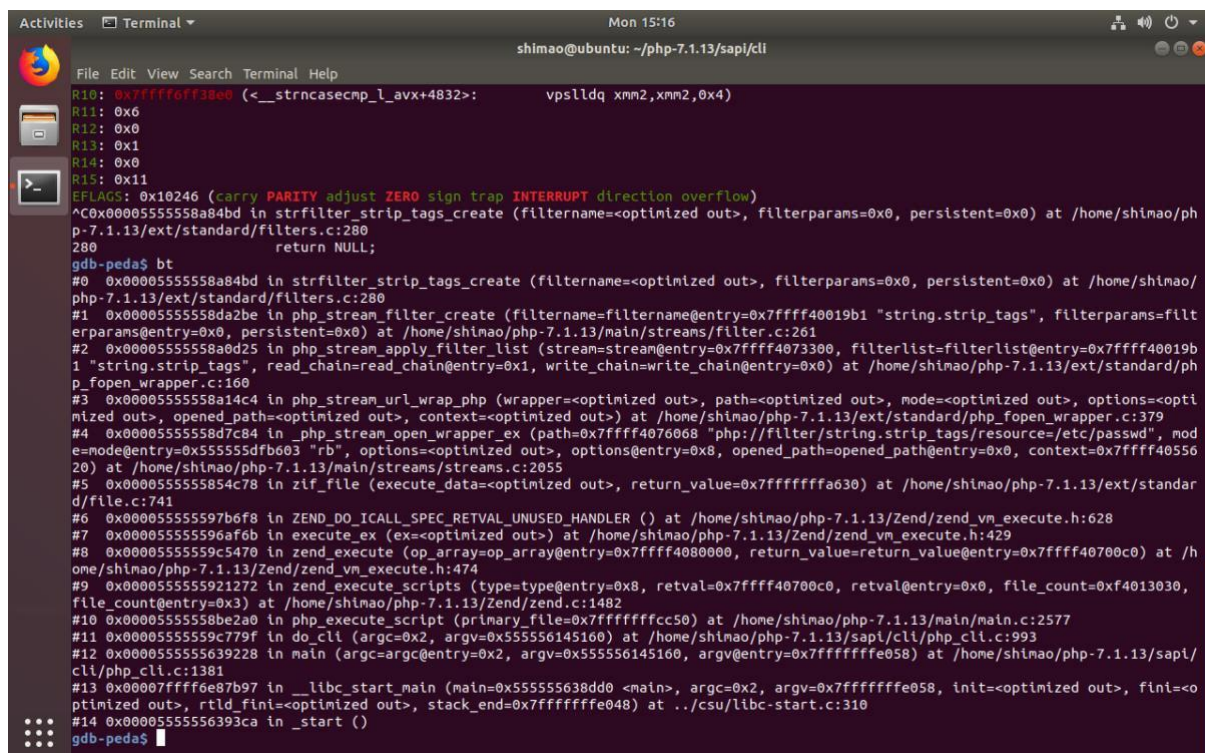
shimao@ubuntu:~/php-7.1.13/sapi/cli$ ./php ./test.php
Segmentation fault

shimao@ubuntu:~/php-7.1.13/sapi/cli$ cat test.php
<?php
file("php://filter/string.strip_tags/resource=/etc/passwd");
shimao@ubuntu:~/php-7.1.13/sapi/cli$
```

SECOND, BACKTRACE WITH GDB

According to bugs.php.net, generating a gdb backtrace is the best way to help PHP developer fix the bug, and it is also a helpful for us to figure out the problem.

1. `gdb ~/php-7.1.13/sapi/cli/php` // then run `./test.php`



```
Mon 15:16
shimao@ubuntu: ~/php-7.1.13/sapi/cli

R10: 0x7ffff6ff38e0 (<__strncasecmp_l_avx+4832>: vpslldq xmm2,xmm2,0x4)
R11: 0x6
R12: 0x0
R13: 0x1
R14: 0x0
R15: 0x11
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
^C0x0000555558a84bd in strfilter_strip_tags_create (filtername=<optimized out>, filterparams=0x0, persistent=0x0) at /home/shimao/php-7.1.13/ext/standard/filters.c:280
280      return NULL;
gdb-peda$ bt
#0 0x0000555558a84bd in strfilter_strip_tags_create (filtername=<optimized out>, filterparams=0x0, persistent=0x0) at /home/shimao/php-7.1.13/ext/standard/filters.c:280
#1 0x0000555558da2be in php_stream_filter_create (filtername=filtername@entry=0x7ffff40019b1 "string.strip_tags", filterparams=filterparams@entry=0x0, persistent=0x0) at /home/shimao/php-7.1.13/main/streams/filter.c:261
#2 0x0000555558a0d25 in php_stream_apply_filter_list (stream=stream@entry=0x7ffff4073300, filterlist=filterlist@entry=0x7ffff40019b1 "string.strip_tags", read_chain=read_chain@entry=0x1, write_chain=write_chain@entry=0x0) at /home/shimao/php-7.1.13/ext/standard/php_fopen_wrapper.c:160
#3 0x0000555558a14c4 in php_stream_url_wrap_php (wrapper=<optimized out>, path=<optimized out>, mode=<optimized out>, options=<optimized out>, opened_path=<optimized out>, context=<optimized out>) at /home/shimao/php-7.1.13/ext/standard/php_fopen_wrapper.c:379
#4 0x0000555558d7c84 in php_stream_open_wrapper_ex (path=0x7ffff4076068 "php://filter/string.strip_tags/resource=/etc/passwd", mode=mode@entry=0x55555dfb603 "rb", options=<optimized out>, options@entry=0x8, opened_path=opened_path@entry=0x0, context=0x7ffff4055620) at /home/shimao/php-7.1.13/main/streams/streams.c:2055
#5 0x000055555854c78 in zif_file (execute_data=<optimized out>, return_value=0x7fffffa630) at /home/shimao/php-7.1.13/ext/standard/file.c:741
#6 0x00005555597b6f8 in ZEND_DO_ICALL_SPEC_RETVAL_UNUSED_HANDLER () at /home/shimao/php-7.1.13/Zend/zend_vm_execute.h:628
#7 0x00005555596af6b in execute_ex (ex=<optimized out>) at /home/shimao/php-7.1.13/Zend/zend_vm_execute.h:429
#8 0x0000555559c5470 in zend_execute (op_array=op_array@entry=0x7ffff4080000, return_value=return_value@entry=0x7ffff40700c0) at /home/shimao/php-7.1.13/Zend/zend_vm_execute.h:474
#9 0x000055555921272 in zend_execute_scripts (type=type@entry=0x8, retval=0x7ffff40700c0, retval@entry=0x0, file_count=0xf4013030, file_count@entry=0x3) at /home/shimao/php-7.1.13/Zend/zend.c:1482
#10 0x0000555558be2a0 in php_execute_script (primary_file=0x7fffffccc50) at /home/shimao/php-7.1.13/main/main.c:2577
#11 0x0000555559c779f in do_cli (argc=0x2, argv=0x555556145160) at /home/shimao/php-7.1.13/sapi/cli/php_cli.c:993
#12 0x0000555555639228 in main (argc=argc@entry=0x2, argv=0x555556145160, argv@entry=0x7fffffefe058) at /home/shimao/php-7.1.13/sapi/cli/php_cli.c:1381
#13 0x00007ffff6e87b97 in __libc_start_main (main=0x55555638dd0 <main>, argc=0x2, argv=0x7fffffefe058, init=<optimized out>, fini=<optimized out>, rtd_fini=<optimized out>, stack_end=0x7fffffefe048) at ../csu/libc-start.c:310
#14 0x0000555556393ca in _start ()
gdb-peda$
```

Therefore, with the information gotten from backtrace, I thought I needed to focus on the part of following functions step by step:

1. `_php_stream_open_wrapper_ex()`
2. `php_stream_url_wrap_php()`
3. `php_stream_apply_filter_list()`
4. `php_stream_filter_create()`
5. `strfilter_strip_tags_create()`

And the segmentation fault happened in the last one (strfilter_strip_tags_create).

[Reference to Generating a gdb backtrace](#)

THIRD, STATIC CODE ANALYSIS

```
1. PHPAPI php_stream *_php_stream_open_wrapper_ex(const char *path, const char *mode,
2.         int options,
3.         zend_string **opened_path, php_stream_context *context STREAMS_DC)
```

The above was the first function call I focused on. It was used to call filter wrapper to open the source of the path argument, which was

“php://filter/string.strip_tags/resource=/etc/passwd” and the mode was “rb”. With the information given by backtrace, the next function call was from line 2055:

```
1. stream = wrapper->wops->stream_opener(wrapper,
2.         path_to_open, mode, options ^ REPORT_ERRORS,
3.         opened_path, context STREAMS_REL_CC);
```

If the wrapper supports the stream to open, the function in another file /ext/standard/php-fopen-wrapper.c would be called:

```
1. php_stream * php_stream_url_wrap_php/php_stream_wrapper *wrapper, const char *path,
2.         const char *mode, int options, zend_string **opened_path, php_stream_context *cont
3.         ext STREAMS_DC) /* {{{ */
```

The function would check the path value which several times of strncasecmp().

```
1. if (!strncasecmp(path, "php://", 6)) {
2.     path += 6;
3. }
4. ....
5. else if (!strncasecmp(path, "filter/", 7)) {
6.     if (strchr(mode, 'r') || strchr(mode, '+')) {
7.         mode_rw |= PHP_STREAM_FILTER_READ;
8.     }
9.     if (strchr(mode, 'w') || strchr(mode, '+' ) || strchr(mode, 'a')) {
10.        mode_rw |= PHP_STREAM_FILTER_WRITE;
11.    }
```

The path which was argument of the function php_Stream_url_wrap_php was the pointer to the address which stored the value of path. In the first if-condition I showed above, the condition was met because the first six characters of path were “php://”, then the address pointer was added by six, which meant next time the path would start from “filter/string...”. Therefore, the if-condition in line:349 of /etc/standard/php_fopen_wrapper.c was met again. In the code between line 6 and line 11 above, mode_rw was set to PHP_STREAM_FILTER_READ because the mode was already set to “rb” previously.

```
1. while (p) {
2.     if (!strncasecmp(p, "read=", 5)) {
3.         php_stream_apply_filter_list(stream, p + 5, 1, 0);
4.     } else if (!strncasecmp(p, "write=", 6)) {
5.         php_stream_apply_filter_list(stream, p + 6, 0, 1);
6.     } else {
7.         php_stream_apply_filter_list(stream, p, mode_rw & PHP_STREAM_FILTER
8.         _READ, mode_rw & PHP_STREAM_FILTER_WRITE);
9.     }
10.    p = php_strtok_r(NULL, "/", &token);
```

```
10.     }
```

Now came to the last part of backtrace showed in this file. Due to the fact that our p was “string.strip_tags/” currently, the else condition was met and went to the line 7 in above. For the function php_stream_apply_filter_list(), I could find the definition in the same file:

```
1. static void php_stream_apply_filter_list(php_stream *stream, char *filterlist, int
  read_chain, int write_chain) /* {{{ */
2. {
3.     char *p, *token = NULL;
4.     php_stream_filter *temp_filter;
5.
6.     p = php_strtok_r(filterlist, "|", &token);
7.     while (p) {
8.         php_url_decode(p, strlen(p));
9.         if (read_chain) {
10.            if ((temp_filter = php_stream_filter_create(p, NULL, php_stream_is_pers
  istent(stream)))) {
11.                php_stream_filter_append(&stream->readfilters, temp_filter);
12.            } else {
13.                php_error_docref(NULL, E_WARNING, "Unable to create filter (%s)", p
  );
14.            }
15.        }
16.        if (write_chain) {
17.            if ((temp_filter = php_stream_filter_create(p, NULL, php_stream_is_pers
  istent(stream)))) {
18.                php_stream_filter_append(&stream->writefilters, temp_filter);
19.            } else {
20.                php_error_docref(NULL, E_WARNING, "Unable to create filter (%s)", p
  );
21.            }
22.        }
23.        p = php_strtok_r(NULL, "|", &token);
24.    }
25. }
```

The first two arguments passed into the function were the entry address of file stream and address pointer to the “string.strip_tags”. The last two argument, which were read_chain and write_chain, were set to 1 and 0 because of mode_rw previously. In the line 9 above, the condition was met and the function php_stream_filter_create() was called.

```
1. PHPAPI php_stream_filter *php_stream_filter_create(const char *filtername, zval *fi
  lterparams, int persistent){
2. ...
3.     if (NULL != (factory = zend_hash_str_find_ptr(filter_hash, filtername, n))
  ) {
4.         filter = factory->
  >create_filter(filtername, filterparams, persistent);
```

I found the definition of this function in the file of /main/streams/filter.c. The condition of finding pointer of filtername in zend_hash table was met, filter was created successfully then the function in another file was called.

```
1. static php_stream_filter *strfilter_strip_tags_create(const char *filtername, zval
  *filterparams, int persistent)
2. {
3.     php_strip_tags_filter *inst;
4.     smart_str tags_ss = {0};
5.
6.     inst = pemalloc(sizeof(php_strip_tags_filter), persistent);
```

```

7.
8.     if (inst == NULL) {
9.     }
10.
11.     if (filterparams != NULL) {
12.         ...
13.     }
14.
15.     if (php_strip_tags_filter_ctor(inst, ZSTR_VAL(tags_ss.s),
16. ZSTR_LEN(tags_ss.s), persistent) != SUCCESS) {
17.         smart_str_free(&tags_ss);
18.         pefree(inst, persistent);
19.         return NULL;
20.     }
21.     smart_str_free(&tags_ss);
22.     return php_stream_filter_alloc

```

The function `strfilter_strip_tags_create()` which was called should be the most important part of my analysis because `gdb` showed me that the segmentation fault happened in this block. The first argument, which was `filtername`, was still “string.strip_tags”, and the other two arguments were both `0x0`. The first two `if`-conditions were not met, so I ignored it here. When it came to the `if`-condition of `php_strip_tags_filter_ctor()`, **the result from backtrace gave the information that `php_strip_tags_filter_ctor` return with FAILURE**. But why?

```

1. static int php_strip_tags_filter_ctor(php_strip_tags_filter *inst, const char *allowed_tags, size_t allowed_tags_len, int persistent)
2. {
3.     if (allowed_tags != NULL) {
4.         if (NULL == (inst->allowed_tags = pemalloc(allowed_tags_len, persistent))) {
5.             return FAILURE;
6.         }
7.         memcpy((char *)inst->allowed_tags, allowed_tags, allowed_tags_len);
8.         inst->allowed_tags_len = (int)allowed_tags_len;
9.     } else {
10.        inst->allowed_tags = NULL;
11.    }
12.    inst->state = 0;
13.    inst->persistent = persistent;
14.
15.    return SUCCESS;

```

I found the definition of the function `php_strip_tags_filter_ctor` in the same file. Here I found the problem, the argument `allowed_tags` and `allowed_tags_len` came from the value of `ZSTR_VAL(tags_ss.s)` and `ZSTR_LEN(tags_ss.s)`. According to [Strings management: zend_string](#) from PHP internals books, the argument of `ZSTR_*` should be an address pointer. However,

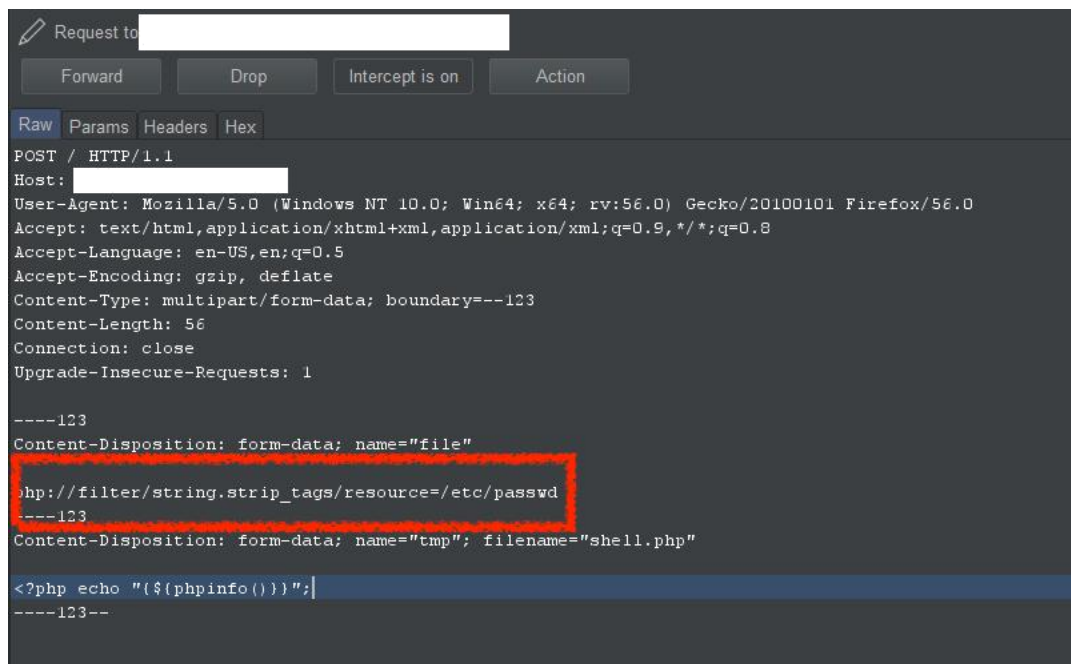
```

1. smart_str tags_ss = {0};

```

The argument was already set to zero before `ctor` function was called. Therefore, `ZSTR_*(0x0000)` would cause to an illegal pointer, which was segmentation fault in the end.

EXPLOIT



(This was the website which I could control the argument of include function. Sorry that I could not show the request link in the picture.)

Here I sent the POST of exploit, when PHP started to deal with multipart/form-data, the first form-data caused to segmentation fault. In the same time, the second form-data became a temporary file and was not cleaned up due to the segmentation fault (*Originally, the temporary file would be cleaned up in soon and you won't have opportunity to include it*). Therefore, I just needed to brute force on the temporary filename to implement local file inclusion attack.



The temporary filenames are all started with “php” and six random alphanumeric characters. To get the temporary filename with brute force, I just used my lovely tool, burpsuite, intruder started attack with six payload sets, each of wordlist had 62 kinds of letters, so I started attack with thread 5. There was also a trick to make the brute force success much faster, I sent the many POST of exploit to generate more and more temporary file at the same time of brute force. This should make the probability higher.

CONCLUSION OF VULNERABILITY

In conclusion, I could see that even segmentation fault could cause to the serious security problem. This problem of strip_tags filter had been fixed in the version of php 7.1.20.

Although the fix was not announced officially in the change log of [Version 7.1.20](#), I still found it in the commit of github.

```
276 277
277 - if (php_strip_tags_filter_ctor(inst, ZSTR_VAL(tags_ss.s), ZSTR_LEN(tags_ss.s), persistent) != SUCCESS) {
278 -     smart_str_free(&tags_ss);
278 + if (php_strip_tags_filter_ctor(inst, allowed_tags, persistent) == SUCCESS) {
279 +     filter = php_stream_filter_alloc(&strfilter_strip_tags_ops, inst, persistent);
280 + } else {
279 281     pefree(inst, persistent);
280 282     return NULL;
281 282 }
```

<https://github.com/php/php-src/commit/791f07e4f06a943bd7892bdc539a7313fb3d6d1e#diff-c2130880b5fed7a54a88f7997809ba50L277>

In the if-condition of `php_strip_tags_filter_ctor`, the zend string management didn't read from the illegal address pointer anymore. Instead, he used the variable of `allowed_tags` to replace it. The initialized value of the variable was `NULL`. Here came the definition of the function.

```
179 - static int php_strip_tags_filter_ctor(php_strip_tags_filter *inst, const char *allowed_tags, size_t allowed_tags_len, int persistent)
179 + static int php_strip_tags_filter_ctor(php_strip_tags_filter *inst, zend_string *allowed_tags, int persistent)
180 180 {
181 181     if (allowed_tags != NULL) {
182 -         if (NULL == (inst->allowed_tags = pemalloc(allowed_tags_len, persistent))) {
182 +         if (NULL == (inst->allowed_tags = pemalloc(ZSTR_LEN(allowed_tags) + 1, persistent))) {
183 183             return FAILURE;
184 184         }
185 -         memcpy((char *)inst->allowed_tags, allowed_tags, allowed_tags_len);
186 -         inst->allowed_tags_len = (int)allowed_tags_len;
185 +         memcpy((char *)inst->allowed_tags, ZSTR_VAL(allowed_tags), ZSTR_LEN(allowed_tags) + 1);
186 +         inst->allowed_tags_len = (int)ZSTR_LEN(allowed_tags);
187 187     } else {
188 188         inst->allowed_tags = NULL;
189 189     }
```

<https://github.com/php/php-src/commit/791f07e4f06a943bd7892bdc539a7313fb3d6d1e#diff-c2130880b5fed7a54a88f7997809ba50L179>

I could see that the line of 101 in screenshot above was still not changed. Therefore, I could be sure that the problem came from the reading illegal address of zend string.

REF LEC TION

In fact, I ran into several challenges in the process of analysis. These challenges also help me learn more skills which could help me analyze the problem more efficiently next time. I would like to share it with two points here.

WHAT IS SEGMENTATION FAULT

I found that it is important to figure out the definition of segmentation fault. I always ignored the cause of segmentation fault when I was a developer, but it was important for analyzing vulnerabilities. After googling for a while, I found that the cause of the segmentation fault can be categorized into several kinds in following:

1. out-of-bound memory access
2. thread safety

3. memory overflow
4. illegal pointer
5. mutex lock

This really helped me when I was debugging the problem of the code. I started to find whether there was any problem described above, and found the illegal pointer one in the end.

THE COMMIT OF THE PROBLEM FILE

It is also important to read the commit of the file which caused to the problem. From the commit showed for the file, I could see that the developer of PHP changed the use of `zend_*`() function to the variable of `allowed_tags`. Therefore, I could deduce that the problem was from `zend_*`() .

Besides, reading the official guide of can also solve many problems. For example, [PHP Manual](#) always put some attention on the guide of the function definition to make developer take care of their projects. Even nearly 80% of the vulnerabilities from PHP framework are caused from their mistake use of the function. Therefore, I consider reading the official guide is necessary for all the programmer. In conclusion, figuring out the definition of the problem itself and reading through the commit of the file are what I learn from this project to help me analyze problem.