#前言

这篇文章将用实战来表明如何用 gdb 从 coredump 恢复现场并游走在程序的堆栈之间。

#问题

双11之前，为了使用我的工具集systemtap-toolkit进行生产环境的活体分析, 需要给 php 开启调试符号，开启方式比较简单,如下所示，

```
./configure --enable-debug ....
```

重新打包发布测试，按理论应该不会有问题，毕竟只是开启调试符号，并没有修改 codebase。 预发测试没问题，ok，生产上线部分流量看看。WTF `PHP-FPM` 经常的报 coredump 。

因缺思婷, 开调试符号会造成 coredump ，还是第一次见。这个现象成功地吸引了我的注意力。

#载入coredump

扔出调试工具 gdb 查查原因。

先载入 coredump 恢复现场。

```
# gdb -c /tmp/core-500-php-fpm-29093-1478780406 $(php-fpm)
Excess command line arguments ignored. (/opt/isys/sbin/php-fpm ...)
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-90.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...

Core was generated by php-fpm: pool www.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000a6e6a5 in _zval_dtor_func (zvalue=0x7f676a8a93c8,
    __zend_filename=0x1090db8 "/data/users/app/rpmbuild/BUILD/php-5.6.16/Zend/zend_execute.h
36              CHECK_ZVAL_STRING_REL(zvalue);
```

#调用栈

看看具体的调用栈。

```
(gdb) bt
#0  0x0000000000a6e6a5 in _zval_dtor_func (zvalue=0x7f676a8a93c8,
    __zend_filename=0x1090db8 "/data/users/app/rpmbuild/BUILD/php-5.6.16/Zend/zend_execute.h
#1  0x0000000000a59abe in _zval_dtor (zvalue=0x7f676a8a93c8,
    __zend_filename=0x1090db8 "/data/users/app/rpmbuild/BUILD/php-5.6.16/Zend/zend_execute.h
#2  0x0000000000a59b8a in i_zval_ptr_dtor (zval_ptr=0x7f676a8a93c8,
    __zend_filename=0x1092fb0 "/data/users/app/rpmbuild/BUILD/php-5.6.16/Zend/zend_variables
#3  0x0000000000a5add2 in _zval_ptr_dtor (zval_ptr=0x7f676a8a96b0,
    __zend_filename=0x1092fb0 "/data/users/app/rpmbuild/BUILD/php-5.6.16/Zend/zend_variables
#4  0x0000000000a6ec1c in _zval_ptr_dtor_wrapper (zval_ptr=0x7f676a8a96b0)
    at /usr/src/debug/php-5.6.16/Zend/zend_variables.c:188
#5  0x0000000000a82bd1 in i_zend_hash_bucket_delete (ht=0x142c328, p=0x7f676a8a9698)
    at /usr/src/debug/php-5.6.16/Zend/zend_hash.c:182
#6  0x0000000000a82ca8 in zend_hash_bucket_delete (ht=0x142c328, p=0x7f676a8a9698)
    at /usr/src/debug/php-5.6.16/Zend/zend_hash.c:192
#7  0x0000000000a849d9 in zend_hash_graceful_reverse_destroy (ht=0x142c328)
    at /usr/src/debug/php-5.6.16/Zend/zend_hash.c:613
---Type <return> to continue, or q <return> to quit---
#8  0x0000000000a5a5f8 in shutdown_executor ()
    at /usr/src/debug/php-5.6.16/Zend/zend_execute_API.c:244
#9  0x0000000000a70e61 in zend_deactivate () at /usr/src/debug/php-5.6.16/Zend/zend.c:960
#10 0x00000000009d6b5d in php_request_shutdown (dummy=0x0)
    at /usr/src/debug/php-5.6.16/main/main.c:1883
#11 0x0000000000b3633b in main (argc=4, argv=0x7ffcc1123638)
    at /usr/src/debug/php-5.6.16/sapi/fpm/fpm/fpm_main.c:1992
```

看看产生 coredump 的代码, 注意第36行:)

```
(gdb) list _zval_dtor_func
27  #include "zend_constants.h"
28  #include "zend_list.h"
29
30
31  ZEND_API void _zval_dtor_func(zval *zvalue ZEND_FILE_LINE_DC)
32  {
33      switch (Z_TYPE_P(zvalue) & IS_CONSTANT_TYPE_MASK) {
34          case IS_STRING:
35          case IS_CONSTANT:
36              CHECK_ZVAL_STRING_REL(zvalue);
37              str_efree_rel(zvalue->value.str.val);
38              break;
39          case IS_ARRAY: {
40                  TSRMLS_FETCH();
41
42                  if (zvalue->value.ht && (zvalue->value.ht != &EG(symbol_table))) {
43                      /* break possible cycles */
44                      Z_TYPE_P(zvalue) = IS_NULL;
45                      zend_hash_destroy(zvalue->value.ht);
46                      FREE_HASHTABLE(zvalue->value.ht);
```

`CHECK_ZVAL_STRING_REL(zvalue);` 看起来是个宏，翻出 php-src 的源码查查 `CHECK_ZVAL_STRING_REL` 的定义.

#分析PHP源码

```
git clone https://github.com/php/php-src.git
git checkout php-5.6.16
```

用 cloc 统计下 php 的代码量:)

```
❯ cloc .                                                                                                2s
    1000 files
   14600 files
   14700 files

   15000 files
   15100 files
   17131 text files.
   16869 unique files.
   14491 files ignored.

http://cloc.sourceforge.net v 1.62  T=121.67 s (21.2 files/s, 11662.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                              996         102590         133526         841438
C/C++ Header                   749          20249          29558         184948
PHP                            448           4740           7718          23523
m4                             110           2421            763          18026
Bourne Shell                    35           1356           1863          12522
XML                             86            177            117           6263
C++                             20           1290            738           6198
Javascript                       3            537            219           2464
lex                              3            546            343           2317
Windows Module Definition        7              1              6           1918
yacc                             3            293            108           1564
Expect                          10            127             17           1515
Pascal                          29            260           1471           1058
make                            13            149             37            716
awk                             10             87             53            541
HTML                             7             37              0            368
Perl                             1             81             28            316
Windows Resource File            7            265             81            302
XSD                              9             33              0            265
XSLT                            12             16             24            160
D                                4             12             38            149
DOS Batch                        7             12              4             69
YAML                             2              8              3             59
DTD                              5             11              0             46
SQL                              1              4              0             40
NAnt script                      1              8              0             25
Windows Message File             1              4              0             24
Ant                              1              5              0             10
-------------------------------------------------------------------------------
SUM:                          2580         135319         176715        1106844
-------------------------------------------------------------------------------
```

看起来很大，估计用ag搜会搜出一坨:(。ok，拿 cscope 来查比较精确。

```
cscope -Rbq
cscope -dq

Global definition: CHECK_ZVAL_STRING_REL

  File       Line
0 zend_API.h 539 #define CHECK_ZVAL_STRING_REL(z) \
1 zend_API.h 543 #define CHECK_ZVAL_STRING_REL(z)



Find this C symbol:
Find this global definition: CHECK_ZVAL_STRING_REL
Find functions called by this
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

看看具体的`CHECK_ZVAL_STRING_REL`定义

```
539 #define CHECK_ZVAL_STRING_REL(z) \
540 ›   if (Z_STRVAL_P(z)[ Z_STRLEN_P(z) ] != '\0') { zend_error(E_WARNING, "String is not ze
```

看起来 PHP 宏用的飞起，呵呵。看看 Z_STRVAL_P 的定义(方法如上)

```
460 #define Z_STRVAL_P(zval_p)  Z_STRVAL(*zval_p)
461 #define Z_STRLEN_P(zval_p)  Z_STRLEN(*zval_p)
444 #define Z_STRVAL(zval)     (zval).value.str.val
445 #define Z_STRLEN(zval)     (zval).value.str.len
```

所以最后的表达式就是`(*zvalue).value.str.val[(*zvalue).value.str.len]`，再看看这几个变量的值。

```
(gdb) p (*zvalue).value.str.val
$3 = 0x0
(gdb) p (*zvalue).value.str.len
$4 = 0
```

显然，程序访问了非法的内存地址0x00.(有兴趣的自己查看程序布局cat /proc/xx/maps)。 造成 coredump 的代码找到了，现在得想办法找出复现条件，由于是在发生 coredump 三天之后才开始调试的，之前的现场只有一份 coredump 。

#复现现场

曾经用 Node.js 实现过 fastcgi 协议，我猜测代码里应该有保存请求信息的对象, 仔细看调用栈，看起来入口比较像。

```
(gdb) frame 11
#11 0x0000000000b3633b in main (argc=4, argv=0x7ffcc1123638)
    at /usr/src/debug/php-5.6.16/sapi/fpm/fpm/fpm_main.c:1992
1992                php_request_shutdown((void *) 0);
```

看看具体的代码是啥

```
(gdb) list /usr/src/debug/php-5.6.16/sapi/fpm/fpm/fpm_main.c:1992
1987                fpm_log_write(NULL TSRMLS_CC);
1988
1989                STR_FREE(SG(request_info).path_translated);
1990                SG(request_info).path_translated = NULL;
1991
1992                php_request_shutdown((void *) 0);
1993
1994                requests++;
1995                if (max_requests && (requests == max_requests)) {
1996                    fcgi_finish_request(&request, 1);
```

`SG(request_info)` 看起来比较像:)，看看 `SG` 是啥

```
147 # define SG(v) (sapi_globals.v)
```

从命名方式来看像是全局变量，打印出值看看:) (由于隐私问题，部分信息被修改，但不影响)

```
(gdb) p sapi_globals.request_info
$1 = {request_method = 0x7f676a876560 "POST",
  query_string = 0x7f676a876430 "qqqq=123",
  cookie_data = 0x7f676a878250 "a=1;b=2"..., content_length = 0, path_translated = 0x0,
  request_uri = 0x7f676a876880 "/yyyy", request_body = 0x7f676a8a8e98,
  content_type = 0x7f676a876670 "", headers_only = 0 '\000', no_headers = 0 '\000',
  headers_read = 0 '\000', post_entry = 0x0, content_type_dup = 0x7f676a8a8dc0 "",
  auth_user = 0x0, auth_password = 0x0, auth_digest = 0x0, argv0 = 0x0, current_user = 0x0,
  current_user_length = 0, argc = 0, argv = 0x0, proto_num = 1000}
```

Good，试试复现构造这个请求。

```
curl -X POST -H "Cookie: a=1;b=2" -H "Content-Length: 0" http://127.0.0.1/yyy?qqqq=123
```

果然，POST 一个 request body 为空的请求比较少见，竟然造成 coredump :(

到目前为止，复现方法和 coredump 原因都已经找到，再继续挖掘调用栈看看，从外到里一层层看，在 `frame 8` 发现新线索:)。

#新线索

```
(gdb) frame 8
#8  0x0000000000a5a5f8 in shutdown_executor ()
    at /usr/src/debug/php-5.6.16/Zend/zend_execute_API.c:244
244         zend_hash_graceful_reverse_destroy(&EG(symbol_table));
```

看语义是在销毁 `&EG(symbol_table)` 这个 hash 造成的，看看具体 hash 里是什么:)

```
47 # define EG(v) (executor_globals.v)
```

看起来又是一个全局变量，打印出看看：）

```
(gdb) p executor_globals.symbol_table
$2 = {nTableSize = 64, nTableMask = 63, nNumOfElements = 0, nNextFreeElement = 0,
  pInternalPointer = 0x0, pListHead = 0x0, pListTail = 0x0, arBuckets = 0x7f676a8a9440,
  pDestructor = 0xa6ebf8 <_zval_ptr_dtor_wrapper>, persistent = 0 '\000',
  nApplyCount = 0 '\000', bApplyProtection = 1 '\001', inconsistent = 0}
```

结构比较复杂，从成员的命名方式可以知道应该是个链表。看看具体的`symbol_table`类型

```
168 struct _zend_executor_globals {
169 ›   zval **return_value_ptr_ptr;
170 
171 ›   zval uninitialized_zval;
172 ›   zval *uninitialized_zval_ptr;
173 
174 ›   zval error_zval;
175 ›   zval *error_zval_ptr;
176 
177 ›   /* symbol table cache */
178 ›   HashTable *symtable_cache[SYMTABLE_CACHE_SIZE];
179 ›   HashTable **symtable_cache_limit;
180 ›   HashTable **symtable_cache_ptr;
181 
182 ›   zend_op **opline_ptr;
183 
184 ›   HashTable *active_symbol_table;
185 ›   HashTable symbol_table;››   /* main symbol table */
```

果然跟之前说的一样，是个`HashTable`。既然是`HashTable`，应该有记录 hash 对应的 key 值, 看看`HashTable`的定义

```
 67 typedef struct _hashtable {
 68 ›   uint nTableSize;
 69 ›   uint nTableMask;
 70 ›   uint nNumOfElements;
 71 ›   ulong nNextFreeElement;
 72 ›   Bucket *pInternalPointer;›  /* Used for element traversal */
 73 ›   Bucket *pListHead;
 74 ›   Bucket *pListTail;
 75 ›   Bucket **arBuckets;
 76 ›   dtor_func_t pDestructor;
 77 ›   zend_bool persistent;
 78 ›   unsigned char nApplyCount;
 79 ›   zend_bool bApplyProtection;
 80 #if ZEND_DEBUG
 81 ›   int inconsistent;
 82 #endif
 83 } HashTable;
```

看`pListHead`和`pListTail`，看样子所有的变量都在这个链表连起来的桶上，看看`Bucket`的定义

```
55 typedef struct bucket {
56 ›   ulong h;›   ›   ›   ›   ›   ›   /* Used for numeric indexing */
57 ›   uint nKeyLength;
58 ›   void *pData;
59 ›   void *pDataPtr;
60 ›   struct bucket *pListNext;
61 ›   struct bucket *pListLast;
62 ›   struct bucket *pNext;
63 ›   struct bucket *pLast;
64 ›   const char *arKey;
65 } Bucket;
```

`arKey` 应该就是我们要找的 key。ok，调用栈上应该有具体的某个Bucket，往里继续找着:)

```
(gdb) frame 7
#7  0x0000000000a849d9 in zend_hash_graceful_reverse_destroy (ht=0x142c328)
    at /usr/src/debug/php-5.6.16/Zend/zend_hash.c:613
613         zend_hash_bucket_delete(ht, ht->pListTail);

(gdb) list /usr/src/debug/php-5.6.16/Zend/zend_hash.c:613
608 ZEND_API void zend_hash_graceful_reverse_destroy(HashTable *ht)
609 {
610     IS_CONSISTENT(ht);
611
612     while (ht->pListTail != NULL) {
613         zend_hash_bucket_delete(ht, ht->pListTail);
614     }
615
616     if (ht->nTableMask) {
617         pefree(ht->arBuckets, ht->persistent);
```

看代码是在遍历 `ht->pListTail` 的某一个值导致的，继续往里看看发生 coredump 时的 `ht->pListTail` 值是多少

```
#6  0x0000000000a82ca8 in zend_hash_bucket_delete (ht=0x142c328, p=0x7f676a8a9698)
    at /usr/src/debug/php-5.6.16/Zend/zend_hash.c:192
192     i_zend_hash_bucket_delete(ht, p);
```

打印出来看看

```
(gdb) p p
$4 = (Bucket *) 0x7f676a8a9698
(gdb) p *p
$5 = {h = 12508711195225833452, nKeyLength = 19, pData = 0x7f676a8a96b0,
  pDataPtr = 0x7f676a8a93c8, pListNext = 0x0, pListLast = 0x0, pNext = 0x0, pLast = 0x0,
  arKey = 0x7f676a8a96e0 "HTTP_RAW_POST_DATA"}
```

奈斯，看样子是在销毁 "HTTP_RAW_POST_DATA" 变量导致的 coredump ，看看 HTTP_RAW_POST_DATA 那里定义:)

```
Text string: HTTP_RAW_POST_DATA

  File                Line
0 php_var.h           157 (name_len == sizeof("HTTP_RAW_POST_DATA") - 1 && !memcmp(name,
                          "HTTP_RAW_POST_DATA", sizeof("HTTP_RAW_POST_DATA") - 1)) ||
1 php_content_types.c  69 "HTTP_RAW_POST_DATA truncated from %lu to %d bytes",
2 php_content_types.c  73 SET_VAR_STRINGL("HTTP_RAW_POST_DATA", data,

);
3 php_content_types.c  76 "Automatically populating $HTTP_RAW_POST_DATA is deprecated and "



Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:

 59 ›   ›   if (populate_raw_post_data(TSRMLS_C)) {
 60 ›   ›   ›   size_t length;
 61 ›   ›   ›   char *data = NULL;
 62
 63 ›   ›   ›   php_stream_rewind(SG(request_info).request_body);
 64 ›   ›   ›   length = php_stream_copy_to_mem(SG(request_info).request_body, &data, PHP_ST
 65 ›   ›   ›   php_stream_rewind(SG(request_info).request_body);
 66
 67 ›   ›   ›   if (length > INT_MAX) {
 68 ›   ›   ›   ›   sapi_module.sapi_error(E_WARNING,
 69 ›   ›   ›   ›   ›   "HTTP_RAW_POST_DATA truncated from %lu to %d bytes",
 70 ›   ›   ›   ›   ›   (unsigned long) length, INT_MAX);
 71 ›   ›   ›   ›   length = INT_MAX;
 72 ›   ›   ›   }
 73 ›   ›   ›   SET_VAR_STRINGL("HTTP_RAW_POST_DATA", data, length);
```

注意第73行 `data` 和 `length`，跟我们之前 `(*zvalue).value.str.val[(*zvalue).value.str.len]` 的相对应， 当我们发起一个 POST Request Body为空的请求时，data将为空,(这里可以直接调试php解释器验证)，ok,真正的根源找到了看看怎么修复。 直觉告诉我 `(*zvalue).value.str.val[(*zvalue).value.str.len]`，之前应该加个判断，不过在不熟悉源码的情况下，还是改 HTTP_RAW_POST_DATA 比较安全：） 当 data 为 NULL 并且 length 为 0 时，只要给 data 赋予一个合法的地址，并保证Z_STRVAL_P(z)[Z_STRLEN_P(z)] 为 '\0' 即可，最粗暴的方式直接用 `data = "";` 试试。

重新编译打包依然 coredump，不过这次 coredump 的原因不一样。

```
#0  0x0000000000a34a18 in zend_mm_check_ptr (heap=0x30bf2d0, ptr=0x107db83, silent=1,
    __zend_filename=0x1090558 "/root/rpmbuild/SOURCES/php-5.6.16/Zend/zend_execute.h",
    __zend_lineno=79,
    __zend_orig_filename=0x1092700 "/root/rpmbuild/SOURCES/php-5.6.16/Zend/zend_variables.c"
1384        if (p->info._size != ZEND_MM_NEXT_BLOCK(p)->info._prev) {
```

看起来不是堆上的地址造成的 coredump ， PHP 应该有一套自己的内存管理机制。没办法了，只能找找对应的 API，按经验试试搜索 empty 和 str。

```
Text string: empty

  File                            Line
0 zend.c                          326 if (Z_STRLEN_P(expr) == 0) { /* optimize away empty
                                      strings */
1 zend.c                          525 zend_hash_copy(compiler_globals->auto_globals,
                                      global_auto_globals_table, NULL, NULL,
                                      sizeof(zend_auto_global) /* empty element */);
2 zend.h                          682 #define STR_EMPTY_ALLOC() CG(interned_empty_string)?
                                      CG(interned_empty_string) : estrndup("", sizeof("")-1

* Lines 1-4 of 2347, 2344 more - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

Good Luck，看 `STR_EMPTY_ALLOC()` 应该是, 加了以下代码直接重新编译测跑测试通过。

```
if (data == NULL && length == 0) {
    data = STR_EMPTY_ALLOC();
}
```

#总结

如果之前熟悉 php 源码，拿到调用栈直接打印 hash 的 key，估计十几分钟内就完事, 最后不得不花了几个小时才搞定。 一个悲伤的事是当我准备顺手提交 PR 给 PHP 官方时，搜了下 Changelog， 发现早有人提交并修复了这个问题, 如果你的情况比较紧急，建议还是看看新版本是否修复吧，至于我只是为了好玩哈哈。