Iconv, set the charset to RCE: Exploiting the glibc to hack the PHP engine (part 3)

18-23 minutes

Introduction

A few months ago, I stumbled upon a 24 years old buffer overflow in the glibc, the base library for linux programs. Despite being reachable in multiple well-known libraries or executables, it proved rarely exploitable — while it didn't provide much leeway, it required hard-to-achieve preconditions. Looking for targets lead mainly to disappointment. On PHP however, **the bug shone**, and proved useful in exploiting its engine in two different ways.

In part one, I demonstrated how you could convert a file read to remote code execution using CVE-2024-2961. The exploit, however, relied on the output of the file read primitive. We'll now cover, in this third and final part, how one can exploit the same bug, but blind: without having any output!

And we'll do it with the same conditions than in the first part: we want to build a crashless, generic exploit, and with a relatively small payload, to be usable in GET requests (inferior to 7000 bytes).

How lucky we were

Precedently, I explained how you could use CVE-2024-2961 to build an exploit for code such as this one:

```
echo file_get_contents($_REQUEST['file']);
```

The exploit relied on three steps:

- First, read `/proc/self/maps` to find the address of the glibc and the PHP heap, making PIE and ASLR useless in the process;
- Then, read `libc.so` to extract the address of `system()`, `malloc()`, and `realloc()`;
- Finally, send a carefully crafted `php://filter/...` payload which corrupts the heap, overwrites function pointers, and eventually executes `system("...")`.

Getting the output of the `file_get_contents()` function was **required** for the first two steps, that provided crucial information for the third one.

However, oftentimes, we have a file read vulnerability, but without any output. We can face code such as this, for instance:

```
if(md5_file($_REQUEST['file']) === '8f199aebac0036c0c1fa2304eecc3d54') {
    echo "Valid file";
} else {
    echo "Invalid file";
}
```

Is it still exploitable? Well, people have demonstrated that it is possible to remotely dump files using an error-based oracle. We could, in theory, use this to dump `/proc/self/maps` and the LIBC. However, as you may know, the technique is lackluster when it comes to large files: the `php://filter/` chains that it build becomes very, very big, fast exceeding the limits imposed by HTTP servers (for instance, the ~8000 characters limitation for URLs). And the LIBC *is* large (2.2MB on my system). Even if we perform compression beforehand, using the `zlib.deflate` filter, it still amounts to a gigantic number of bytes, far out of the realm of possibility. In any case,

another blocking factor that we would like to circumvent is the use of `open_basedir`, which limits the directories from which you can read files.

As a consequence, to **exploit this vulnerability, blind**, we need a different algorithm, way closer to standard binary exploit: we have to get a leak, then an arbitrary read, and finally use the information obtained to perform a write and get code execution. And that means that things are going to get a little bit harder.

Exploitation idea

We're not starting from scratch, however! If we figure out the address of the main PHP heap, and of `system()`, `malloc()`, and `realloc()` (in the libc), we can then use the same technique as in the « sighted » exploit described in part 1: change `zend_mm_heap.custom_heap._free` to `system`, and free an arbitrary chunk to get code execution.

Since we cannot depend on files to get this information, we can instead rely on the data contained in the process' memory. If we manage to build a primitive to dump parts of it, we can sequentially leak pointers to different memory regions and parse them to get our prize. In other words, the goal is to get an **arbitrary read primitive**, and use it to parse everything to death until we get the information we are looking for.

To this end, we need to carefully manipulate PHP's heap using a `php://filter` string. In part 1, we learned how PHP represented stream data in memory, using a *bucket brigade*, a linked list of *buckets*, which each contained a *buffer* and a size. By carefully picking our filters and the data, we learned how to:

- allocate as many buckets (byte buffers) as we want using the `zlib.inflate` filter;
- pick their sizes arbitrarily using the `dechunk` filter;
- make them « appear » and « disappear » by creating `dechunk` russian dolls, allowing us to change their size several times.

Overwriting buckets

By chance, the C structure that represents buckets, called `php_stream_bucket`, is going to be crucial for our exploitation.

```
typedef struct _php_stream_bucket {
    php_stream_bucket *next;
    php_stream_bucket *prev;
    php_stream_bucket_brigade *brigade;
    char *buf;
    size_t buflen;
    uint8_t own_buf;
    uint8_t is_persistent;
    int refcount;
} php_stream_bucket;
```

Buckets are linked together through their `prev` and `next` fields, and hold a reference to their `brigade`. They hold data to which their `buf` field points to, and whose size is stored in `buflen`. Buckets and their buffer are allocated on the heap.

If we managed, using our memory corruption, to **control a bucket structure**, it would allow us to leak arbitrary parts of memory: we could make `buf` point wherever we like, and then **use the php_filter_chains_oracle_exploit to leak memory**, byte-per-byte. A nice **combination** of web and binary.

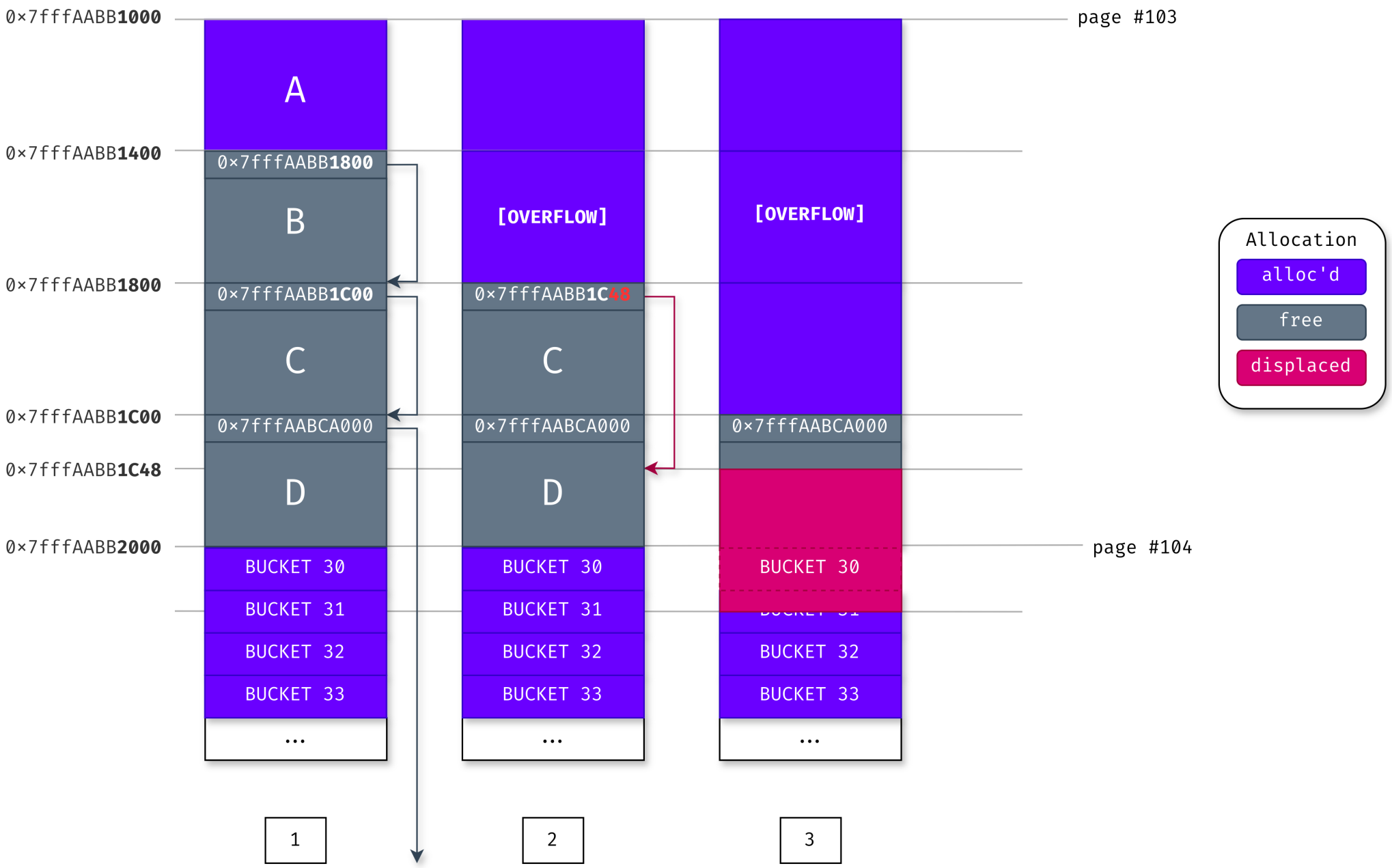
Precise goal

We want to alter a `php_stream_bucket` structure in the heap, in order to change the address of its buffer, and then use `php_filter_chains_oracle_exploit` (abbreviated as *PFCOE*) to dump the stream. Now, since *PFCOE* works with base64, we want to set `buflen` to *3*, in order to have an aligned base64 quartet to dump. The design needs to be very stable, because *PFCOE* will issue several requests to dump each B64 digit, each time **altering the looks of the PHP heap**. And, if somehow our memory corruption fails in some cases, the oracle would be wrong, and the exploit would fail.

In addition to this, we want the payloads sent by the exploit to fit in an URL. In other words, we **don't want their size to exceed ~7000 bytes**. In the « sighted » exploit, we used a combination of 12 filters, and as such the final payload was around 1000 bytes long, which is very small. However, here, we use *PFCOE* to dump bytes, and the tool is very character-hungry, especially when it tries do dump big streams: the farther a base64 digit is from the beginning of a stream, the bigger the `php://filter` payload will get.

Overwriting a 0x30 chunk

To overwrite a bucket, we setup the heap to have a page of *0x400* chunks, and a page for *0x30* chunks (the size of `php_stream_bucket` structures) right under. We have 4 chunks in the top page, `A`, `B`, `C` and `D`. We then use the same technique as in part 1 to displace chunk `D` and make it overlap with the 0x30 page. To understand the idea in more details, let's look at this three step process. First, let's say that `A` is allocated (step 1), and that the bottom page is filled with buckets:



Overwriting a bucket using a 0x400 chunk

We can then use the `convert.iconv.UTF-8.ISO-2022-CN-EXT` filter to trigger *CVE-2024-2961* from `B`, and make it overflow for one byte, right into the first byte of `C`, and thus overwrite its `next` pointer's LSB, making it point to `D+48h` (step 2). The second chunk of the free list is now *48h* bytes lower, and therefore overlaps with the first bytes of the next page. Since *0x400* chunks can hold data of size *0x381* to *0x400* (data smaller gets allocated using the *0x380* bin), we can use this displaced chunk to **overwrite part or the entirety of the `php_stream_bucket` structure** located at the top of the page underneath.

Lost brigade

Heap setup aside, the exploitation seems pretty straightforward: we can just partially overwrite a bucket's `buf` and make it point a little bit higher or lower, leaking part of the heap. We'd extract some pointers from it, then use these pointers to leak other memory regions, then others, and so on.

Things are not that simple, sadly: to partially overwrite `buf`, we need to completely overwrite the fields that come before.

```
typedef struct _php_stream_bucket {
    php_stream_bucket *next;
    php_stream_bucket *prev;
    php_stream_bucket_brigade *brigade;
    char *buf; // <----- target
    size_t buflen;
    uint8_t own_buf;
    uint8_t is_persistent;
    int refcount;
} php_stream_bucket;
```

`next` and `prev` can be set to `NULL` without too much trouble, but then comes `brigade`. Although no crash will happen if we nullify it too, this will essentially make our bucket **impossible to remove from the linked list**, producing an infinite loop. Eventually, the process would timeout and abort, but we would not be able to make use of our altered bucket.

As a result, before we get anything going, we need to **leak the address of the bucket brigade**.

Leaking the brigade

Now, if we didn't care about crashing PHP, this would not be too much of a problem *(remember, if a worker crashes, PHP respawns a new one, with the same memory layout)*. We could overwrite the least-significant byte of the address of the brigade, and cross fingers: if PHP crashes, or enters an infinite loop, it means that we messed up the byte. If it does not, we guessed correctly, and can go to the

second byte, then the third, and so on.

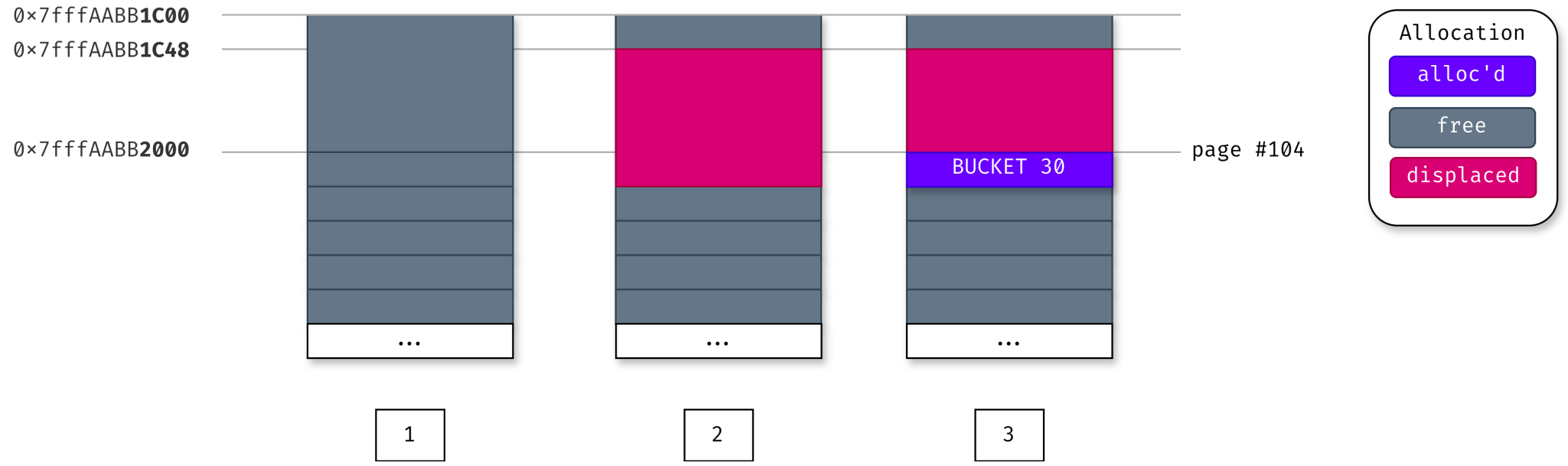
It'd be decently **fast**, super **reliable**, and would make the exploit relatively **easy** to implement. But it would produce **hundreds of crashes**, which would be a clear indicator of exploitation. Also, it is a web exploit, which by my book forbids us from crashing the server. Therefore, I chose to go crashless.

Crashless exploitation

Note: again, we're scratching the surface here, but writing down every intricacy of the exploit would make this blogpost even longer. I have greatly commented the exploit, if you want more details.

Leaking the brigade

To leak the `brigade` field of a `bucket`, we will simply inverse the order of allocations between the displaced chunk in the top page and the bucket on the bottom page. The `php_stream_bucket` structure would therefore get « inprinted » into the buffer!



Inprinting a bucket structure into a 0x400 buffer

Suppose that we have successfully modified the *0x400* free list so that the head chunk is at `0x7fffAABB1C48` instead of `0x7fffAABB1C00`, and thus overlaps with the next page, which, at this point, contains **free chunks** of size *0x30* (step 1). We then allocate a buffer in said chunk (step 2), and then allocate **lots of buckets**. One will overlap with the buffer and be **written into it** (step 3).

If we were able to read the whole stream, we'd be able to see the bucket structure, and its pointer to the brigade. But we can't. Therefore, we need to extract it. To do this, we use *PFCOE*, but we first have to move the leaked data to the beginning of the stream – otherwise, as mentioned before, the payload gets too big.

To demonstrate how we can do this with minimal effort, let us visualize the stream when the *inprinting* happens.

To perform the overflow, we created chunks `B`, `C`, and `D+0x48h`. We thus have at least 3 buckets of size *0x400*. In addition, we forced PHP to allocate buckets in the page underneath by creating lots of buckets whose buffer have a size of *0x30*. Let us imagine, first, that each buffer is filled with null bytes. We have:

	ASCII	HEXADECIMAL	
Bucket #1	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×400 bytes
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	...	...	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Bucket #2	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×400 bytes
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	...	...	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
Bucket #3	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×3e8 bytes
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	...	...	
	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
	·Pe· .... ·Pe·	**f0 50 65 81 a5 7f 00 00 c0 50 65 81 a5 7f 00 00**	inprinted bucket
	0·4u .... ·4i·	**30 90 34 75 fc 7f 00 00 00 34 69 81 a5 7f 00 00**	
	.... ··33	**00 04 00 00 00 00 00 00 01 00 33 33 01 00 00 00**	
Bucket #4	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #5	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #6	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #7	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #8	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
		...	
Bucket #210	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #211	....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes

A `php_stream_bucket` structure gets inprinted into a stream bucket buffer

In the last *0x400* buffer, at offset (*0x400-0x48* =) *0x3b8*, a bucket structure gets inprinted. How do we get rid of the contents of the previous two buckets, and most of the third one? Well, we can, again, make use of `dechunk` to prefix the leaked data with a size, padded with thousands of zeroes:

	ASCII	HEXADECIMAL	
Bucket #1	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #2	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #3	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 030↲ ·Pe· ···· ·Pe· ···· 0·4u ···· ·4i· ···· ···· ···· ··33 ····	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 **33 30 0A** **f0 50 65 81 a5 7f 00 00 c0 50 65 81 a5 7f 00 00** **30 90 34 75 fc 7f 00 00 00 34 69 81 a5 7f 00 00** **00 04 00 00 00 00 00 00 01 00 33 33 01 00 00 00**	0×3e8 bytes inprinted bucket
Bucket #4	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #5	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #6	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #7	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #8	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
...			
Bucket #210	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #211	↲0↲· ···· ···· ····	**0A 30 0A** 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes

A dechunkable structure that keeps only the leaked bucket bytes

The stream now contains:

```
0000000000000000..0030↲
<fake bucket structure>
↲0↲...
```

That's it! A single additional filter, `dechunk`, now removes everything but the leak, which we can then dump using *PFCOE*.

Arbitrary read

With the address of the leaked brigade in the bag, we are now able to completely control a bucket, and as such control its `buf` and `buflen` fields. Finally! Let us visualize the stream again. This time, we use the displaced chunk to overwrite a the first `php_stream_bucket` structure of the page underneath: we set its `next` and `prev` to `NULL`, keep the `brigade` intact, and set `buf` to `0x112233445566` and `buflen` to `3`.

	ASCII	HEXADECIMAL	
Bucket #1	···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· … ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 … 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×400 bytes
Bucket #2	···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· … ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 … 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×400 bytes
Bucket #3	···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· ···· … ···· ···· ···· ···· ·Pe· ···· ·Pe· ···· 0·4u ···· fUD3 "··· ···· ···· ··33 ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 … 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30 90 34 75 fc 7f 00 00 66 55 44 33 22 11 00 00 03 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00	0×3e8 bytes fake bucket
Bucket #4	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #5	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #6	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #7	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #8	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
…			
Bucket #189	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #190	???	?? ?? ??	0×3 bytes
Bucket #191	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
…			
Bucket #210	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #211	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes

Overwriting bucket #190 using a bucket buffer

We end up overwriting one of the *0x30* buckets. Since we null its `next` pointer, it becomes the last bucket of the chain, discarding the ones that come after.

We get our leak, but with lots of extraneous data in front. To remove it, we need to play with `dechunk` again: the overwritten bucket (bucket *#190* in the example) needs to be preceded by a special chunk (that contains ↲, indicating the end of the `dechunk` size header).

	ASCII	HEXADECIMAL	
Bucket #1	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #2	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #3	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ... 0000 0000 0000 003- ·Pe· ···· ·Pe· ···· 0·4u ···· fUD3 "··· ···· ···· ··33 ····	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ... 30 30 30 30 30 30 30 30 30 30 30 30 30 30 **33 2D** **f0 50 65 81 a5 7f 00 00 c0 50 65 81 a5 7f 00 00** **30 90 34 75 fc 7f 00 00 66 55 44 33 22 11 00 00** **03 00 00 00 00 00 00 00 01 00 33 33 01 00 00 00**	0×3e8 bytes fake bucket
Bucket #4	@@@@ @@@@ @@@@ @@@@	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40	0×30 bytes
Bucket #5	@@@@ @@@@ @@@@ @@@@	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40	0×30 bytes
Bucket #6	@@@@ @@@@ @@@@ @@@@	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40	0×30 bytes
Bucket #7	@@@@ @@@@ @@@@ @@@@	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40	0×30 bytes
Bucket #8	@@@@ @@@@ @@@@ @@@@	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40	0×30 bytes
...			
Bucket #189	@@@@ @@@@ @@@@ @@@↲	40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 **0A**	0×30 bytes
Bucket #190	???	?? ?? ??	0×3 bytes
Bucket #191	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
...			
Bucket #210	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes
Bucket #211	···· ···· ···· ····	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0×30 bytes

Overwriting bucket #190 using a 0x400 chunk, and making the stream dechunkable

The stream becomes (with ??? indicating the 3 leaked bytes):

```
0000000000000000..0003-
<fake bucket structure>
@@@@@@@@@@...@@@@@@@@@@↲
???
```

Due to the generousness of the implementation of the `dechunk` algorithm by PHP, applying the filter results in removing everything but the last 3 bytes.

But we have a problem: we don't know which bucket we overwrote! In the example, it is the 190th, but depending on the state of the heap, it could be any.

Luckily, this can easily be determined using an extra step. We put a special bucket that contains `0↲` at some offset `i`: if after the `dechunk`, the stream is empty, the bucket we overwrite is after `i`; otherwise, it is before. Proceeding via dichotomy, this only adds a few requests to the whole exploit, which is completely acceptable.

When we get this final piece of information, we can build dechunkable structure, and keep only the wanted 3 bytes of our leak.

At last! We can **read arbitrary memory**. But is that implementation good enough?

Hanging by a thread

Although the implementation does very well in lab conditions, I could not help but feel **unsatisfied**.

See, leaking bytes takes time. Along a long period of time, a PHP script's environment may change, and it may allocate a few more, or a few less, memory chunks. That makes the exploit **hang by a thread**: if along the few minutes it runs in, the PHP script makes as much as one more (or one less) allocation of size *0x30*, the overwritten chunk's offset will change, and mess up our `dechunk` setup. The oracle test will fail, and as a result, the exploit will incorrectly leak one byte, and eventually **fail**.

Luckily, I was able to find a more elegant solution, that very much hardened the exploit. The idea was to make the buckets have a size (`buflen`) of zero right before we overwrite one of them. It may seem like a trivial idea, but it is actually hard to pull off: most filters will actually delete a bucket if their `buflen` is zero (which makes sense: why keep an empty bucket?). The final stream would look like so:

	ASCII	HEXADECIMAL	
Bucket #1	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 … 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 … 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #2	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 … 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 … 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0×400 bytes
Bucket #3	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 … 0000 0000 0000 003- ·Pe· ···· ·Pe· ···· 0·4u ···· fUD3 "··· ···· ···· ··33 ····	30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 … 30 30 30 30 30 30 30 30 30 30 30 30 30 30 **33 2D** **00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00** **30 90 34 75 fc 7f 00 00 66 55 44 33 22 11 00 00** **03 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00**	0×3e8 bytes fake bucket
Bucket #4	↲	0A	1 byte
Bucket #5			0 bytes
Bucket #6			0 bytes
Bucket #7			0 bytes
Bucket #8			0 bytes
…			
Bucket #189			0 bytes
Bucket #190	???	?? ?? ??	0×3 bytes
Bucket #191			0 bytes
…			
Bucket #210			0 bytes
Bucket #211			0 bytes

The attack then becomes resistant to changes in the heap structure, relying only on the number of *0x400* allocations not to change. Luckily for us, this chunk size is not much used in the PHP engine.

Finding `system()`

Armed with our clean, reliable arbitrary read, we can dump anything of relevance to get code execution. Sadly, the process of leaking bytes is extremely slow. I got ~2 bytes per second during my tests (although admittedly I was attacking a gdb-attached, single worker apache process). That makes browsing the code looking for ROP gadgets seem completely unrealisable. In any case, we cannot make any assumption regarding the PHP version or the CPU architecture: we want a generic exploit.

I was not able to find a simple « one-gadget » type exploit solely contained in the PHP binary, so I did the same as in the first exploit: I looked for the address of `system()`, `malloc()`, and `realloc()`, in order to overwrite the `zend_mm_heap.custom_heap` structure. Here are the steps the exploit goes through to find the required addresses:

- Leak the head of the bucket brigade to get the address of some PHP heap block `zend_mm_chunk`;
- Leak the address of the main heap `zend_mm_heap` from the heap block;
- Parse the heap metadata to find a page that contains `zend_array` structures;
- Find such a structure in the page, and extract its `pDestructor` field, which contains `zval_ptr_dtor`;
- Walk back to the top of the PHP ELF from `zval_ptr_dtor`'s address;
- Parse the ELF's program headers, looking for the *dynamic* section;
- Find *STRTAB*, *SYMTAB*, and *JMPREL* segments;
- Find the address of any function of the LIBC using these segments;
- Walk back to the top of the LIBC ELF from that function's address;
- Parse the ELF's program headers, looking for the *dynamic* section;
- Find *STRTAB*, *SYMTAB*, and *GNU_HASH* segments;
- Find **the addresses of `system()`, `malloc()`, and `realloc()`** in the hashtable.

After everything has been dumped, we get the address of the PHP main heap and `system()`. The exploit issues a final request to get **code execution**, in the same fashion as in the original exploit.

Fast-tracking

As you may have guessed, this is not very **fast**, but it yielded interesting optimisation problems that I had to tackle. One of them is that using the API of *PFCOE* directly is inefficient, because we oftentimes don't want to dump memory, but compare it to a value. For instance, when looking for the ELF header of the PHP binary, we iterate over pages and compare their first 3 bytes to `b'\x7fEL'`.

A naive implementation consists in dumping 4 base64 digits (3 bytes), and compare them to `f0VM`. A better implementation would compare the base64 digits one by one. Dump the first digit, compare it to `f`, and continue. However, the most efficient implementation is to build tests that directly ask the server if the character is `f`. As a result, the dozens of requests required to compare memory to a value would drop to one or two. I implemented this fast-tracking technique for a few redundant base64 digits and the speed of the exploit greatly increased.

However, the exploit is still **not that fast**: in my lab conditions, it runs for around 15 minutes. Some parts could be sped up using multithreading, however.

The exploit

The exploit is available on our Github. Bear in mind that **this is a POC**; it will not always work out of the box. I'm however confident that it it oftentimes will, and hopeful that it will remain, in other cases, a very good starting point for motivated individuals.

Since the exploit runs for so long, each value of importance can be set using CLI arguments, allowing you to run it in several iterations.

Here is a demo of the complete exploit. It runs for 17 minutes (sped up 8 times):

Conclusion

This third part closes the blog series on iconv, PHP, and CVE-2024-2961. This 25 year old bug, which did not seem like much at first, yielded two exploit cases: one giving a new attack vector for file read primitives, that prompted the first and the third part of the series, exploiting `php://filter`, and the second, through direct `iconv()` calls, that allowed to showcase a Roundcube RCE.

More importantly, it provided a way to get into the intricacies of PHP, understanding its engine, and showcasing the possibilities of remote, binary exploitation on this beautiful language.