Iconv, set the charset to RCE: Exploiting the glibc to hack the PHP engine (part 2)

34-43 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 introduced the bug by recounting its discovery, its limitations, and I demonstrated its use on PHP by converting file read vulnerabilities to RCEs. In this blog post, I will explore a new way of exploiting the vulnerability on PHP, using direct calls to `iconv()`, and illustrate the vulnerability by targeting Roundcube, a popular PHP webmail. Again, I will demonstrate the impact on the ecosystem, by revealing unexpected ways of reaching `iconv()` when using mbstring.

If you are not familiar with web exploitation, PHP, or the PHP engine, mind not: I will explain relevant notions along the way.

Another trigger

While being able to trigger the bug using `php://filter` is very convenient, the most obvious way to reach a call to `iconv()` it to use its eponymous API. In PHP, it has the following prototype:

iconv

(PHP 4 >= 4.0.5, PHP 5, PHP 7, PHP 8)

iconv — Convert a string from one character encoding to another

Description

```
iconv(string $from_encoding, string $to_encoding, string $string): string|false
```

Converts **string** from **from_encoding** to **to_encoding**.

The difference between this function and its C equivalent is that the buffer management, which in C has to be done by the caller, is now invisible, as it is handled, under the hood, by PHP. In part 1, we learned that we were very dependant on the looks of the output buffer: in many cases, the bug may very well be impossible to exploit.

So, is the `iconv()` implementation of PHP vulnerable? When we convert a string of size *N* to another charset using `iconv()`, PHP allocates an output buffer of size *N+32*, in the hope of « avoid(ing) realloc() (in) most cases » [1]. If the buffer is not big enough [2], it is made bigger [3].

```
// ext/iconv/iconv.c

PHP_ICONV_API php_iconv_err_t php_iconv_string(const char *in_p, size_t in_len, zend_string
**out, const char *out_charset, const char *in_charset)
{
    ...

    in_left= in_len;
    out_left = in_len + 32; /* Avoid realloc() most cases */ // [1]
    out_size = 0;
    bsz = out_left;
    out_buf = zend_string_alloc(bsz, 0);
    out_p = ZSTR_VAL(out_buf);

    while (in_left > 0) {
        result = iconv(cd, (ICONV_CONST char **) &in_p, &in_left, (char **) &out_p, &out_left);
        out_size = bsz - out_left;
        if (result == (size_t)(-1)) {
            if (ignore_ilseq && errno == EILSEQ) {
                if (in_left <= 1) {
                    result = 0;
                } else {
                    errno = 0;
                    in_p++;
                    in_left--;
                    continue;
                }
            }

            if (errno == E2BIG && in_left > 0) { // [2]
                /* converted string is longer than out buffer */
                bsz += in_len;

                out_buf = zend_string_extend(out_buf, bsz, 0); // [3]
                out_p = ZSTR_VAL(out_buf);
                out_p += out_size;
                out_left = bsz - out_size;
                continue;
            }
        }
        break;
    }

    ...
}
```

Therefore, the output buffer is 32 bytes bigger than the input buffer, making the overflow triggerable easily. A poc is available here. It boils down to:

```
$input =
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" .
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" .
    "AAA劄劄\n劄劄\n劄劄\n劄\n劄\n劄\n劄";
$output = iconv("UTF-8", "ISO-2022-CN-EXT", $input);
```

Now that we know that the bug is triggerable using `iconv()`, we can look for targets. Again, the preconditions are as follow: we need to control the output charset, and at least part of the input buffer. Now, what type of software may satisfy these? My initial idea was to look at email clients, because emails imply encoding. But before we get into the details of the attack, let's get through a little bit more of theory.

Remote, binary exploits on PHP: theory

Involontary mitigation

While PHP has been written off by so-many as insecure, remote binary exploits targeting PHP are – to say the least – not well documented. How does an attacker, armed with a buffer overflow such as the one we have, compromise the PHP engine and get remote code execution? Before we get our hands dirty, I'll show you why it is not as easy as it seems.

You should have a decent idea of how the PHP heap works, from part 1. Its design is straightforward, and it is not protected in any way (yet?). However, its main attack mitigation, to me, comes from a simple (and probably not security related) design choice: **a heap only handles one request**. When you send an HTTP request to PHP, it will create a new heap, parse and allocate your parameters (GET, POST, *etc.*), compile and run the requested script, return the HTTP response, and then, when everything is said and done, *delete* the heap.

Think about your standard remote exploitation. We can roughly divide it in three steps: *setup*, *trigger*, and *usage*. Take a *use-after-free*, for instance: you may interact with the server, first, to shape the heap of the target, maybe spray a few structures, or arrange some free lists. That's *setup*. Then, you'd send a second request, *triggering* the bug, and making the application free some chunk, while leaving a pointer dangling. Right after, you'd issue an additional straw, replacing the missing chunk by something more of your liking. The fourth request would be the nail on the coffin: using your type-confused structure to your advantage, and starting up some ROP chain, thus *using* your bug.

With PHP, we need **all of these steps** to happen in the course of **one request-response exchange**. And after we send the HTTP parameters, our hands are tied: there are no more ways to interact with the engine, and we expect the *setup*, *trigger*, and *usage* to happen, on their own, before we get the HTTP response, and the heap gets destroyed.

To circumvent this problem, people often targeted functions that let you, by design, interact with PHP while the request is running. That's why I targeted code related to databases in PHP, a few years back: when PHP sends SQL queries and receives results, it gives us a way to *exchange data*, force PHP into making allocations, deallocations, *etc.* More generally, vulnerabilities in functions such as `unserialize()` constitute an ideal target, because it lets you trigger a bug, and then create arbitrary objects, strings, arrays...

In part 1, when attacking `php://filter`, we had kind of a similar situation, where we could perform operations on the heap by using carefully picked filters and buckets, and provoke the memory corruption at an arbitrary moment by converting to *ISO-2022-CN-EXT*. But in this new case, with direct calls to `iconv()`, we are left in a very unsettling spot: the function will only let us *trigger* the bug. To perform the *setup*, and then *use* it, we'll need another way.

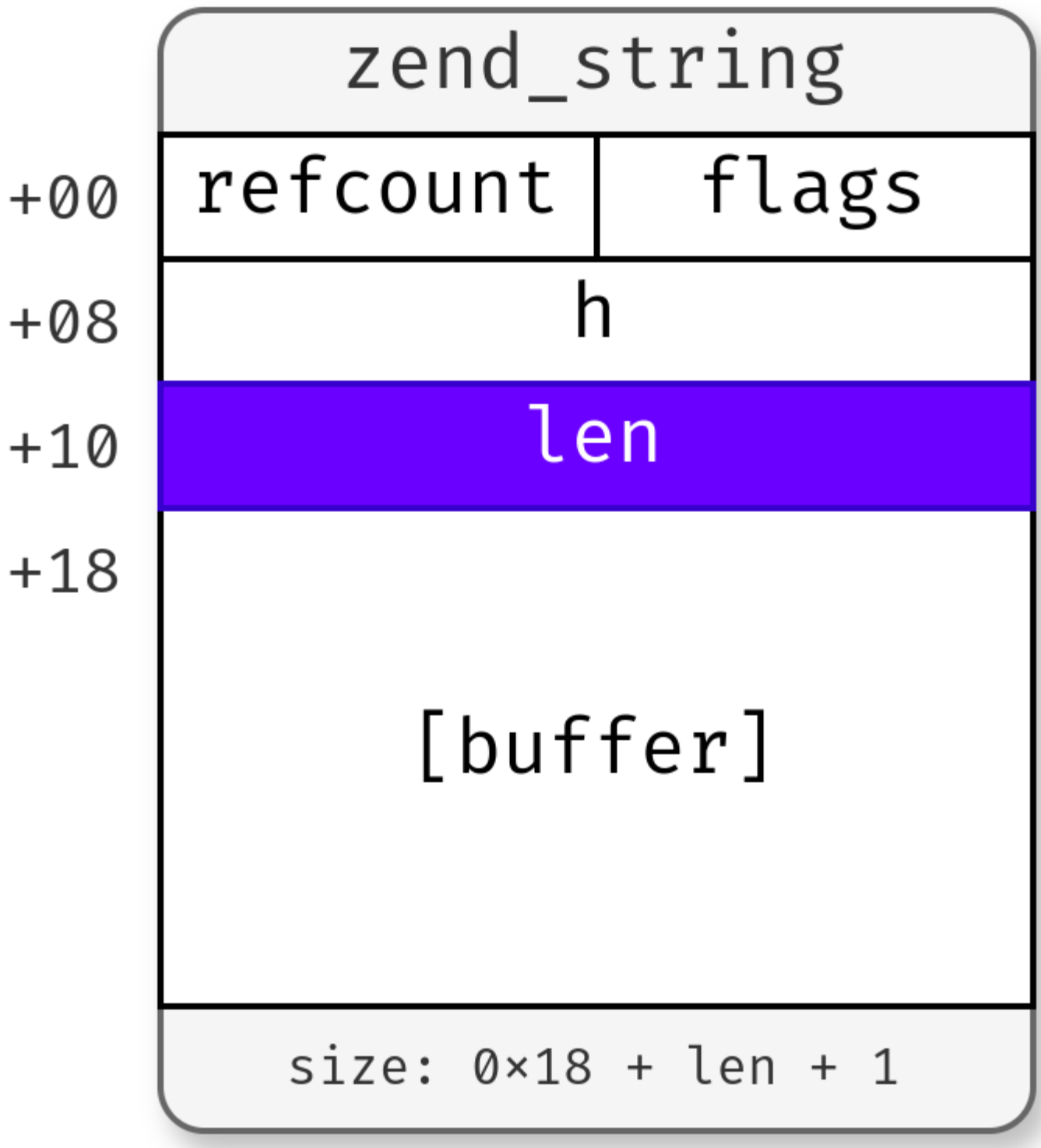
Encouraging architecture

We have things going for us, however. The environment is attack-friendly: to handle HTTP requests, PHP-FPM and mod PHP have a main/worker architecture, where a root process controls a few, less privileged, workers. Whenever a worker dies, the main process restarts it by forking. A two-fold advantage. First, if we somehow crash a worker, it gets respawned. No risk of DOSing a server. Second, the memory layout (ASLR, PIE) is the same accross workers: if we leak addresses from one, we are garantied that their MSBs will be identical to the others.

Theorical exploitation goals

The standard way of doing remote PHP exploits is thus to use the vulnerability twice: once to get a leak, and then to execute code. To make this happen, we can corrupt, in turn, two structures: `zend_string`s and `zend_array`s.

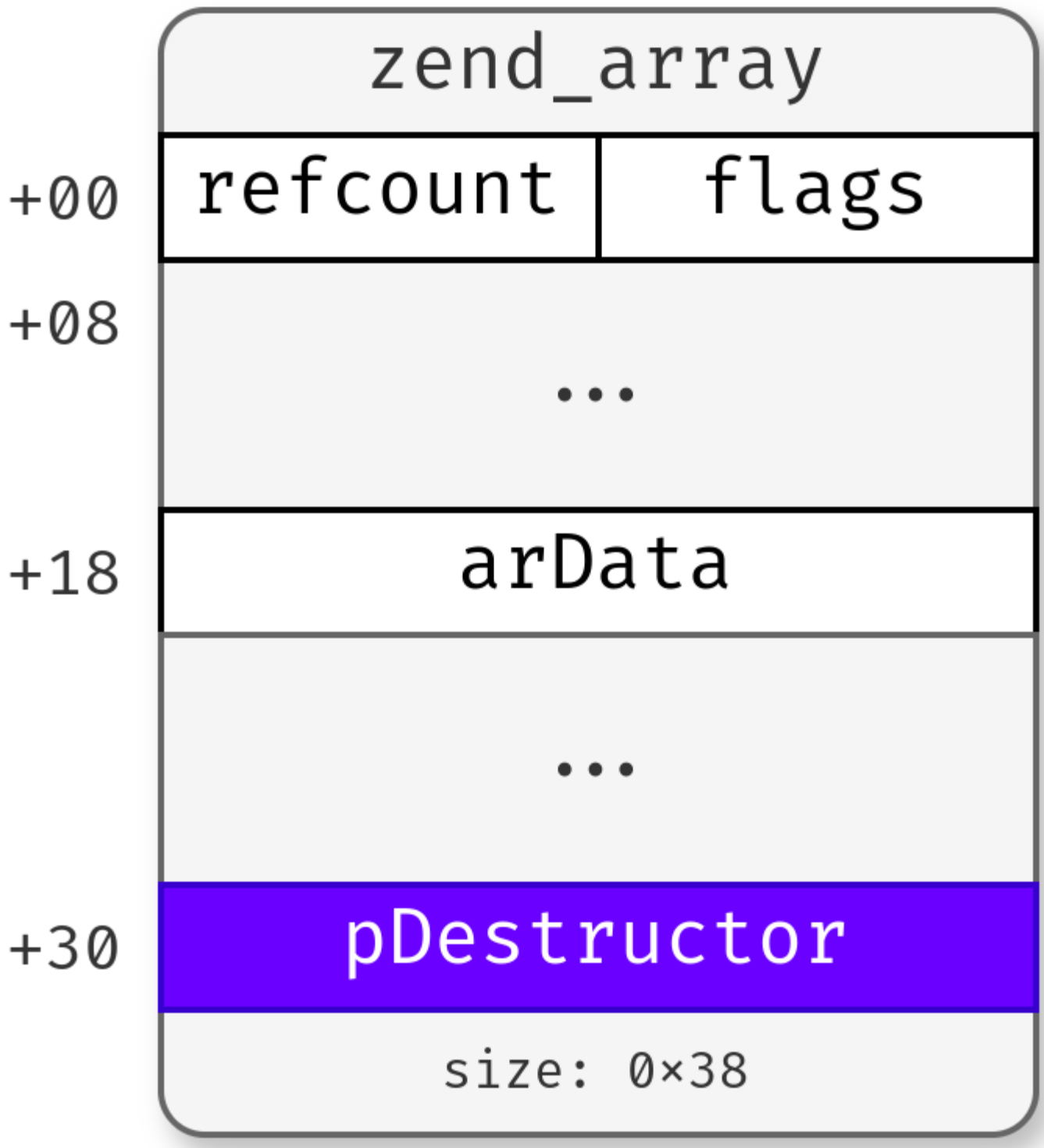
The `zend_string` structure represents a PHP string. It consists of a few fields, followed by a buffer.



The `zend_string` structure

In PHP, strings are not a NULL-terminated collection of bytes; their `len` field defines their size. Thus, to display a string `s`, PHP displays `len` bytes starting from `s+18h`. If we manage to artificially increase the value of this field, we can leak memory.

With leaked memory, things get easier. We can easily locate ourselves in the heap, and get pointers to the main binary. The next step, executing code, can be done by overwriting the last field of the `zend_array` structure, that represents a PHP array:



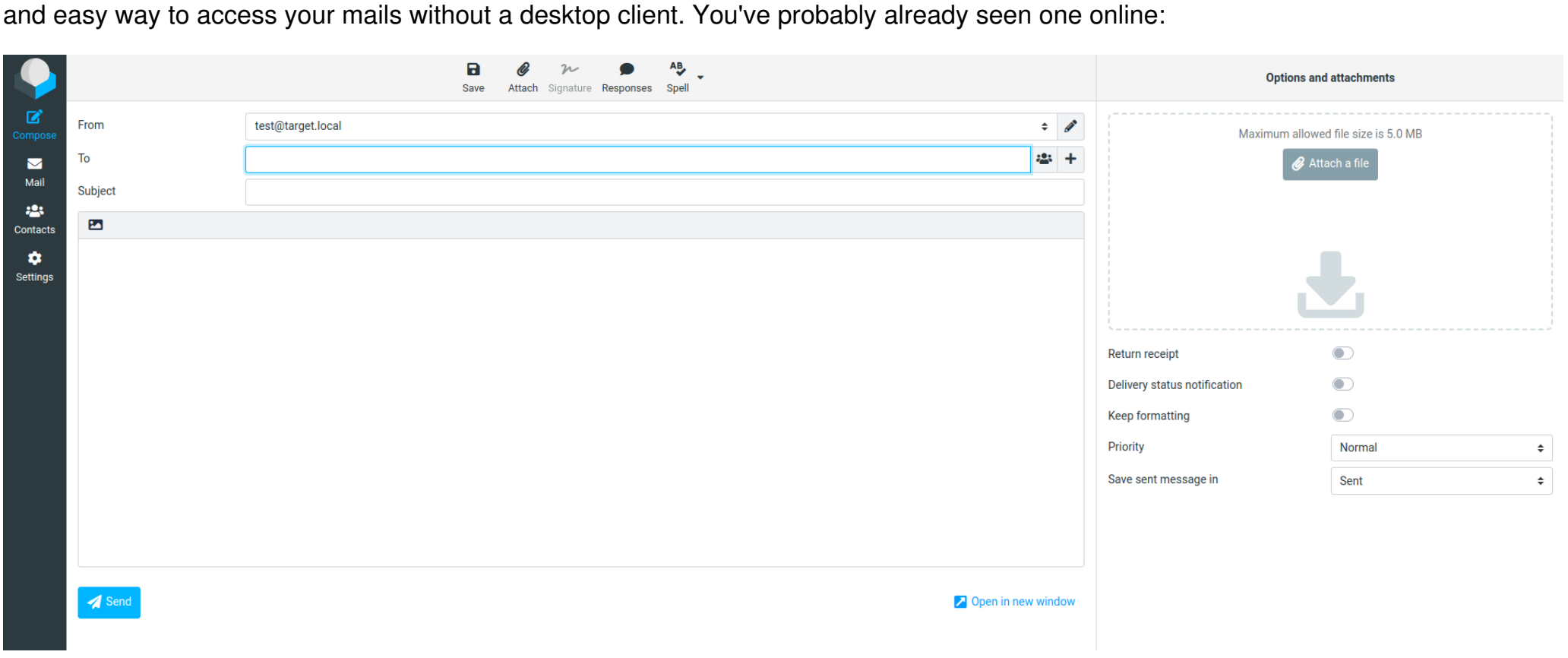
The `zend_array` structure

`pDestructor` is a pointer to the function responsible for deleting elements from an array. It generally points to `zval_ptr_dtor`, the variable destruction function of PHP. Changing its value allows us to get RIP: when the array gets deleted, its elements get deleted as well, and therefore `pDestructor` gets called.

But enough theory for now.

Attacking Roundcube

Roundcube is probably the most popular PHP webmail. It is often used by mail providers, web hosters or private companies as a quick and easy way to access your mails without a desktop client. You've probably already seen one online:



Roundcube interface

And sadly, it fits every of our preconditions, and lets us achieve **remote code execution** as a standard user.

Reaching the bug

When sending an email using Roundcube, one can specify the recipients, carbon copy, and blind carbon copy using the `_to`, `_cc`, and `_bcc` fields. Since you have all probably already sent an email, I won't describe what they are; you know that they represent a collection of email addresses.

Now, in addition to these fields, the user can send a `_charset` HTTP parameter [1]. In this case, Roundcube will use `iconv()` to converted the aforementioned parameters to the charset before they are processed. The code looks like this (*greatly simplified*):

```
# /program/include/rcmail_sendmail.php
class rcmail_sendmail
{
    public function headers_input()
    {
        ...

        // set default charset
        if (empty($this->options['charset'])) { // [1]
            $charset = rcube_utils::get_input_string('_charset', rcube_utils::INPUT_POST) ?:
$this->rcmail->output->get_charset();
            $this->options['charset'] = $charset;
        }

        $charset = $this->options['charset'];

        ...

        $mailto  = $this->email_input_format(rcube_utils::get_input_string('_to',
rcube_utils::INPUT_POST, true, $charset), true);
        $mailcc  = $this->email_input_format(rcube_utils::get_input_string('_cc',
rcube_utils::INPUT_POST, true, $charset), true);
        $mailbcc = $this->email_input_format(rcube_utils::get_input_string('_bcc',
rcube_utils::INPUT_POST, true, $charset), true);

        ...

        if (!empty($this->invalid_email)) { // [2]
            return display_error('emailformaterror', 'error', ['email' => $this-
>invalid_email]);
        }
    }
}
```

While `rcube_utils::get_input_string()` is a simple wrapper to get HTTP parameters, and convert them to `$charset`, `email_input_format()` is a complex function that verifies that the list of emails is valid. In fact, if one of the provided emails is not, it will be copied in `$this->invalid_email`, and displayed in an error message such as: `Invalid email address: <email>` [2].

We can trigger the vulnerability using either `_to`, `_cc`, or `_bcc`.

Getting a leak

To get a leak, we need to overwrite the `len` field of a `zend_string`, before it gets displayed. We'll call this string the *target string*. In our case, we have a very simple candidate: if one of the emails we send is invalid, Roundcube will display an error message containing said email. We can send such an email in `_to`, and use it as the *target string*!

Now, our primitive is far from being a *write-what-where*, or even an arbitrary overflow. It writes, at most, 3 bytes out-of-bounds. If we were to overflow directly into a `zend_string`, the only thing we could overwrite would be its `refcount`. Clearly, we cannot directly use the bug to do what we want. Instead, we can leverage a 1-byte overflow into a free chunk pointer, similarly to the technique used in part 1, in order to displace it, and make a chunk overlap with the *target string*, allowing us to overwrite its header.

While this all works in theory, we are facing the *1-heap-per-request* mitigation. How can we shape the heap before the bug triggers? And once we have altered the LSB of a free list pointer, how do we make PHP allocate more chunks, to overwrite the header of the *target string*?

Heap shaping 101

Using GET, POST, and cookies, one can force PHP to allocate strings of arbitrary length. Every time you send a key-value pair such as `key=value`, PHP allocates a `zend_string` to store the key, and two to store the value. In addition, you can make PHP deallocate chunks by sending a new value for the key: `key=value&key=other-value` would cause PHP to allocate `key`, then `value` twice, then `other-value` twice, and finally free the two `value` strings. To fill a page with chunks of size *0x400*, for instance, and leave the third

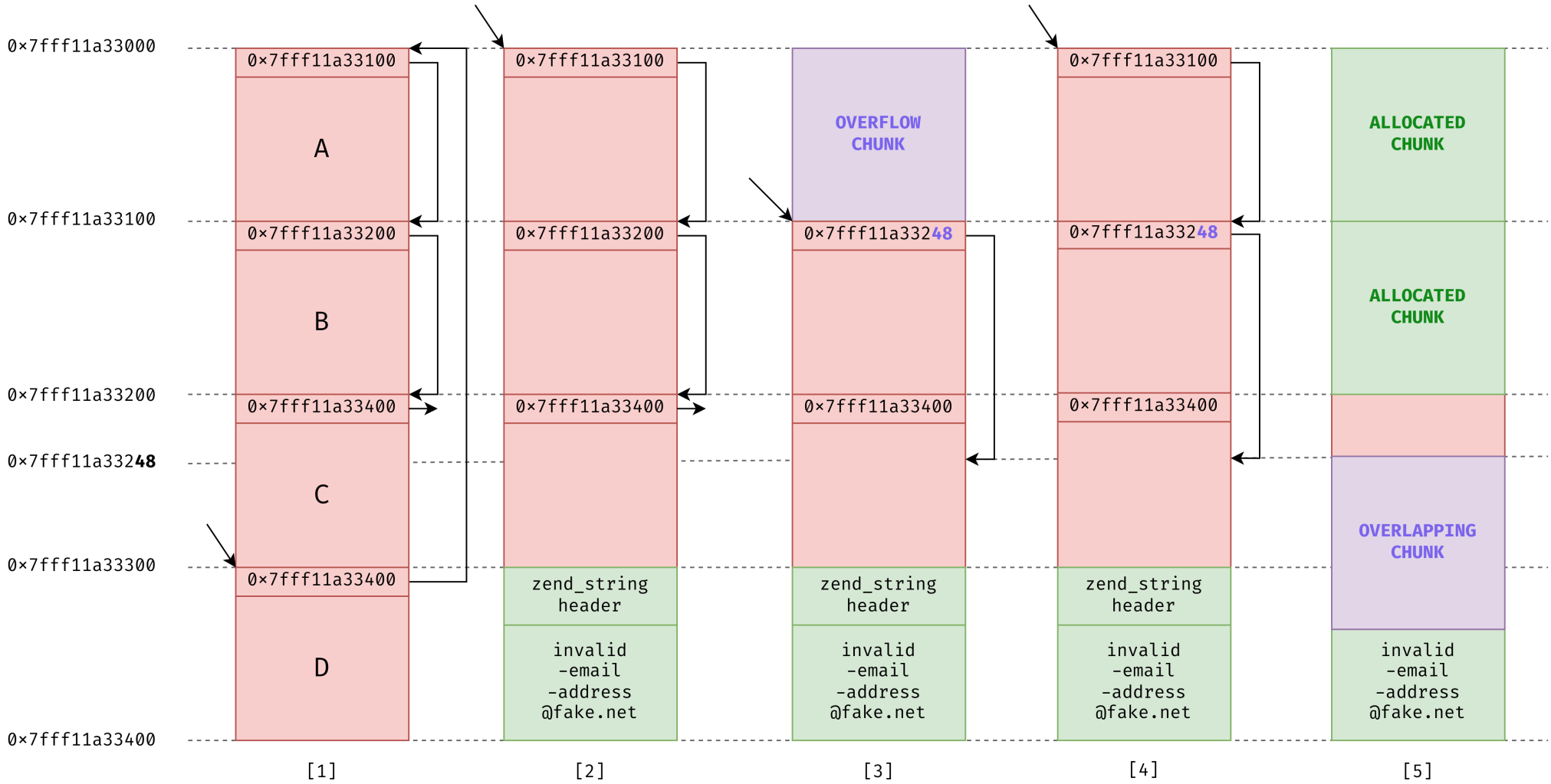
one freed, you could use the following combination (a `zend_string` of size *N* is stored over *N+0x19* bytes):

```
# Imagining that we have a page of four unallocated 0x400 chunks: C1 C2 C3 C4
# With a "standard" free list of C1→C2→C3→C4
 a=AA...AAAAA (0x3e7 times)  # Allocates two 0x400 chunks in C1 and C2
&b=BB...BBBBB (0x3e7 times)  # Allocates two more in C3 and C4
&b=                          # Frees C3, then C4
&CC...CCCCC (0x3e7 times)=   # Allocates C4
```

Using HTTP parameters, we can therefore shape the heap into our liking, right after it is created. While this is very nice, it is not perfect: modern PHP applications will, when compiled and run, perform thousands of heap operations, thus completely garbling our work. Imagine the whole process: parsing the code, comments, strings, objects, compiling the code to PHP VM instructions, and then running them, manipulating data, entering and exiting functions, *etc.* The best plan, when you can, is to try and attack on chunk sizes that are less used by the application, so that the program does not mess up your setup too much.

Code gadgets

Now that we can influence the looks of the heap, we can build a five step process to get our leak:



Exploitation steps on chunks of size 0x100

We first shape the heap so that 4 *0x100* chunks A, B, C, D are contiguous and free, and the free list is: D→A→B→C (fig 1).

After making PHP write the `zend_string` that stores the invalid email address at (fictional) address `0x7fff11a33300` (D) (fig 2), we overflow from the chunk at address `0x7fff11a33000` (A), overwriting the LSB of the pointer to `0x7fff11a33200` (C), which becomes `0x7fff11a33248` (fig 3). After the bug has triggered, we have A→B→C+48h (fig 4). Then, with 3 more allocations, we allocate a chunk that overlaps with the *target string* (fig 5), allowing us to overwrite its `zend_string` header and, more precisely, its `len` field.

We know how to perform the setup (steps 1, 2) and trigger the bug (steps 3, 4). One step is missing, though: how do we allocate the chunks after corrupting the freelist? At this point in the execution, the script is on its own. The only thing that can make PHP allocate anything is the script itself. Therefore, to perform our required allocations, we need to look at ways to make the PHP application do it for us.

Let's check the target function again:

```
# /program/include/rcmail_sendmail.php
class rcmail_sendmail
{
    public function headers_input()
    {
        ...

        $mailto  = $this->email_input_format(rcube_utils::get_input_string('_to',
rcube_utils::INPUT_POST, true, $charset), true);
        $mailcc  = $this->email_input_format(rcube_utils::get_input_string('_cc',
rcube_utils::INPUT_POST, true, $charset), true); // [1]
```

```
        $mailbcc = $this->email_input_format(rcube_utils::get_input_string('_bcc',
rcube_utils::INPUT_POST, true, $charset), true);

        ...

        if (!empty($this->invalid_email)) {
            return display_error('emailformaterror', 'error', ['email' => $this-
>invalid_email]); // [2]
        }
    }
}
```

Supposing that we use `_to` to set an invalid email, and then `_cc` to trigger the bug, we can use anything that happens in between [1] and [2] to allocate our chunks. Let's have a look at `email_input_format()` (again, greatly simplified):

```
# /program/include/rcmail_sendmail.php
class rcmail_sendmail
{
    /**
     * Parse and cleanup email address input (and count addresses)
     *
     * @param string $mailto Address input
     * @param bool   $count  Do count recipients (count saved in $this-
>parse_data['RECIPIENT_COUNT'])
     * @param bool   $check  Validate addresses (errors saved in $this-
>parse_data['INVALID_EMAIL'])
     *
     * @return string Canonical recipients string (comma separated)
     */
    public function email_input_format($mailto, $count = false, $check = true)
    {
        ...

        $emails  = rcube_utils::explode_quoted_string("[,;]", $mailto); // [1]

        foreach($emails as $email) {
            if(!is_valid_email($email)) {
                $this->invalid_email = $email;
                return "";
            }
        }

        return implode(", ", $emails);
    }
```

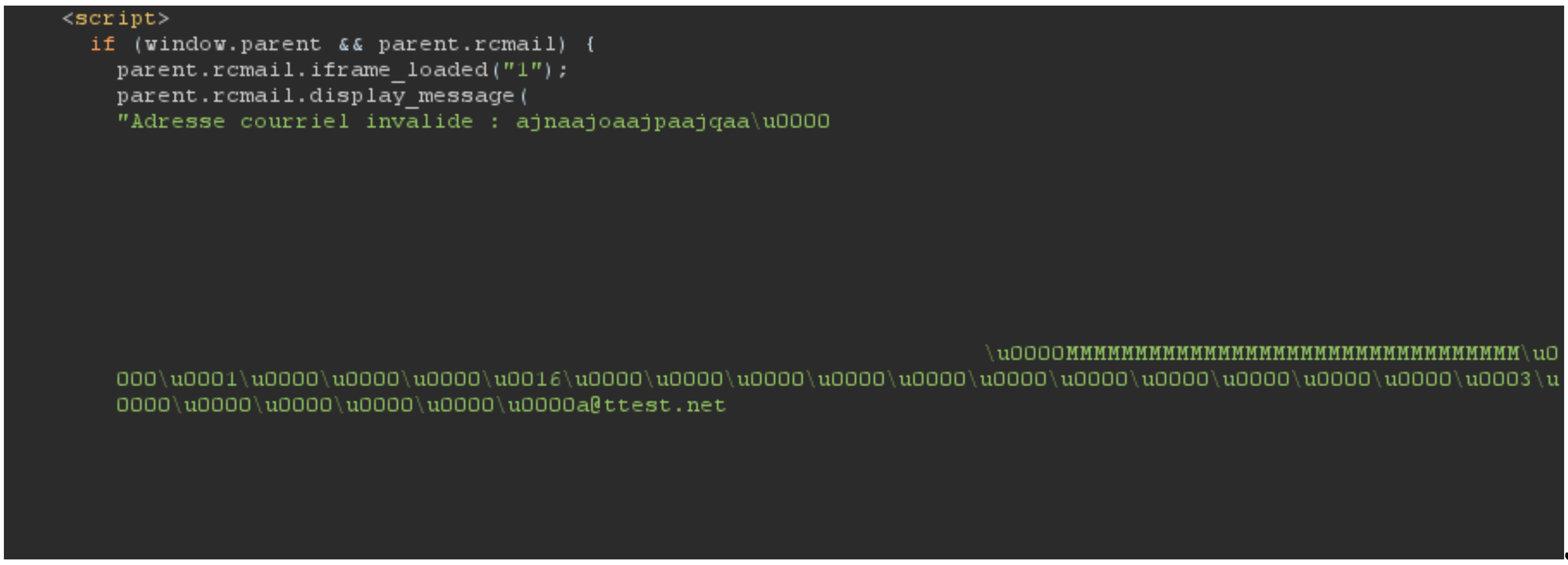
The method splits `$mailto`, a list of emails, into an array [1]. A perfect way to force PHP into allocating chunks!

We now have a complete strategy:

- Shape heap using HTTP parameters (step 1)
- Use `_to` to send an invalid email, setting `$this->invalid_email` (step 2)
- Use `_cc` to trigger the bug, modifying the free list (step 3, 4)
- Use `_bcc` to force PHP to allocate strings, overwriting the length of `invalid_email` (step 5)

When the error message gets displayed, memory gets leaked.

After building an exploit, I managed to make Roundcube display an error with my modified email (`Adresse courriel invalide` is french for `Invalid email address`), but it was... lackluster.



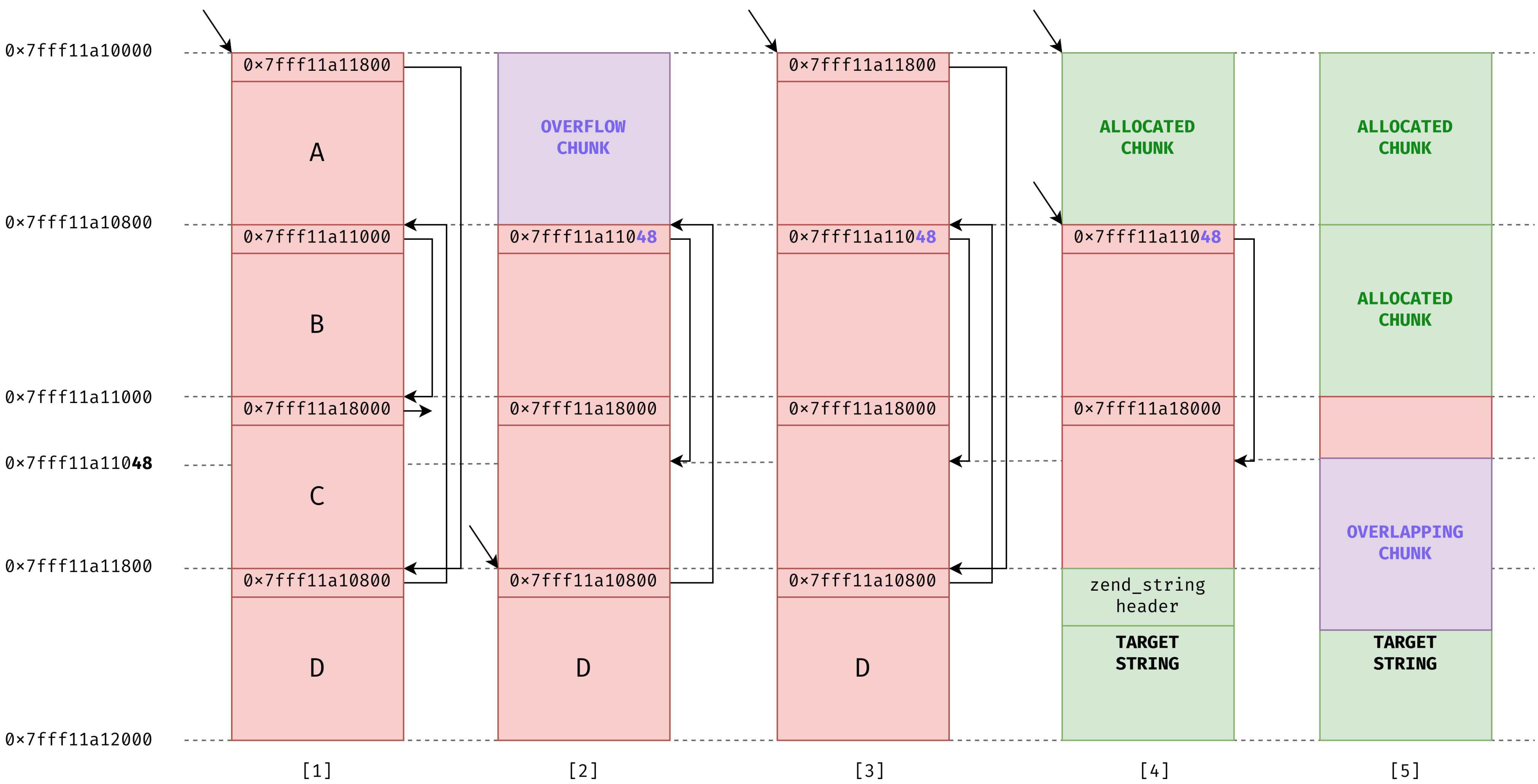
JSON-encoded error message

The error message only contained spaces, unicode-escaped null bytes, and ASCII characters. What happened? In fact, Roundcube displays error messages as JSON. To encode them, it uses the `json_encode()` API, with the `JSON_INVALID_UTF8_IGNORE` flag. As a consequence, characters that are not valid UTF-8 get thrown away. Since most data in memory is not, our leak does not contain anything of interest.

Revised leak strategy

Our *target string* was not the most fruitful one, but the idea is still correct. Instead, we need to find a variable that gets displayed « as-is » (or, with less modifications).

Like most web applications, Roundcube formats its output very late in the execution, way **after we trigger the bug**. Evidently, this is where most candidates *target strings* get allocated. We thus need to change our exploit algorithm. We'll still use 4 chunks, and will still displace `C` to make it overlap with `D`, which will contain the *target string*. But this time, we trigger the overflow **before** the *target string* is allocated.



Better exploitation steps (on chunks of size 0x800)

The previous free list was `D→A→B→C`. Now, we need `A→D→B→C` (fig 1). We can then overflow from `A` to `B` (fig 2), and get `A→D→B→C'` (fig 3), with `C'` overlapping with `D`.

We cannot allocate chunks easily like in the previous section, because the providencial `explode_quoted_string()` now happens **before** the allocation of the *target string*. Furthermore, we cannot just blindly allocate 3 chunks: since the free list is now `A→D→B→C'`, we need to make PHP allocate one chunk, then the *target string* (fig 4), and then two others (fig 5).

Let's perform, one by one, the steps of our strategy. First, we will use `_to` to perform the overflow, and thus end up in step 2. To force an allocation in `A`, we'll use `_bcc`, in order to make `email_input_format()` return a string which fits in a *0x800* chunk. The free list becomes `D→B→C'`.

We now need to tackle the hard part: finding a *target string*. I went through the whole stack trace responsible for displaying error messages, and finally ended up in `rcmail_output_html::get_js_commands()`:

```
# program/include/rcmail_output_html.php
class rcmail_output_html extends rcmail_output
{
    protected function get_js_commands(&$framed = null)
    {
        $out             = '';
        $parent_commands = 0;
        $parent_prefix   = '';
        $top_commands    = [];

        // these should be always on top,
        // e.g. hide_message() below depends on env.framed
        if (!$this->framed && !empty($this->js_env)) {
            $top_commands[] = ['set_env', $this->js_env];
        }
        if (!empty($this->js_labels)) {
            $top_commands[] = ['add_label', $this->js_labels];
        }

        // unlock interface after iframe load
        $unlock = isset($_REQUEST['_unlock']) ? preg_replace('/[^a-z0-9]/i', '',
$_REQUEST['_unlock']) : 0;
        if ($this->framed) {
            $top_commands[] = ['iframe_loaded', $unlock];
        }
        else if ($unlock) {
            $top_commands[] = ['hide_message', $unlock];
        }

        $commands = array_merge($top_commands, $this->js_commands);

        foreach ($commands as $i => $args) {
            $method = array_shift($args);
            $parent = $this->framed || preg_match('/^parent\./', $method);

            foreach ($args as $i => $arg) {
                $args[$i] = self::json_serialize($arg, $this->devel_mode);
            }

            if ($parent) {
                $parent_commands++;
                $method        = preg_replace('/^parent\./', '', $method);
                $parent_prefix = 'if (window.parent && parent.' . self::JS_OBJECT_NAME . ')
parent.';
                $method        = $parent_prefix . self::JS_OBJECT_NAME . '.' . $method;
            }
            else {
                $method = self::JS_OBJECT_NAME . '.' . $method;
            }

            $out .= sprintf("%s(%s);\n", $method, implode(',', $args));
        }

        $framed = $parent_prefix && $parent_commands == count($commands);

        // make the output more compact if all commands go to parent window
        if ($framed) {
            $out = "if (window.parent && parent." . self::JS_OBJECT_NAME . ") {\n"
                . str_replace($parent_prefix, "\tparent.", $out)
                . "}\n";
        }

        return $out;
    }
}
```

This method generates javascript code that gets displayed, **raw**, in the HTTP response.

It is pretty complex, which is, in a way, a good thing: since the return value, `$out`, is bound to be displayed at some point, each variable that gets concatenated to it is a potential *target string*. In addition, each line of code here performs one or several allocation, or deallocation, or reallocation... a way to handle step 5. Each line of code is therefore a *gadget* that may or may not help us in our quest to

overwrite the string header.

Sadly, here, no gadget as simple as an `explode_quoted_string()`: we'll need to be smarter.

Finding the target

Let's simplify the code by removing conditions that we do not enter:

```
01: protected function get_js_commands()
02: {
03:     $out             = '';
04:     $top_commands    = [];
05:
06:     // unlock interface after iframe load
07:     $unlock = isset($_REQUEST['_unlock']) ? preg_replace('/[^a-z0-9]/i', '',
$_REQUEST['_unlock']) : 0;
08:     $top_commands[] = ['iframe_loaded', $unlock];
09:
10:     $commands = array_merge($top_commands, $this->js_commands);
11:
12:     foreach ($commands as $i => $args) {
13:         $method = array_shift($args);
14:
15:         foreach ($args as $i => $arg) {
16:             $args[$i] = self::json_serialize($arg, $this->devel_mode); // [1]
17:         }
18:
19:         $method = 'if (window.parent && parent.rcmail) parent.rcmail.' . $method;
20:         $out .= sprintf("%s(%s);\n", $method, implode(',', $args)); // [2]
21:     }
22:
23:     $out = "if (window.parent && parent.rcmail) {\n"
24:          . str_replace('if (window.parent && parent.rcmail) parent.rcmail.', "\tparent.",
$out)
25:          . "}\n";
26:
27:     return $out;
28: }
```

In our case, when the code reaches `get_js_commands()`, `$this->js_commands` is an array containing a single element, a 2-item array: `["display_message", "Addresse courriel invalide: <email-we-sent>"]`. The `$commands` array thus consists of 2 elements:

```
[
    ["hide_message", "<unlock-value>"],
    ["display_message", "Addresse courriel invalide: <invalid-email>"],
]
```

Each line is then used to iteratively build part of some javascript code in `$out`, before returning the variable. In each the iterations of the loop at lines 12 to 21, we control `$args[0]`, which gets JSON-serialized, and then formatted using `sprintf()`. Let's take each of the two iterations one by one. Before we enter the `foreach()`, the free list is `D→B→C'`: the next allocation has to be our *target string*.

If we give a size of *0x6a1* to the HTTP parameter `_unlock`, the first iteration of the loop will end with `$out` having a size of more than `0x700` bytes. It therefore gets allocated in chunk `D`. We then need 2 more allocations to overwrite `$out`'s length, before its size changes. We need make them happen in the last iteration of the loop.

To pull this off, we set `invalid_email` to *0x63c* ASCII bytes, followed by *0x37* null bytes. When the error message gets JSON-serialized [2], `$args[0]` grows greatly in size because of the null bytes it contains: each null byte becomes its unicode-escaped representation, `\u0000`. As a result, the JSON-encoded `$args[0]` has a size of around *0x786* bytes. It thus gets allocated in `B`. The `sprintf()` call adds a few bytes to this, and results in a new chunk of size *0x800* being allocated, in `C'`. At this point, we have successfully overwritten `D`'s header: `$out`'s size has greatly increased, right before it gets modified again by being concatenated with the result of `sprintf()` [2]!

And finally, we get our much desired leak:

```
000000a0  62 65 27 3e  3c 73 63 72  69 70 74 3e  0a 69 66 20  |be'>|<scr|ipt>|·if |
000000b0  28 77 69 6e  64 6f 77 2e  70 61 72 65  6e 74 20 26  |(win|dow.|pare|nt &|
000000c0  26 20 70 61  72 65 6e 74  2e 72 63 6d  61 69 6c 29  |& pa|rent|.rcm|ail)|
000000d0  20 7b 0a 09  70 61 72 65  6e 74 2e 72  63 6d 61 69  | {··|pare|nt.r|cmai|
000000e0  6c 2e 69 66  72 61 6d 65  5f 6c 6f 61  64 65 64 28  |l.if|rame|_loa|ded(|
000000f0  22 55 55 55  55 55 55 55  55 55 55 55  55 55 55 55  |"UUU|UUUU|UUUU|UUUU|
00000100  55 55 55 55  55 55 55 55  55 55 55 55  55 55 55 55  |UUUU|UUUU|UUUU|UUUU|
*
00000790  55 55 22 29  3b 0a 00 4f  4f 4f 4f 4f  4f 4f 4f 4f  |UU")|;··O|OOOO|OOOO|

                                ...

000027a0  02 00 00 00  ff ff ff ff  91 3e 49 90  cb 7b 24 9f  |····|····|·>I·|·{$·|
000027b0  08 bc 27 29  1e 7f 00 00  02 00 00 00  00 00 00 00  |··')|····|····|····|
000027c0  02 00 00 00  ff ff ff ff  82 76 80 1d  dc 1e ad d1  |····|····|·v··|····|
000027d0  38 bc 27 29  1e 7f 00 00  02 00 00 00  00 00 00 00  |8·')|····|····|····|
000027e0  01 00 00 00  93 00 00 00  1a 48 30 60  b7 55 2a e4  |····|····|·H0`|·U*·|
000027f0  38 c5 26 29  1e 7f 00 00  02 00 00 00  00 00 00 00  |8·&)|····|····|····|
00002800  02 00 00 00  ff ff ff ff  3e d5 c3 06  ae 06 5a fb  |····|····|>···|··Z·|
00002810  68 bc 27 29  1e 7f 00 00  02 00 00 00  00 00 00 00  |h·')|····|····|····|
00002820  02 00 00 00  ff ff ff ff  18 38 0a 66  c0 69 bb a7  |····|····|·8·f|·i··|
00002830  a0 bc 27 29  1e 7f 00 00  00 bd 27 29  1e 7f 00 00  |··')|····|··')|····|
00002840  06 00 00 00  ff ff ff ff  20 a8 d2 5d  3a f0 4c d0  |····|····|  ··]|:·L·|
00002850  d0 bc 27 29  1e 7f 00 00  58 bd 27 29  1e 7f 00 00  |··')|····|X·')|····|
00002860  06 00 00 00  ff ff ff ff  5a b2 58 05  a0 ff 08 be  |····|····|Z·X·|····|
00002870  28 bd 27 29  1e 7f 00 00  02 00 00 00  00 00 00 00  |(·')|····|····|····|
00002880  02 00 00 00  ff ff ff ff  d1 ef 23 f4  33 8f d4 f0  |····|····|··#·|3···|
00002890  80 bd 27 29  1e 7f 00 00  00 60 08 36  1e 7f 00 00  |··')|····|·`·6|····|
000028a0  00 00 00 00  00 00 00 00  e8 06 00 00  00 00 00 00  |····|····|····|····|
000028b0  4d 4c 5b 31  5d 00 00 00  00 00 00 00  00 00 00 00  |ML[1|]···|····|····|
```

Successfully leaking memory

Note: `sprintf()` allocates a chunk of size 0x800 to store the result string, which forces us into attacking on this chunk size.

Two paths

By carefully setting up the heap, we can leak at the same time pointers to the PHP binary and pointers close to the position of our target string. As a result, ASLR and PIE become irrelevant. In addition, we know how to corrupt a free list already, and therefore we can **allocate one chunk** anywhere in the heap. But it is not *game over* yet. At this point, there are two ways one can go.

The first one is the logical continuation of our work: using our binary corruption to execute code. It generally involves dumping parts of the binary to find interesting offsets, and then getting a ROP chain started. The second one involves performing a *data-only* attack. Each has their advantages and inconvenients. While I demonstrated a binary exploit at OffensiveCon, I find the *data-only* attack more elegant, and I will therefore show it instead. It serves as a good way to dive even deeper in the PHP engine.

Data-only attacks

The advantage of data-only attacks over binary ones are that we do not rely on machine code. Instead, we use our low level bug to corrupt PHP variables and alter the execution of the script (a very simple example could be setting an hypothetical `$is_admin` flag to `true` to escalate our privileges).

While we can build reasonably complex structures, we can only reference heap addresses. As a result, not every variable can be overwritten: we can target simple types (`bool`, `int`, `string`) and `arrays`, but not `objects`, because the structure that represents them, `zend_object`, includes pointers to the main binary (which we do not know much about!).

An ideal target are session variables, stored in the `$_SESSION` array, for a few reasons. First, they linger after the exploitation (provided it does not crash). Second, they get saved at the very end of the execution of the script, leaving us "time" to modify them. And third, they are generally interesting in an attacker's point of view: who hasn't dreamed of being able to change their role to `superadmin`?

In Roundcube, no concept of roles, however. But browsing through the code, we can actually find better:

```
# program/lib/Roundcube/rcube_user.php
class rcube_user
{
    function get_prefs()
```

```
    {
        if ($_SESSION['preferences_time'] < time() - 5 * 60) {
            $saved_prefs = unserialize($_SESSION['preferences']); // <-----------
            $this->rc->session->remove('preferences');
            $this->rc->session->remove('preferences_time');
            $this->save_prefs($saved_prefs);
        }
        ...
    }
}
```

A call to the PHP deserialisation function, `unserialize()`! Being able to perform deserialisation generally means RCE on big frameworks, and Roundcube is no exception. Indeed, it uses *Guzzle*, a popular library to do HTTP requests. Using PHPGGC, we can generate a `guzzle/fw1` payload, and convert the deserialisation into an arbitrary file write.

Evidently, under normal usage, `$_SESSION['preferences']` cannot be modified by a user. We are not just any user, however: we can write in the heap! Therefore, we can just make two chunks overlap and overwrite the `zend_string` of this session variable!

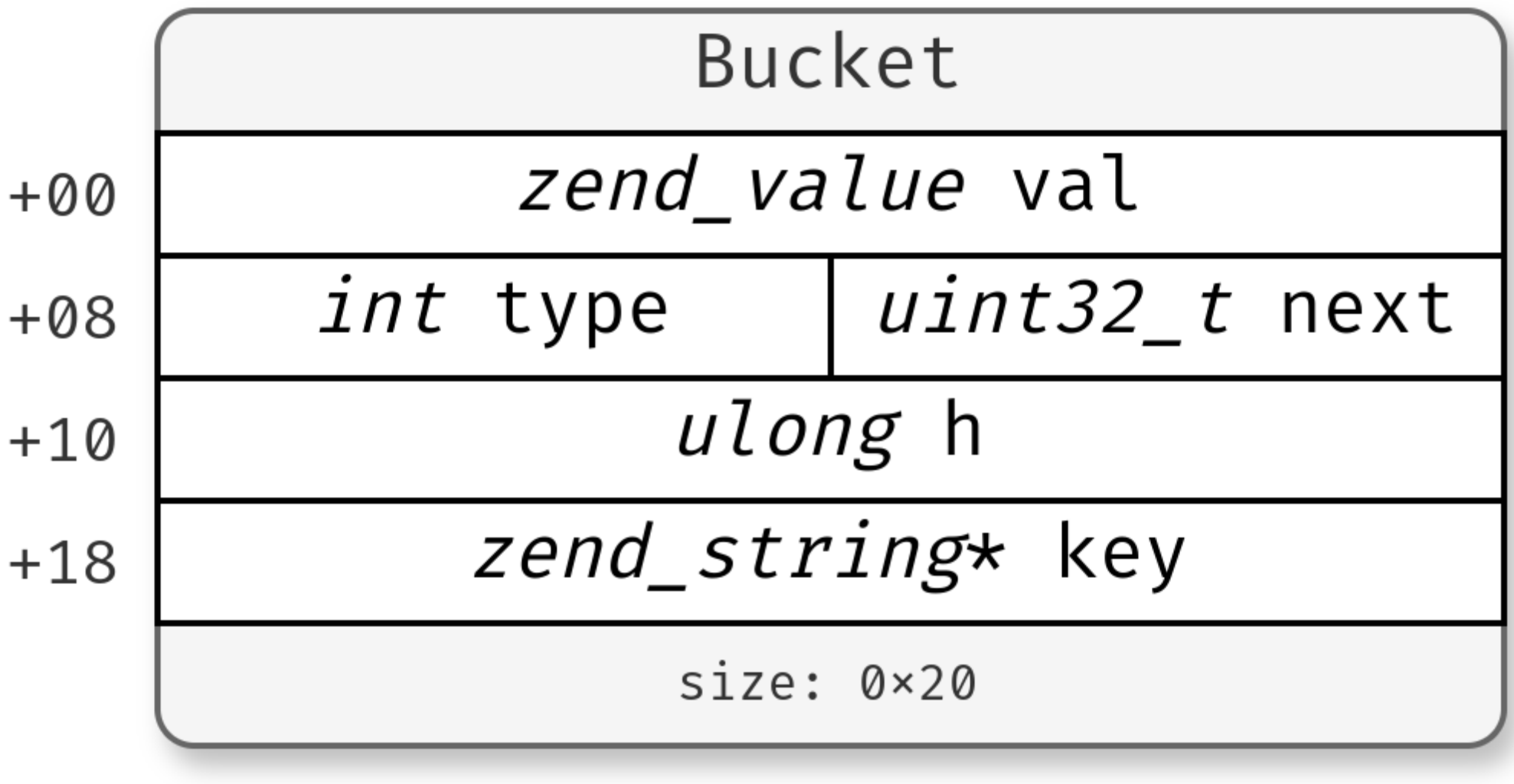
But again, a problem: under default configuration, `$_SESSION['preferences']` never gets set. There is nothing to overwrite! All is not lost, though: we can go deeper and use our arbitrary allocation to add an element to the `$_SESSION` array. How do we do this?

We'll need to dive in the implementation of PHP arrays.

PHP arrays

PHP arrays consist of key/value pairs, where the value can have any type, but the key can either be an integer or a string. We will only cover, here, string keys.

Each pair is held in a structure called a `Bucket`:

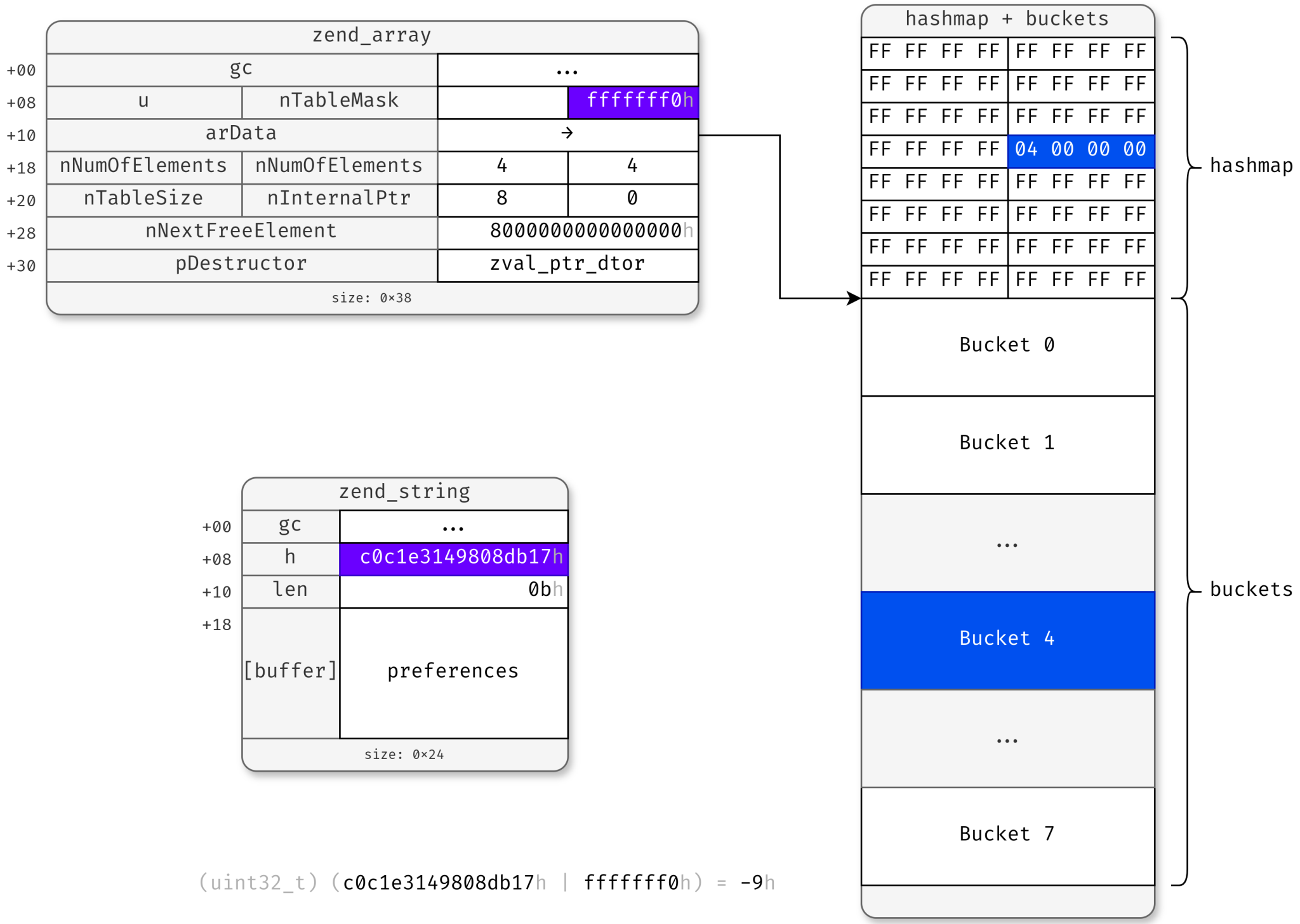


The `Bucket` structure

The first element, `val`, is either a simple value (*long*, *float*) or a pointer to a value (`zend_string`, `zend_object`, *etc.*). `type` defines the variable type. `next` indicates the index of the next `Bucket` in the list (more on this later). The `key` is stored in `Bucket.key`, and its DJBX33A hash is stored in `Bucket.h`.

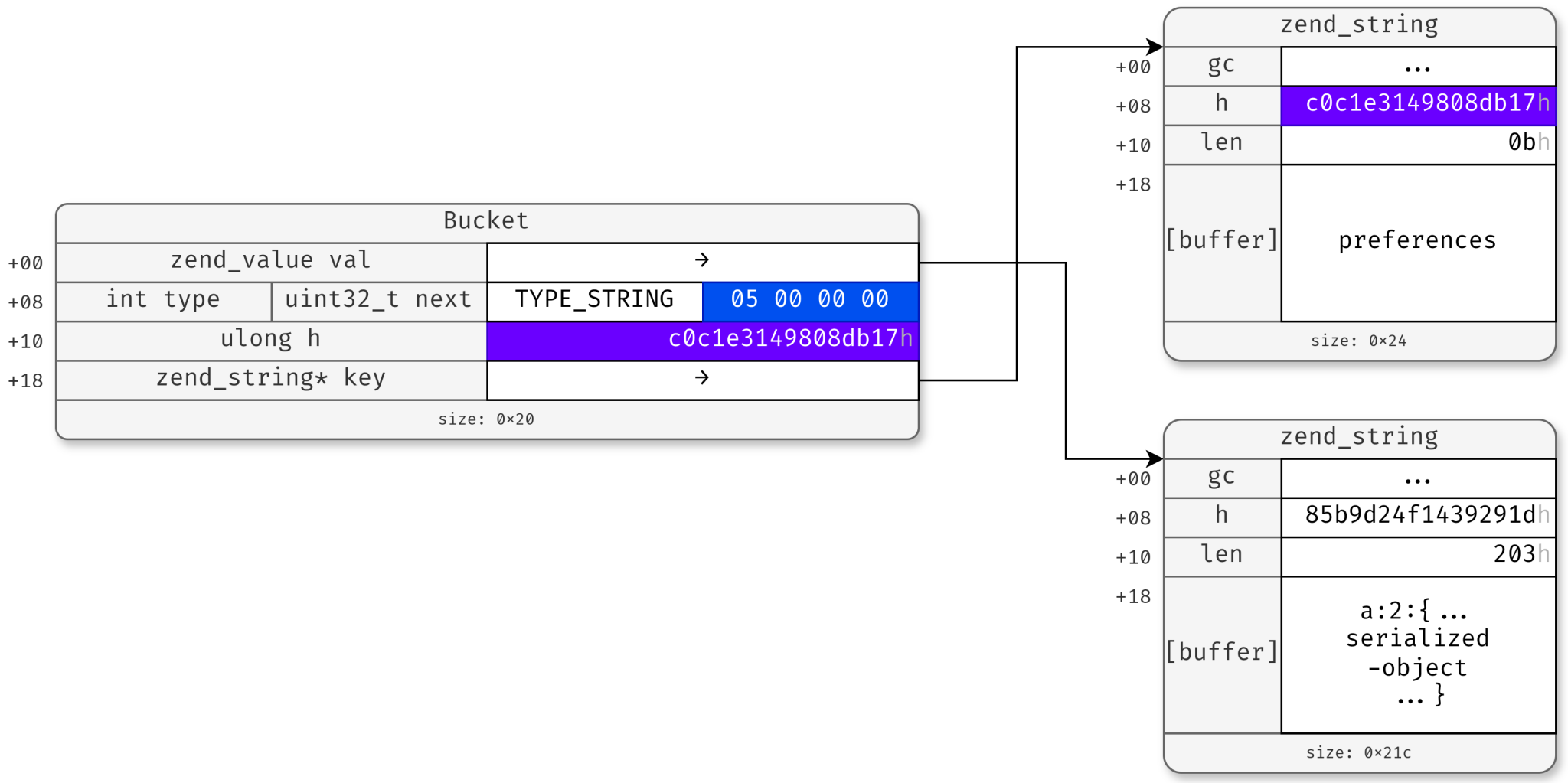
When a non-empty array is created, PHP allocates a `zend_array` structure, and another chunk that consists of a list of `uint32_t` values, the *hashmap*, followed by a list of *8* `Buckets`.

`zend_array.arData` points to this butterfly structure: on the bottom of this structure is the list of *buckets*, and on the top is the *hashmap*. The *hashmap* lets you convert `zend_string` hashes into an index in the list of *buckets*: when one tries to access the value for some key, PHP will *OR* the table mask (`zend_array.nTableMask`) with the hash of the key (`zend_string.h`), to get a negative `int32_t`, which it'll use as an index from the `(uint32_t[]) arData` pointer.



Accessing an hashmap

In the example above, we are looking for `preferences` in an array of *8* elements. The index is equal to `(int32_t) (0xffffffff0h | 0xc0c1e3149808db17) = 0xfffffff7 = -9`. By picking the 9th element starting from the end of the hashmap, we get 4 (in *blue*). As a result, PHP examines the fifth bucket.



A bucket and its key-value pair. The key is `preferences` and value is a serialized string.

To make sure that this is indeed the right bucket (two different strings may have the same hash, or hashes that map to the same index), PHP checks the hash of the bucket against the hash of the provided key. If they are equal (in our example, they are), it compares the size and value of the keys. If they are equal (in our example, they are, again), it is the correct bucket, and PHP returns the value. In the case where the expected key is not the same as the one of the bucket, PHP goes to the bucket whose index is indicated by the `next`

value (5, in *blue*, in our example), and keeps going until it finds it. `FF FF FF FF` is a special value that means that there is no bucket.

Overwriting the session array

Overwriting the hashmap and one bucket is therefore enough to add a key-value pair to the array.

Now, remember, our primitive lets us alter a free list pointer, and thus allocate a `zend_string` (or anything, but `zend_string`s are the most useful) anywhere in the heap, *i.e.* we can make PHP allocate an arbitrary chunk.

The session array consists of 32 elements, and as such its *hashmap+bucket* chunk has size *0x500* (*0x100* for hashmap, *0x400* for buckets). We want to make our fake heap pointer point right above.

It is rather simple, but we have a few more precautions to take. First, we cannot just point anywhere: when PHP allocates our arbitrary chunk, it'll think that this is really a free chunk (stupid PHP). Therefore, it'll expect its 8 first bytes to be a pointer to the next one. Supposing that it is not a valid pointer, if PHP makes one more allocation of the same size, we crash. In addition, when our fake chunk gets freed, it might get allocated again, and contain data that we do not control. We need to shield it from being reallocated; otherwise, it might completely break our work.

To handle the first problem, we can create, using HTTP parameters, a net of *0x500* chunks filled with null bytes and separated with a hole, hoping for the *hashmap+bucket* chunk to be allocated in between two of them. If we point into such a chunk, PHP will read a null pointer as the next free list element, and think that the free list was exhausted, saving us from a crash.

To tackle the second one, as soon as we allocate our arbitrary chunk, we also allocate lots of chunks of size *0x500*. When they all get freed, in order, the last chunks will precede our chunk in the freelist, protecting it from allocations.

And here we are: using our binary bug, we alter the contents of our session, and set `preferences` to an arbitary string. We then issue an HTTP request to the index, where `$_SESSION['preferences']` gets deserialized. Using the `guzzle/fw1` payload, we write a file to `public_html/shell.php`.

Demo

Here is a demo targeting Roundcube 1.6.6 under PHP 8.3.

The exploit is available here. As usual, it is commented, and reveals parts of the exploit I did not include in the blogpost.

Impact on the ecosystem

So, direct calls to `iconv()` are exploitable, and impactful. But is it the only PHP function affected by CVE-2024-2961? Not at all.

First, `iconv()` has a lot of sibling functions, such as `iconv_strrpos()`, `iconv_substr()`... which may be vulnerable (I have not checked). But there is an even scarier, and very unexpected, sink.

PHP has a very popular extension called `mbstring`. The extension, built in C, allows you to manipulate strings under various charsets,

and perform character set conversions. It is a dependency of many frameworks and CMSes.

But `mbstring` is not installed by default. What happens when it is not (you need superuser rights to do so), but you still want to use a library or framework that relies on it? Well, in such a case, you can use the PHP implementation of the library. The project, called `symfony/polyfill-mbstring`, has exactly the same API: both can be used interchangeably. And it is very popular, with **823+ million** installs.

But how does *polyfill-mbstring* do its job, converting from one charset to another, without using `mbstring`? Well, it uses... `iconv()`.

Therefore, you might be thinking you are using `mbstring`, and as such, are not vulnerable, but you are instead using the PHP-implemented, polyfill version, which uses `iconv()`.

With the avenement of `composer`, the PHP package manager, the line might even be easier to cross than expected. If you install two projects, one depending on `ext-mbstring` (the original C extension), and the other on `polyfill-mbstring` (the PHP equivalent), the installation will succeed whether the `mbstring` extension is installed or not.

And, sure enough, when you run the POC that I provided, but this time using `mb_convert_encoding()` instead of `iconv()`:

```
- $output = iconv("UTF-8", "ISO-2022-CN-EXT", $input);
+ $output = mb_convert_encoding($input, "ISO-2022-CN-EXT", "UTF-8");
```

You get a crash as well.

While I have stopped my analysis here (looking for targets is very time-consuming), I am hopeful that this is not the last we have seen of CVE-2024-2691 and PHP.

Conclusion

After exploiting PHP filters, we have now leveraged **CVE-2024-2961** through direct `iconv()` calls to compromise a famous webmail, Roundcube. This took us deeper in the internals of the PHP engine, and leads the way for potential new exploits, using the obvious sinks, and the less obvious ones.

Now that we have demonstrated impact through two means in PHP, we need to cover the last question: what happens if the file read primitive that you have is *blind*?

Stay tuned for part 3!

We're hiring!

Ambionics is an entity of Lexfo, and we're hiring! To learn more about job opportunities, do not hesitate to contact us at rh@lexfo.fr. *We're a french-speaking company, so we expect candidates to be fluent in our beautiful language.*

If you have any questions, you can hit me up on Twitter @cfreal_.