Iconv, set the charset to RCE: Exploiting the glibc to hack the PHP engine (part 1)

35-44 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.

Due to the amount of material, the impact and exploitation of the bug will be documented over a three part series. In this first part of the series, I'll describe how I encountered the bug, why suitable targets are rare, and finally dive into the PHP engine to demonstrate a new exploitation vector: **converting file read primitives into remote code execution in PHP applications**.

Note: part 2 is available here

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

Discovery: a story about filters

File read primitives in PHP

Let's first cover the basics. Say that, while performing an assessment, you find a file read primitive such as this one:

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

What can you do with it? Well, read files, obviously. You can read `/etc/passwd`, for instance. But PHP also lets you use other protocols, such as `http://` or `ftp://`. As a result, you can ask PHP to go get the front page of google for you, using `http://google.com`; or download a file from a FTP server, using `ftp://user:passwd@ftp.target.com/file.bin`. But that's not all; PHP also implements custom protocols, such as `phar://`.

`phar://` lets you read inside a PHAR archive. *PH*AR stands for *PHP* archive, the same way that *J*AR stands for *Java* archive. It is a grouping of files such as:

- Source code
- Resources
- Serialized metadata

This protocol has been the downfall of PHP for many years, because when you use it to access a PHAR file, its metadata gets deserialized. The usual PHAR attack looks like this:

1. Upload a PHAR archive to target server (PHAR files are very polyglottish, so you can make them look like an image, a PDF, or anything, really)
2. Access the PHAR file using your file read primitive, with `phar:///path/to/file.phar/test`
3. An arbitrary payload is deserialized

Converting deserialisation into code execution can be done in many ways, but people generally rely on the go-to deserialisation tool on PHP, PHPGGC.

You cannot overstate the impact of PHAR attacks. From their creation in 2018, they have been pivotal to getting shells on PHP targets. But the party is coming to an end:

- Starting from PHP 8.0 (released in 2020), `phar://` does not deserialise the metadata anymore. (they didn't use the metadata anyways, so why deserialise it). This kills PHAR attacks entirely.
- Big applications (such as Drupal or Magento) have been disabling the `phar://` protocol
- Deserialisation is going to become harder to exploit as time goes by: libraries are patching their deserialisation chains, and typing is making a comeback, drastically reducing deserialisation paths.

But `phar://` was not the only protocol useful protocol for attackers; another one yielded great results as well: `php://filter`.

An introduction to PHP filters

Over several years, people have taken interest in `php://filter`, another PHP specific protocol (if the name didn't make it obvious). It provides a way to apply transformations to a stream before it is returned. The syntax is:

```
php://filter/[filters...]/resource=[resource]
```

The resource can be anything we've already talked about in the previous section: a simple file, an HTTP response, a file from an FTP server...

The filters are a list of transformation that you want PHP to apply on the stream. Here, we ask PHP to convert the contents of the resource to base64 using the `convert.base64-encode` filter:

```
php://filter/convert.base64-encode/resource=/etc/passwd
```

It returns:

```
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYXNoCmJpbjp4OjE6MTpiaW46L2Jpbjovc2Jpbi9u
b2xvZ2luCmRhZW1vbjp4OjI6MjpkYWVtb246L3NiaW46L3NiaW4vbm9sb2dpbgphZG06eDozOjQ6
...
Yi92bnN0YXQ6L2Jpbi9mYWxzZQpyZWRpczp4OjEwMjoxMDM6cmVkaXM6L3Zhci9saWIvcmVkaXM6
L2Jpbi9mYWxzZQo=
```

You can add as many filters as you need. Here, I ask PHP to base64-encode the stream twice:

```
php://filter/convert.base64-encode|convert.base64-encode/resource=/etc/passwd
```

And I get:

```
Y205dmREcDRPakE2TURweWIyOTBPaTl5YjI5ME9pOWlhVzR2WVhOb0NtSnBianA0T2pFNk1UcGlh
...
RXdNam94TURNNmNtVmthWE02TDNaaGNpOXNhV0l2Y21Wa2FYTTZMMkpwYmk5bVlXeHpaUW89
```

Obviously, base64-encoding is not the only thing you can do. Many filters are available. They include:

- `string.upper`, which converts a string to uppercase
- `string.lower`, which converts a string to lowercase
- `string.rot13`, which does some BC crypto
- `convert.iconv.X.Y`, which converts charset from X to Y

Let's take a look at the last filter: `convert.iconv.X.Y`. Say that I need to convert my file from UTF8 to UTF16. I can use:

```
php://filter/convert.iconv.UTF-8.UTF-16/resource=/etc/passwd
```

Which gives (in hex form):

```
00000000: fffe 7200 6f00 6f00 7400 3a00 7800 3a00  ..r.o.o.t.:.x.:.
00000010: 3000 3a00 3000 3a00 7200 6f00 6f00 7400  0.:.0.:.r.o.o.t.

                                ...

00000a40: 2f00 6200 6900 6e00 2f00 6600 6100 6c00  /.b.i.n./.f.a.l.
00000a50: 7300 6500 0a00                           s.e...
```

The multitude of filters and the possibility to chain them lead to some great research on PHP, such as here, here, or here. In fact, using precisely picked filters (a *filter chain*), attackers can do fantastic things, such as changing the contents of a file entirely, or using an error-based oracle to extract its bytes one-by-one.

For instance, here is a filter chain that preprends `Hello world!` to `/etc/passwd`:

```
php://filter/convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.CSGB2312.UTF-32|
convert.iconv.IBM-1161.IBM932|convert.iconv.GB13000.UTF16BE|convert.iconv.864.UTF-32LE|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.IBM860.UTF16|
convert.iconv.ISO-IR-143.ISO2022CNEXT|convert.base64-decode|convert.base64-encode|
convert.iconv.855.UTF7|convert.iconv.INIS.UTF16|convert.iconv.CSIBM1133.IBM943|
convert.iconv.GBK.SJIS|convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|
convert.iconv.L5.UTF-32|convert.iconv.ISO88594.GB13000|convert.iconv.BIG5.SHIFT_JISX0213|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.JS.UNICODE|
convert.iconv.L4.UCS2|convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|
convert.iconv.CP-AR.UTF16|convert.iconv.8859_4.BIG5HKSCS|convert.base64-decode|convert.base64-
encode|
convert.iconv.855.UTF7|convert.iconv.SE2.UTF-16|convert.iconv.CSIBM921.NAPLPS|
convert.iconv.CP1163.CSA_T500|convert.iconv.UCS-2.MSCP949|convert.base64-decode|
convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.L4.UTF32|convert.iconv.CP1250.UCS-2|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.UTF8.UTF16LE|
convert.iconv.UTF8.CSISO2022KR|convert.iconv.UTF16.EUCTW|convert.iconv.ISO-8859-14.UCS2|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.INIS.UTF16|
convert.iconv.CSIBM1133.IBM943|convert.iconv.GBK.BIG5|convert.base64-decode|convert.base64-
encode|
convert.iconv.855.UTF7|convert.iconv.CP1046.UTF16|convert.iconv.ISO6937.SHIFT_JISX0213|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.L5.UTF-32|
convert.iconv.ISO88594.GB13000|convert.iconv.BIG5.SHIFT_JISX0213|convert.base64-decode|
convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.IBM869.UTF16|
convert.iconv.L3.CSISO90|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|
convert.iconv.ISO2022KR.UTF16|
convert.iconv.L6.UCS2|convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|
convert.iconv.L6.UNICODE|convert.iconv.CP1282.ISO-IR-90|convert.base64-decode|convert.base64-
encode|
convert.iconv.855.UTF7|convert.iconv.JS.UNICODE|convert.iconv.L4.UCS2|
convert.iconv.UCS-2.OSF00030010|
convert.iconv.CSIBM1008.UTF32BE|convert.base64-decode|convert.base64-encode|
convert.iconv.855.UTF7|
convert.iconv.IBM869.UTF16|convert.iconv.L3.CSISO90|convert.base64-decode|convert.base64-encode|
convert.iconv.855.UTF7|convert.iconv.CP861.UTF-16|convert.iconv.L4.GB13000|
convert.iconv.BIG5.JOHAB|
convert.base64-decode|convert.base64-encode|convert.iconv.855.UTF7|convert.iconv.L6.UNICODE|
convert.iconv.CP1282.ISO-IR-90|convert.base64-decode|convert.base64-encode|
convert.iconv.855.UTF7|
convert.iconv.INIS.UTF16|convert.iconv.CSIBM1133.IBM943|convert.iconv.GBK.SJIS|convert.base64-
decode|
convert.base64-encode|convert.iconv.855.UTF7|convert.base64-decode/resource=/etc/passwd
```

And the result:

```
Hello, world!!!root:x:0:0:root:/root:/bin/bash...
```

PHP filters: prefix, suffix, and crash

Sadly, file reads are not always as easy as a:

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

Oftentimes, the file will not be returned as-is, but it will be parsed or checked in some way. As an example, I often encountered variations of this piece of code, that expects your file to be valid JSON:

```
$data = file_get_contents($_POST['url']);
$data = json_decode($data);
echo $data->message;
```

We have a file read here, but the contents is then JSON-deserialized, and only part of the document is returned. In order to read standard files, such as `/etc/passwd`, we would need to add an arbitrary prefix **and suffix** to a stream. Something like: `{"message": "<contents-of-/etc/passwd>"}`. In late 2023, the state of affairs was that you could use `php://filter` chains to add a prefix to a stream, but not a suffix. So I started working on an algorithm to do the latter*.

At the time, I did not know anything about charsets, or encodings (to be perfectly honest, I still don't know the difference). To get started, I built a bruteforce script which stacked a few `iconv` filters on top of each other, and displayed the results. Something like:

```
php://filter/convert.iconv.A.B/convert.iconv.C.D/convert.iconv.E.F/resource=data:,test123
```

And a some point, my «fuzzer» **crashed**.

Having worked with PHP for most of my life, I was quick to point fingers. But little did I know, the bug resided much lower in the call chain: all the way down to the **glibc**.

** Note: the research yielded a tool that was published in December, 2023: wrapwrap.*

CVE-2024-2961: A bug in the glibc

When PHP converts from one charset to another, it uses **iconv**, an API to «convert characters in input buffer using conversion descriptor to output buffer». This API is, on Linux, implemented by the glibc.

The API is very simple. You first open a conversion descriptor, that indicates the input and the output charsets.

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

Then, you can use `iconv()` to convert the input buffer `inbuf` to its new charset in `outbuf`, the output buffer.

```
size_t iconv(iconv_t cd,
             char **restrict inbuf, size_t *restrict inbytesleft,
             char **restrict outbuf, size_t *restrict outbytesleft);
```

Buffer management is the responsibility of the caller. If the output buffer is not big enough, `iconv()` will return an error indicating so, and you will be able to reallocate `outbuf` and continue the conversion by calling `iconv()` again. What the function garantees is that it will never read more than `inbytesleft` bytes from `inbuf`, or **write** more than `outbytesleft` bytes to `outbuf`. Never? Well, in *theory*...

Out-of-bound write when converting to ISO-2022-CN-EXT

It just so happens that, when converting data to the `ISO-2022-CN-EXT` charset, *iconv* might fail to check if enough space remains in the output buffer before writing to it.

In reality, `ISO-2022-CN-EXT` is actually a collection of charsets: when it needs to encode a character, it will choose the appropriate charset, and emit an escape sequence to indicate that the decoder needs to switch to such charset.

The code below is the part which is responsible for emitting such escape sequences. It is made of 3 `if` blocks, that each write different escape sequences to `outbuf` (pointed to by `outptr`). If you look at the first one [1], you can see that it is prefixed by another `if()`

block that checks that the output buffer is big enough to fit 4 characters. The other two `if()` [2][3] do **not**. As a result, the escape sequences might be written out-of-bounds.

```
// iconvdata/iso-2022-cn-ext.c

/* See whether we have to emit an escape sequence.  */
if (set != used)
{
    /* First see whether we announced that we use this
        character set.  */
    if ((used & SO_mask) != 0 && (ann & SO_ann) != (used << 8)) // [1]
    {
        const char *escseq;

        if (outptr + 4 > outend) // <-------------------- BOUND CHECK
        {
            result = __GCONV_FULL_OUTPUT;
            break;
        }

        assert(used >= 1 && used <= 4);
        escseq = ")A\0\0)G)E" + (used - 1) * 2;
        *outptr++ = ESC;
        *outptr++ = '









;
        *outptr++ = *escseq++;
        *outptr++ = *escseq++;

        ann = (ann & ~SO_ann) | (used << 8);
    }
    else if ((used & SS2_mask) != 0 && (ann & SS2_ann) != (used << 8)) // [2]
    {
        const char *escseq;

        // <-------------------- NO BOUND CHECK

        assert(used == CNS11643_2_set); /* XXX */
        escseq = "*H";
        *outptr++ = ESC;
        *outptr++ = '









;
        *outptr++ = *escseq++;
        *outptr++ = *escseq++;

        ann = (ann & ~SS2_ann) | (used << 8);
    }
    else if ((used & SS3_mask) != 0 && (ann & SS3_ann) != (used << 8)) // [3]
    {
        const char *escseq;

        // <-------------------- NO BOUND CHECK

        assert((used >> 5) >= 3 && (used >> 5) <= 7);
        escseq = "+I+J+K+L+M" + ((used >> 5) - 3) * 2;
        *outptr++ = ESC;
        *outptr++ = '
```

```
;
        *outptr++ = *escseq++;
        *outptr++ = *escseq++;

        ann = (ann & ~SS3_ann) | (used << 8);
    }
}
```

To trigger the bug, we need to force `iconv()` to emit an escape sequence, right before the end of the output buffer. To do so, we can use exotic characters, such as: 劄, 䂚, 峛 or 湿. The result is an overflow of 1 to 3 bytes, with the following values:

- `$*H` [`24 2A 48`]
- `$+I` [`24 2B 49`]
- `$+J` [`24 2B 4A`]
- `$+K` [`24 2B 4B`]
- `$+L` [`24 2B 4C`]
- `$+M` [`24 2B 4D`]

A quick POC demonstrates the bug:

```
/*
$ gcc -o poc ./poc.c && ./poc
*/
...

void hexdump(void *ptr, int buflen)
{
    ...
}

void main()
{
    iconv_t cd = iconv_open("ISO-2022-CN-EXT", "UTF-8");

    char input[0x10] = "AAAAA劄";
    char output[0x10] = {0};

    char *pinput = input;
    char *poutput = output;

    // Same size for input and output buffer: 8 bytes
    size_t sinput = strlen(input);
    size_t soutput = sinput;

    iconv(cd, &pinput, &sinput, &poutput, &soutput);

    printf("Remaining bytes (should be > 0): %zd\n", soutput);

    hexdump(output, 0x10);
}
```

This produces, on vulnerable systems:

```
$ gcc -o poc ./poc.c && ./poc
Remaining bytes (should be > 0): -1
000000: 41 41 41 41  41 1b 24 2a  48 00 00 00  00 00 00 00    AAAA A.$* H... ....
```

Nine bytes have been written, despite telling `iconv()` to write eight at most.

Checking the commit history, I noticed that the bug was very old: it appeared in year 2000, making it 24 years old.

Now, what could be done with this bug?

Conditions and primitive

With this vulnerability, I had a 1-to-3 bytes overflow, with non-controlled characters. This was not much. In addition to this, there were preconditions. I needed find a call to `iconv()` in which I:

- controlled the output charset (`ISO-2022-CN-EXT`)
- controlled part of the input buffer (to put in the beautiful chinese characters)

With this in mind, I started looking for targets. From grepping `iconv` in my `/lib` and `/bin` directory to iterating over hundreds of OSS projects, I found a few interesting targets. None were actually exploitable.

As an example, let's look at a very promising target: `libxml2`.

libxml2: an ocean of bytes

libxml2 only processes XML in *UTF-8*. If an XML document is not *UTF-8*, it will be converted to it, then processed, and then converted back to its original charset when all is done. The conversions are done using `iconv()`.

As a result, we can meet our preconditions with such a document:

```
<?xml version="1.0" encoding="ISO-2022-CN-EXT"?>
<root>&21124;</root>
```

Note: 21124 is the unicode codepoint for 劄.

Now, remember: the buffer management is the responsibility of the caller. And when `libxml2` uses `iconv()` to convert our document back to its original charset, it allocates an **output buffer** which is **4 times as big** as the **input buffer** (code). Too big for us: we cannot reach the bounds of the buffer to overflow. A dead end.

pkexec: 4 bytes are too many

Another interesting target was **pkexec**, a setuid binary present in many linux distros. The binary lets you pick the charset for every message it outputs, by setting the `CHARSET` environment variable. Example:

```
$ CHARSET=ISO-2022-CN-EXT pkexec 'trigger劄' 2>&1 | hexyl
┌────────┬─────────────────────────┬─────────────────────────┬────────┬────────┐
│00000000│ 43 61 6e 6e 6f 74 20 72 ┊ 75 6e 20 70 72 6f 67 72 │Cannot r┊un progr│
│00000010│ 61 6d 20 74 72 69 67 67 ┊ 65 72 1b 24 2a 48 1b 4e │am trigg┊er•$*H•N│
│00000020│ 4c 61 0f 3a 20 4e 6f 20 ┊ 73 75 63 68 20 66 69 6c │La•: No ┊such fil│
│00000030│ 65 20 6f 72 20 64 69 72 ┊ 65 63 74 6f 72 79 0a    │e or dir┊ectory_ │
└────────┴─────────────────────────┴─────────────────────────┴────────┴────────┘
```

Internally, `pkexec` uses the `GLib` to output its messages. It does the following:

```
#define NUL_TERMINATOR_LENGTH 4

outbuf_size = len + NUL_TERMINATOR_LENGTH;

outbytes_remaining = outbuf_size - NUL_TERMINATOR_LENGTH;
outp = dest = g_malloc (outbuf_size);

...

err = g_iconv (converter, NULL, &inbytes_remaining, &outp, &outbytes_remaining);
```

While it allocates a buffer of *N + 4* bytes, it only tells iconv about *N* bytes. Our overflow is, at most, 3 bytes long. Therefore, no matter how hard we try, we cannot reach outside the buffer.

Another dead end.

Conditions and primitive (updated)

Filled with disappointment, I could only update my list of requirements. To exploit the bug, we needed to:

- control the output charset (`ISO-2022-CN-EXT`)
- control part of the input buffer
- **have a suitable output buffer**

Exploiting PHP filters

Even after looking for days, I did not manage to find a valid target. Blindly grepping for `iconv()` calls in libraries and binaries, going through the open source ecosystem, looking for a triggerable instance of the bug, I was desperately looking for a crash. One. crash. To no avail.

To get my hopes back up, I went back to PHP: after all, it *did* crash, without me even asking for it.

The goal was simple: converting boring file reads vulnerabilities into remote code execution.

A primer on PHP's heap

Note: in this section, and in each section describing the internals of PHP, I will make approximations and pass over certain things.

To get where this is going, we need to understand how the PHP heap works (at least, *part of it*). Do not worry, it is a very simple heap.

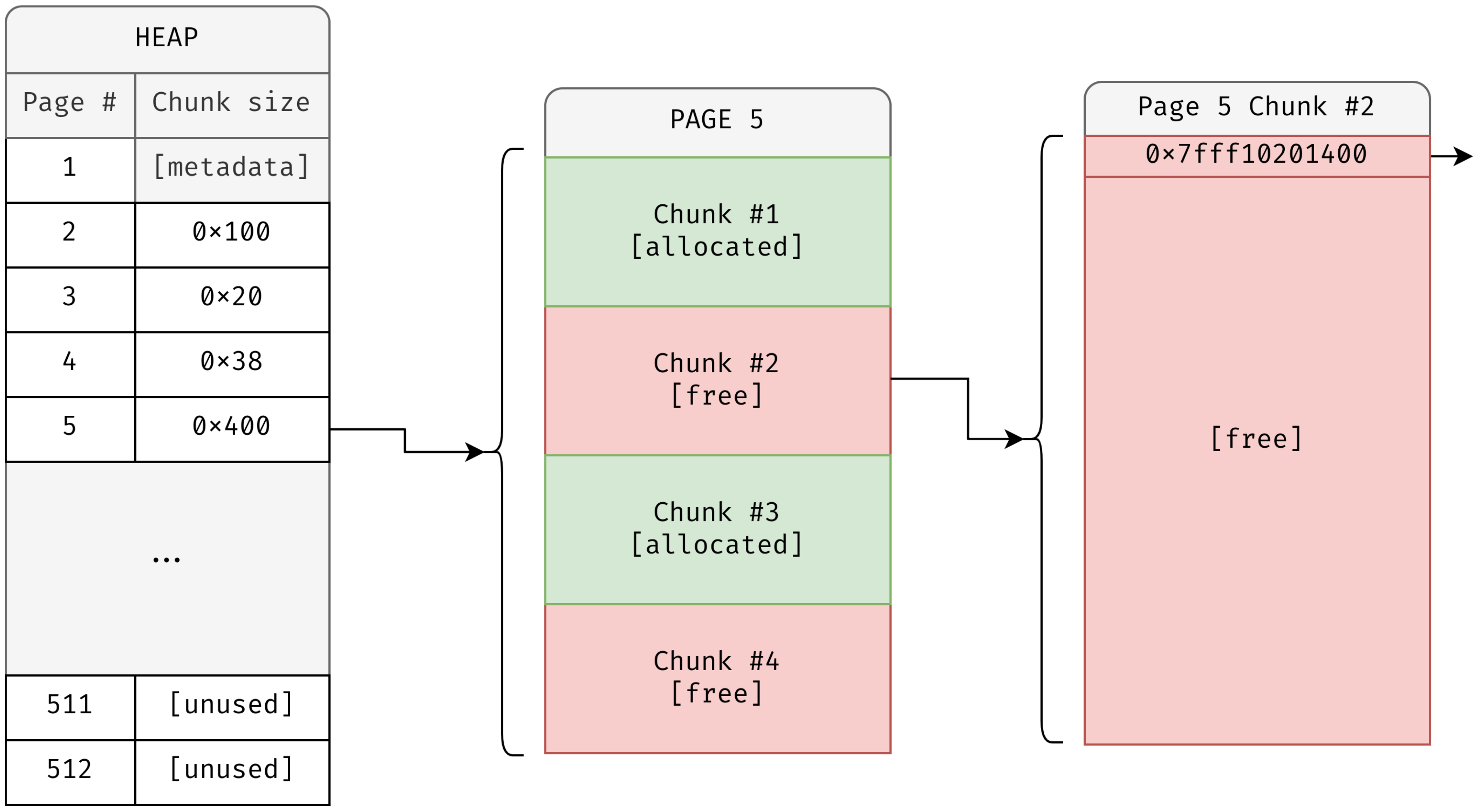
To allocate with PHP, you use `emalloc(N)`, with *N* the number of bytes that you want. You get back a pointer to a chunk (a memory block) that can store at least *N* bytes. When you're done with your chunk, you free it using `efree(ptr)`. PHP has chunks of various sizes (8, 0x10, 0x18, ... 0x200, 0x280, ...).

The PHP heap consists of a region of 2MB, split in 512 pages of 0x1000 bytes. Each page may contain chunks of a certain size. For instance, page 10 may contain chunks of size 0x100, page 11 chunks of size 0x38, page 12 chunks of size 0x180, *etc.* There is no metadata in between chunks.

When you free a chunk, it gets put at the beginning of a singly linked list called the free list. There is a free list for each chunk size. As an example, if I were to free a chunk of size 0x38, it would go in the free list for chunks of size 0x38. If I free a chunk of size 0x200, it goes in the free list for chunks of size 0x200...

To allocate *N* bytes, PHP looks into the free list for the corresponding chunk size, takes out the head, and returns it. If the free list is empty (*i.e.* all available chunks have already been allocated), PHP looks into the heap metadata to find a page that is not used. It then creates empty chunks in such a page, and puts them in the free list.

Free lists are LIFO, meaning that when I free a chunk of some size, it becomes the head of the free list. When I allocate, the head is taken out. This is very similar to the glibc's tcache, but unlimited.



Visual representation of PHP's heap

In the example above, we have a visual representation of the heap on the left. It contains 512 pages, and here page *5* stores chunks of size *0x400*. If we look at the contents of this page, we can see that it contains *4* chunks (as *4 × 0x400 = 0x1000*, the size of a page). Here, chunk #1 and #3 are allocated, and chunk #2 and #4 are freed. Therefore, they are in the free list for the chunks of size *0x400*.

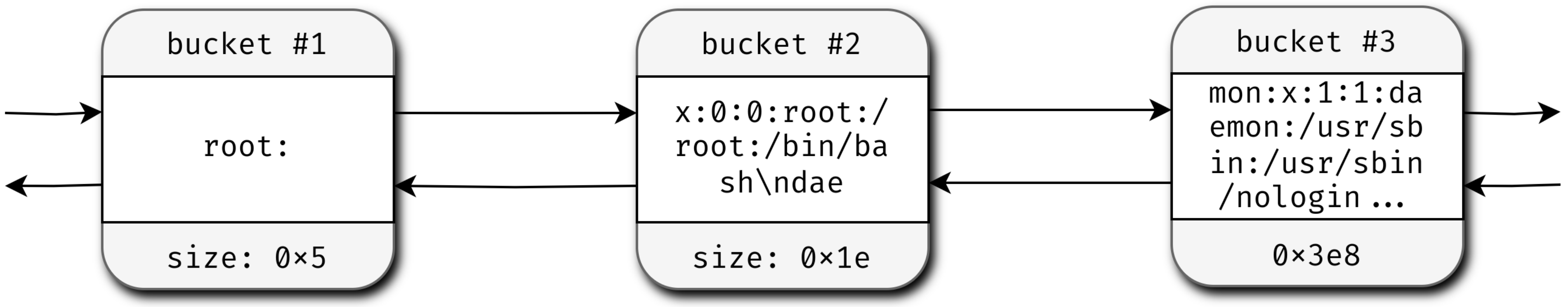
The free list being a singly-linked list, each unallocated chunk contains, as its 8 first bytes, a pointer to the next free chunk. That is what we see in chunk #2: a pointer to *0x7ff10201400*, which is the address of the next free chunk for size *0x400*. Now, if we were to **overflow** from chunk #1 to chunk #2, we'd overwrite this pointer. That is a good starting point for an exploit: even with a one-byte overflow, we can change a free list pointer, thus **altering the free list**.

One should note that **PHP creates a new heap** for each HTTP request. This is one of the reasons that make remote PHP exploitation hard – but this will get covered in part 2.

PHP filters internals

Now that we know how PHP allocates and deallocates, we can have a look at how PHP handles a `php://filter/` string. We're lucky here: we don't need to get into details of internal PHP structures, such as `zval`, `zend_string`, `zend_array`, to name a few.

To process a filter, PHP will first get the stream (*i.e.* read the resource). It stores the stream in a collection of buckets, doubly-linked structures that each contain a buffer of some size. Sticking to our `/etc/passwd` example, we may have 3 buckets: the first one may contain the first 5 bytes of the file, the second bucket 30 more, and the third one another 1000. Linked together, they constitute a *bucket brigade*.

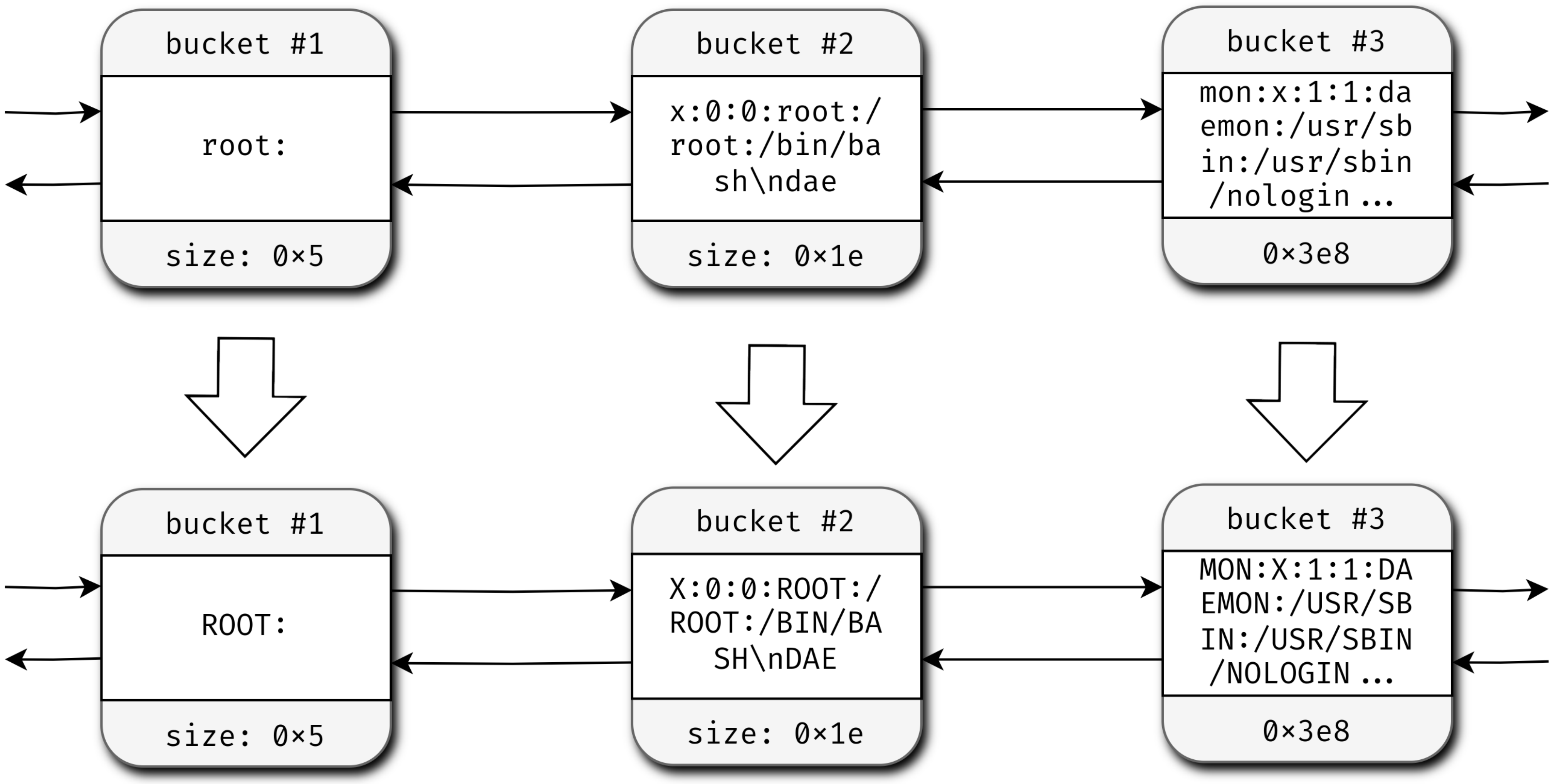


bucket #1
root:
size: 0×5
bucket #2
x:0:0:root:/root:/bin/bash\ndae
size: 0×1e
bucket #3
mon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin ...
0×3e8

A bucket brigade of 3 buckets containing `/etc/passwd`

This is a standard way of representing a stream as a collection of buffers of various sizes. You could imagine this as a list of packets received over the network. Packet 1 contains the first N bytes of data, packet 2 contains the next M bytes, *etc.*

Now that PHP has read the contents of a resource into a stream, represented by a *bucket brigade*, it can apply filters onto it. It takes the first filter, and processes the first bucket. To do so, it allocates an output buffer of the same size as the bucket's buffer (in our example, that would be 5 bytes), and does the conversion. If the filter is `string.upper` for instance, it converts every lowercase character in the input buffer into its uppercase equivalent in the output buffer. It can then create a new bucket which points to this buffer.



bucket #1
root:
size: 0×5
bucket #2
x:0:0:root:/root:/bin/bash\ndae
size: 0×1e
bucket #3
mon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin ...
0×3e8
bucket #1
ROOT:
size: 0×5
bucket #2
X:0:0:ROOT:/ROOT:/BIN/BASH\nDAE
size: 0×1e
bucket #3
MON:X:1:1:DAEMON:/USR/SBIN:/USR/SBIN/NOLOGIN ...
0×3e8

Applying `string.upper` on a bucket brigade

It then processes bucket 2, then bucket 3, and so on until it reaches the last bucket. It now has a *new bucket brigade* with every output bucket. It can now apply the second filter onto this brigade, and keep going until the last filter has been processed.

Situation and goals

We're done with the definitions. Let's go back to our original vulnerability: a file read.

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

Now that we can trigger a memory corruption using the `convert.iconv.XXX.ISO-2022-CN-EXT` filter, we want remote code execution. And it does not look too hard to exploit.

First, since we have a file read primitive, we can read binaries (PHP, Apache, ...). We can even download the libc and check if it is patched! We don't care about ASLR and PIE either: we can read `/proc/self/maps`. Finally, it feels like we can almost arbitrarily allocate or deallocate buffers using buckets, which is convenient.

On the other side, there are many contexts in which you can get a file read primitive: you might get it on a Symfony 4.x running on PHP 7.0, or in an obscure Wordpress plugin running on PHP 8.3, or even during blackbox assessments. The *ideal* exploit needs to be resilient: it has to work against most targets, without any tweaks.

Exploitation

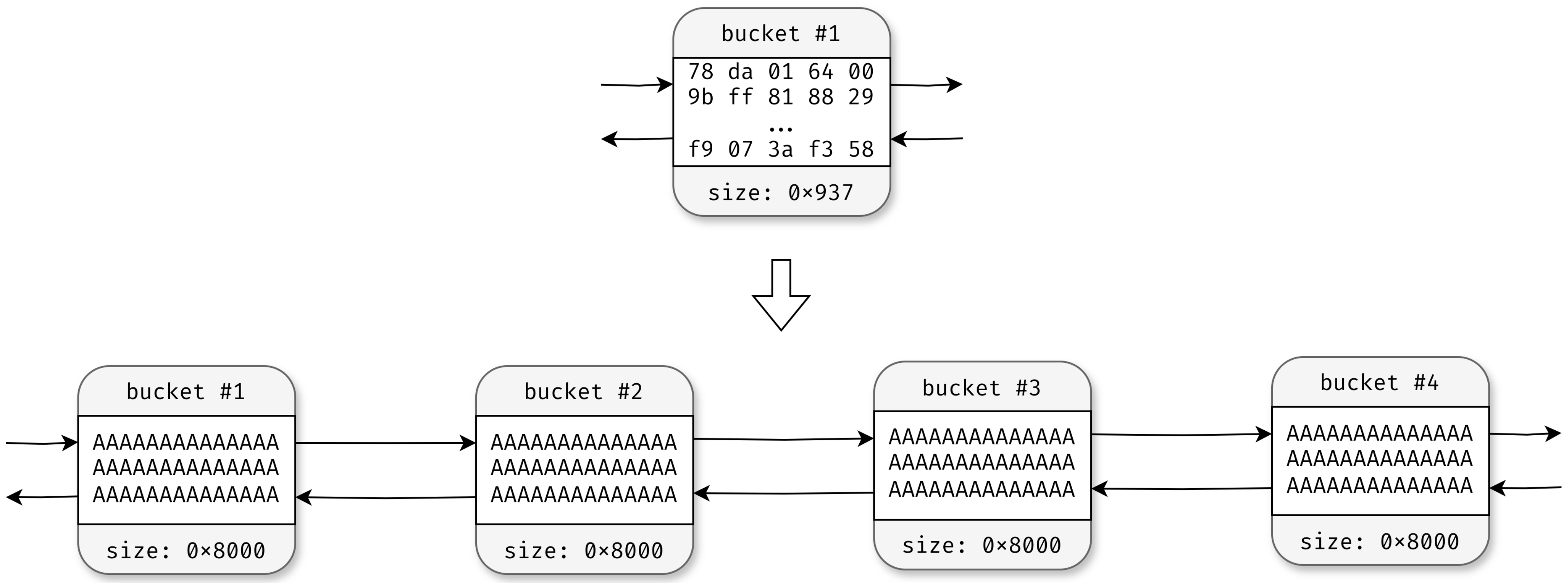
With all this in mind, let's start exploiting. The idea is to use a single-byte buffer overflow to modify the LSB of the pointer to a free chunk, in order to get control over some free list.

Single bucket

The first problem that we face is that, despite having the bucket brigade technology, PHP only creates **one** bucket. If you read a file, you get one bucket with the whole file. If you request an HTTP URL, PHP creates one bucket with the whole HTTP response. With `ftp://`, one bucket as well. This is very *impractical*, to say the least: we cannot use buckets to pad the heap, spray stuff, or even use the altered free list.

Think about it: with a single bucket, we can overflow into a free chunk and modify a free list, but we are then out of buckets, and we need at least 2 more allocations to use our altered free list!

Luckily for us, one filter saves the day: `zlib.inflate`. This filter takes our stream and decompresses it. To do so, it allocates a buffer of 8 pages (*0x8000* bytes) and inflates our stream into it. If it is not big enough, it creates a new buffer of the same size to store the rest of the data. If these two are still not enough, it creates another buffer. Each buffer is then added to a bucket. Perfect: we can use this filter to create as many buckets as we want, which is a good step forward.



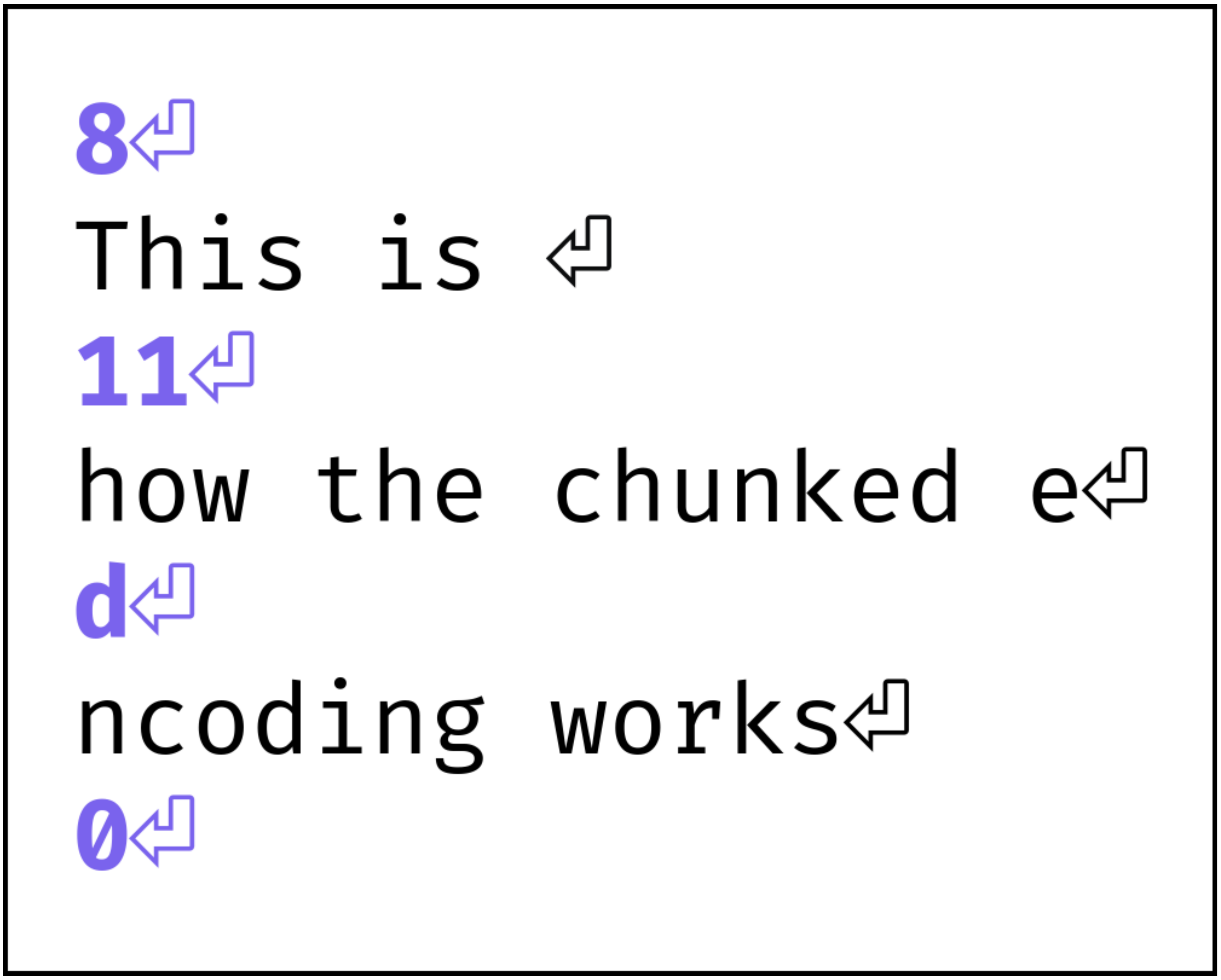
Applying `zlib.inflate` to create several buckets

However, these buckets have buffers of size *0x8000*, which is not a good size for exploitation; buffer these sizes get allocated in a different fashion than the one I told you about, and don't land into free list when freed. We need to resize our buckets.

Dechunking properly

To do so, we'll use a filter that is not documented by PHP, but is well known by attackers: `dechunk`. This filter decodes a string which is HTTP-chunked encoded.

HTTP-chunked is a very simple encoding where you send data by chunk (not a *heap* chunk, a *data* chunk). First, you send a size as ASCII-hex, followed by a newline, and then, the chunk of data of the corresponding size, followed by a newline. Then you send another size, another chunk, another size, another chunk, and you indicate the end of the data by sending a size of *0* (*zero*).

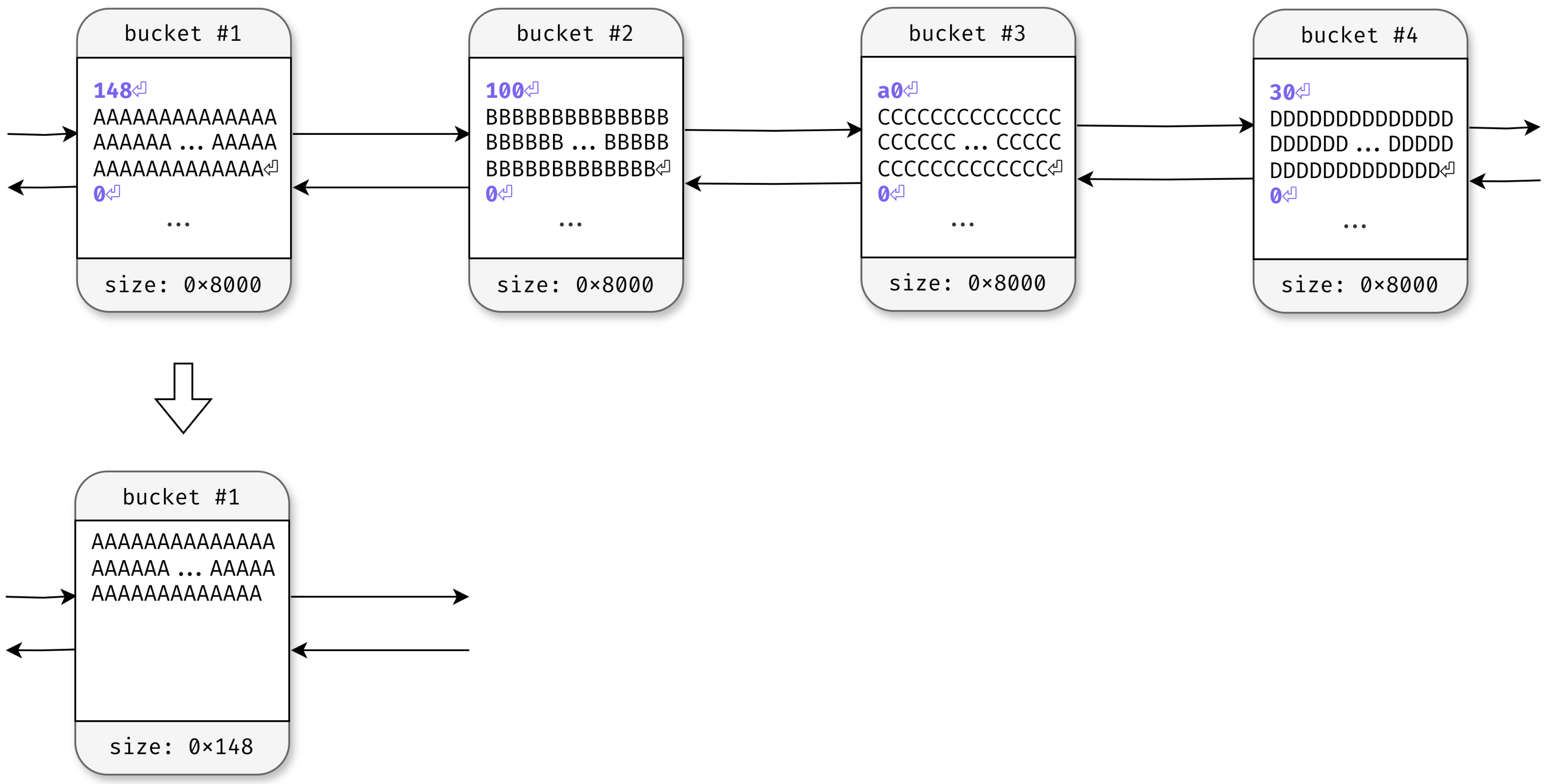


```
8⏎
This is ⏎
11⏎
how the chunked e⏎
d⏎
ncoding works⏎
0⏎
```

Data encoded using the HTTP-chunked encoding

In the example, the first chunk is *8* bytes long, the second is *17* bytes long (*11h*), and the last one *13* bytes long. After `dechunking`, the result would be: `This is how the chunked encoding works.`

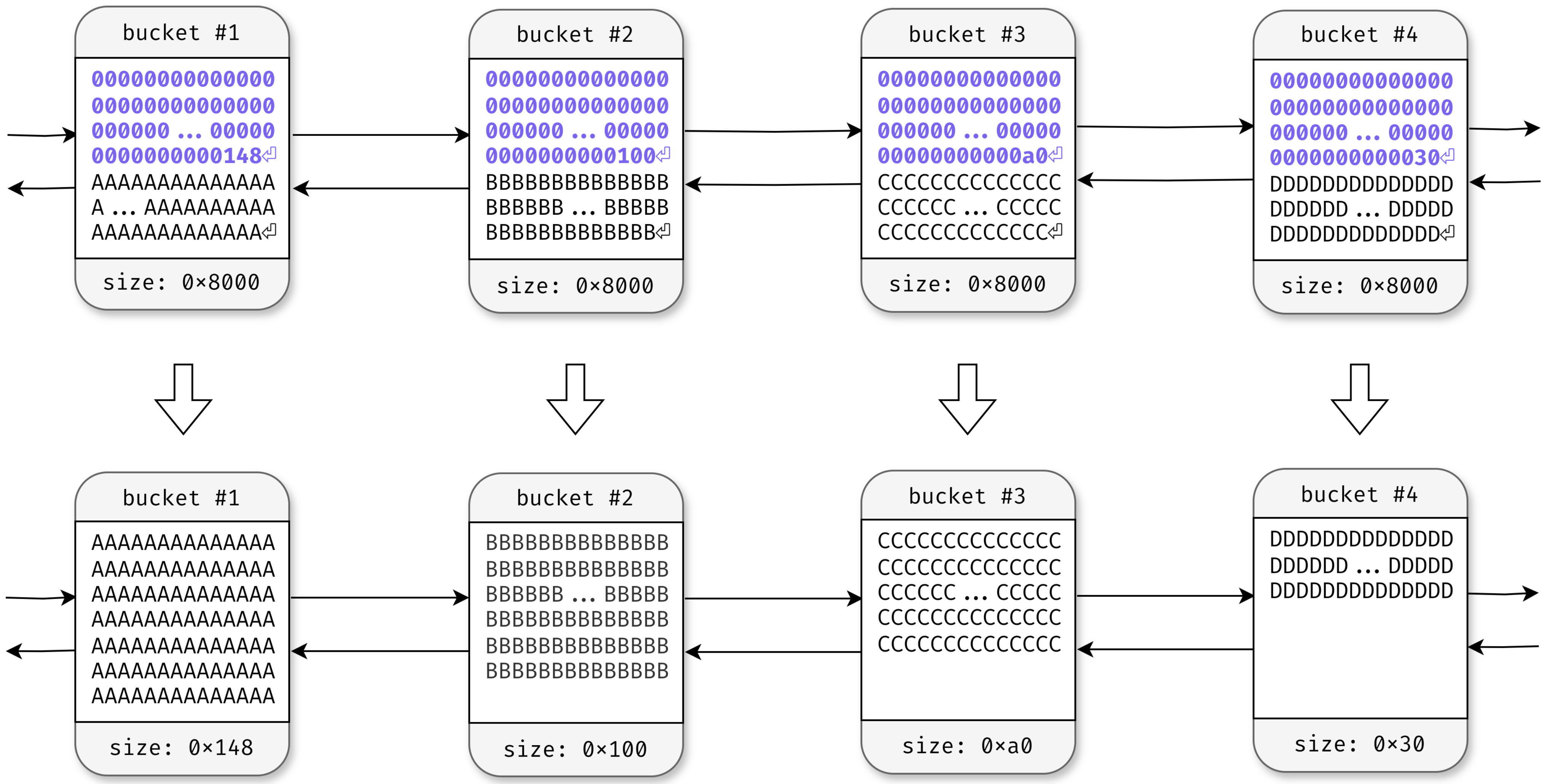
With this filter, resizing our buckets sounds like a child's play: in each bucket, we prefix the data with the size that we want (e.g. *0x148* in the first one, *0x100* for the second, etc.), then we put the data, and then a final *0* indicating that we are done.



Setting up buckets for `dechunk`

It looks good, but it will **not work**. Although being processed separately, the buckets are not **independent**: they are all parsed as one big stream. When the `dechunk` filter processes the stream, it reads the size in the first bucket, *0x148*, takes 0x148 bytes out, and then reads a size of *zero*, which causes it to stop parsing. It does not go to the second bucket. It just stops parsing entirely. The final result of our operation is that we went from having several buckets (good) back to a single bucket (bad).

Luckily, it is not too difficult to find a way to circumvent this: in each bucket, we provide one size, and one data chunk. To do so, instead of naively writing a size, we pad it with thousands of zeroes, in order to get something like this:



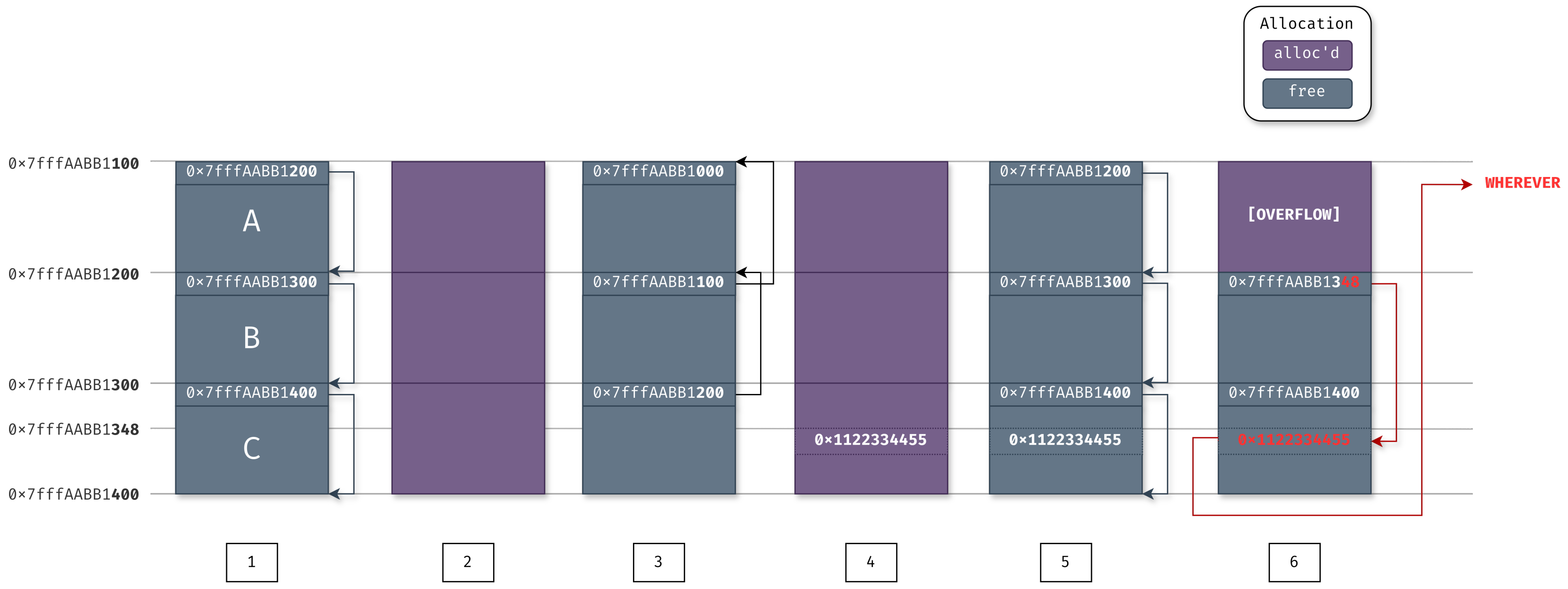
Properly setting up buckets for `dechunk`

Now, after bucket 1 has been processed, the dechunk parser jumps to bucket 2, ready to read a new size, and then to bucket 3, and so on. It works! We can now create **as many buckets** as we want, with the **size** that we want. We have made a huge leap forward.

Free list control: write-what-where

Our goal is now to alter some free list by overwriting the LSB of some pointer with the value *48h* (`H` in ASCII). In order to get the same effect unconditionally, we target chunks of size *0x100*, because the LSB of chunks' addresses is always zero. This means that the effect of our overflow is always the same: **adding 0x48 to a chunk pointer**.

To exploit, we follow a very standard procedure of 6 steps. We name the free list for chunks of size 0x100 `FL[0x100]`.

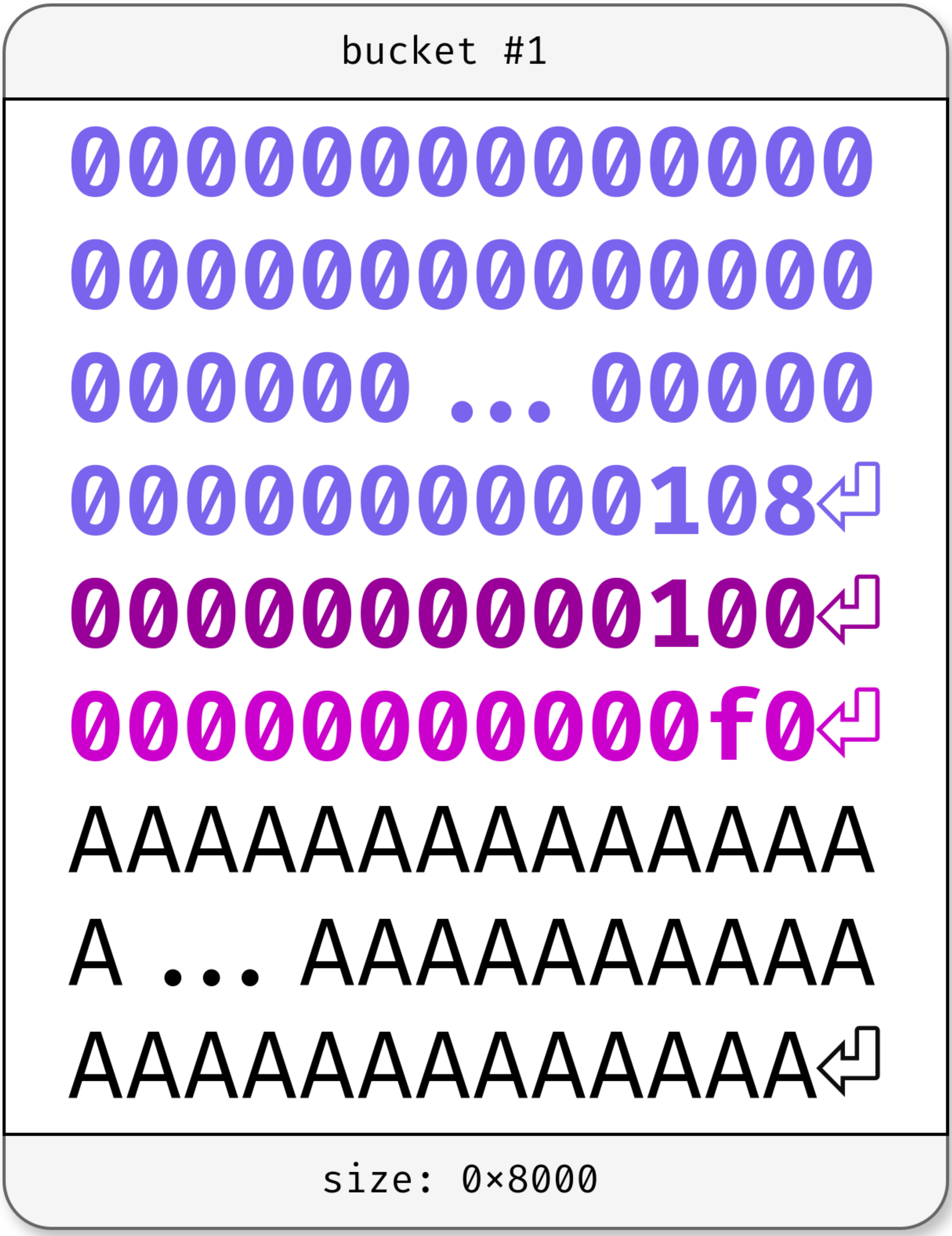


Controlling `FL[0x100]`

Consider that we have managed to pad the heap by allocating lots of *0x100* chunks. We thus have, somewhere in memory, three contiguous free chunks A, B, and C, with A being head of `FL[100]`. A points to B, which points to C. We can allocate the 3 of them (step 2), and free them again (step 3). At this point, the free list is reversed: we have C→B→A. We then allocate again, but this time we put an arbitrary pointer `0x1122334455` at offset 48h of C (step 4). We free them again (step 5), and get the exact same state as in step 1, but this time with a small difference: at `C+48h`, we have an arbitrary pointer. We can now perform the overflow from chunk `A`, which shifts the pointer contained in `B`. It now points to `C+48h`, and as a result the free list is now `B→C+48h→0x1122334455`. With `3` more allocations, we can make PHP allocate at our arbitrary address.

We now have a **write-what-where**; this is almost over.

But let me get back to the implementation of the exploit. In the various steps described here, we have chunks that get allocated, and then freed. But we cannot truly get rid of buckets: we can only make their size change. However, we are only **interested in chunks of size 0x100**. It is as if the other chunks did not exist. Therefore, I built each bucket as an **HTTP-chunked russian doll**:



A russian doll of a bucket: its size changes on each dechunk

For each step of the exploit, the dechunk filter is called: the size of each bucket thus changes. Some have their size become 0x100, thus "appearing" in the exploitation, and some become smaller, thus disappearing. It gives us a perfect way to have buckets materialize at a specific moment, and throw them away when we don't need them anymore.

With that out of the way, let's get code execution.

Code execution

Although we see memory regions by reading `/proc/self/maps`, we don't *precisely* know where we are in the heap. Luckily, we can completely ignore this problem by locating PHP's heap. It is easily identifiable, due to its alignment (*~0x1fffff*) and size (2MB). At its top resides a `zend_mm_heap` structure, that contains very useful fields:

```
struct _zend_mm_heap {
    ...
    int                 use_custom_heap;
    ...
    zend_mm_free_slot *free_slot[ZEND_MM_BINS]; /* free lists for small sizes */
    ...
    union {
        struct {
            void      *(*_malloc)(size_t);
            void       (*_free)(void*);
            void      *(*_realloc)(void*, size_t);
        } std;
    } custom_heap;
};
```

First, it holds every free list. By overwriting the free lists, we get an arbitrary number of *write-what-where*, with arbitrary sizes. We can use these to overwrite the last field, `custom_heap`, which contains alternative functions for `emalloc()`, `efree()` and `erealloc()` (similarly to `__malloc_hook` and its siblings in the glibc). Then, we set `use_custom_heap` to 1, and call `free()` on a bucket, giving us an arbitrary function call with a controlled argument. Since we have access to binaries using the file read, we could build fancy ROP chains, but we want something as generic as possible; I therefore set `custom_heap._free` to `system`, allowing us to run an arbitrary bash command, in a CTF fashion.

Note: There are a few (many) details to the exploitation that I've left out, but the exploit is heavily commented.

Exploit performance

Our exploit runs 3 requests: it downloads `/proc/self/maps`, and extracts both the address of PHP's heap and the filename of the libc. It then downloads the libc binary to extract the address of `system()`. Finally, it performs a final request to perform the overflow and execute our arbitrary command.

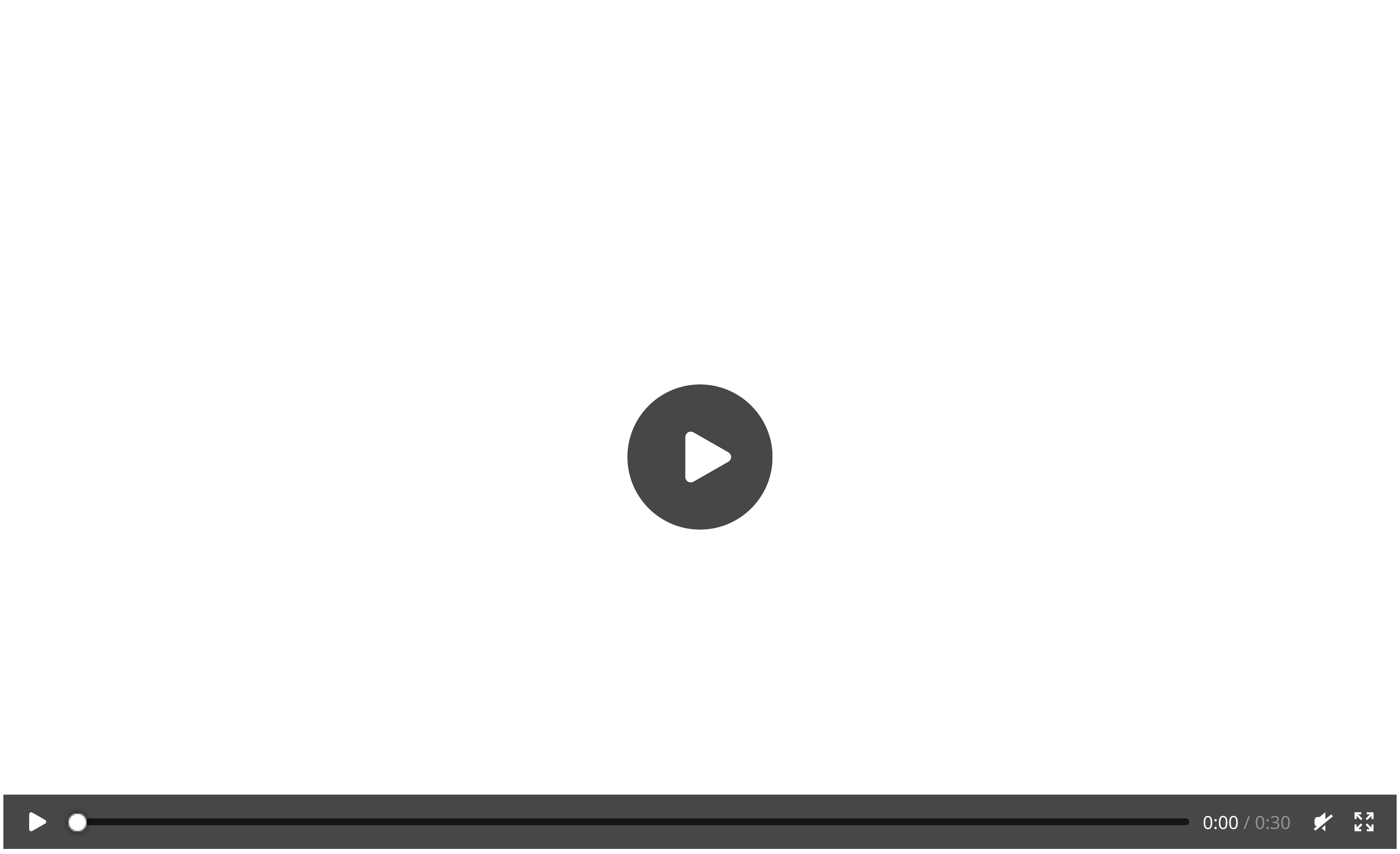
It performs very well:

- Works against **any target**
  - From PHP 7.0.0 (2015) to 8.3.7 (2024)
  - Any PHP application: Wordpress, Laravel, etc.
- It is **100% reliable**
  - Due to its implementation, it will never (?) produce a crash
  - A binary exploit that feels like a web exploit!
- Payload is **smaller** than **1000 bytes**
  - by using `zlib.inflate` and only 12 filters, the payload is really small
  - It fits in a GET request
- Self-contained exploit
  - No need to send additional parameters as GET or POST: the exploit does everything by itself, from padding the heap to setting up the free list, and finally getting code execution

It is a single, less than 1000 bytes payload that results in **remote code execution** for 10 years worth of PHP versions.

Demo

To illustrate, I'll target a **Wordpress** instance running on **PHP 8.3.x**. To introduce a file read vulnerability, I added the **BuddyForms plugin (v2.7.7)**, which is flawed with CVE-2023-26326. The bug was originally reported as a PHAR deserialisation bug, but Wordpress does not have any deserialisation gadget chains. In any case, the target runs on PHP 8+, so it is not vulnerable to PHAR attacks.



Note: if you read the advisory by the original finder, you may see that right before the file read primitive, a `getimagesize()` call is performed to check if the file is an image. Therefore, to allow the exploit to read `/proc/self/maps` and the libc, I used wrapwrap to make them look like GIF images.

Impact

What is the impact on the PHP ecosystem? This is not a new vulnerability, but instead a new exploitation vector for a vulnerability. There are, however, a multitude of ways to make PHP read a file; file read primitives are very current in web applications.

Standard sinks

Evidently, every standard *file read* sink of PHP is affected: `file_get_contents()`, `file()`, `readfile()`, `fgets()`, `getimagesize()`, `SplFileObject->read()`, *etc.* **File writes** are also affected (`file_put_contents()` and its siblings).

Using vulnerabilities

SQL injection to RCE

If you get an SQL injection on PDO/MySQL, you might be able to use `LOAD DATA LOCAL INFILE`:

```
LOAD DATA LOCAL INFILE 'php://filter/cnext...';
```

XXE

XXEs are now RCEs.

```
<?xml version="1.0" ?>
<!DOCTYPE root [
    <!ENTITY exploit SYSTEM "php://filter/cnext...">
]>
<root>&exploit;</root>
```

As a replacement for PHAR

Contrarily to PHAR attacks, functions that just perform checks on files, such as `file_exists()`, or `is_file()`, are **not** affected. However, in other cases, the exploit can be used as a replacement for PHAR attack, as shown in the demo. Disabling `phar://` or updating to PHP 8 will not save you.

Parsing libraries

Every library that somehow manipulates URLs might be vulnerable. Here are some new targets that I found while working on the exploit:

- meyfa/php-svg: Most popular SVG manipulation library
- symfony/translation: XLIFF parser is vulnerable

For instance, the PHP-SVG library can be attacked with such a payload:

```
<svg width="100" height="100">
    <image href="php://filter/cnext/..." width="1" height="1" />
</svg>
```

HTML-to-PDF parsers, such as dompdf, tcpdf, and others may also be targets.

Class instantiations

Sometimes, when attacking PHP, you encounter the following primitive:

```
new $_GET['cls']($_GET['argument']);
```

This excellent blogpost from PTswarm describes many ways to get a file read from this primitive, which may all be used to trigger the exploit. Examples include `SoapClient`, `Imagick`, `tidy`, or `SimpleXMLElement`.

As an improvement to a gadget chain

If you find a file read `unserialize()` gadget chain, you can use the exploit to upgrade it to RCE. With recent applications and the fact that PHP libraries are using types more and more, it might come in handy.

Others, probably

As long as you control the prefix of a file-read or file-write sink, you have an RCE!

Timeline

- **Last year** Crash found
- **February** Started working on bug
- **March 26th** Bug reported to glibc security team
    - They did an amazing job!
- **April 04th** Bug reported to linux distros
- **April 17th** Bug released as CVE-2024-2961

Note: The glibc security team was fast, courteous, and technically able. They shipped a patch (and all that comes with it) within a week. Many thanks!

Conclusion

This concludes the first part of the series on **CNEXT** (CVE-2024-2961). The exploit is now available on our GitHub. There is still much more to explore: what about *direct calls* to `iconv()` ? What happens if the file read is *blind*?

In part 2, we'll dive deeper in the PHP engine to target an `iconv()` call found in a very popular **PHP webmail**. I'll describe the impact of such direct calls on the PHP ecosystem, and show you some *unexpected* sinks. Finally, in part 3, we'll cover blind file read exploitation.

Stay tuned!

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_.