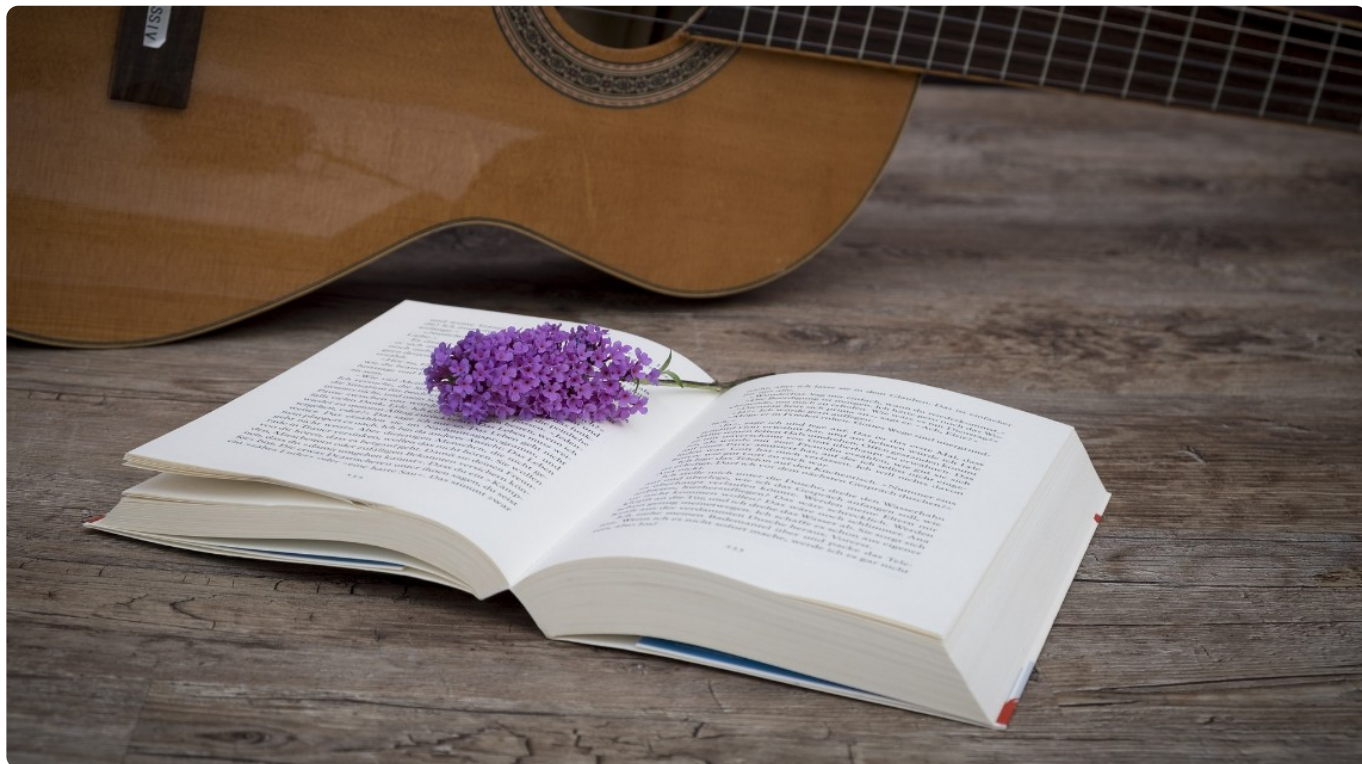


## 29 | 数据和代码：数据就是数据，代码就是代码

2020-05-23 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽

时长 21:14 大小 19.46M



你好，我是朱晔。今天，我来和你聊聊数据和代码的问题。

正如这一讲标题“数据就是数据，代码就是代码”所说，Web 安全方面的很多漏洞，都是源自把数据当成了代码来执行，也就是注入类问题，比如：

客户端提供给服务端的查询值，是一个数据，会成为 SQL 查询的一部分。黑客通过修改这个值注入一些 SQL，来达到在服务端运行 SQL 的目的，相当于把查询条件的数据变为了查询代码。这种攻击方式，叫做 SQL 注入。



对于规则引擎，我们可能会用动态语言做一些计算，和 SQL 注入一样外部传入的数据只能当做数据使用，如果被黑客利用传入了代码，那么代码可能就会被动态执行。这种攻击方式，叫做代码注入。

对于用户注册、留言评论等功能，服务端会从客户端收集一些信息，本来用户名、邮箱这类信息是纯文本信息，但是黑客把信息替换为了 JavaScript 代码。那么，这些信息在页面呈现时，可能就相当于执行了 JavaScript 代码。甚至是，服务端可能把这样的代码，当作普通信息保存到了数据库。黑客通过构建 JavaScript 代码来实现修改页面呈现、盗取信息，甚至蠕虫攻击的方式，叫做 XSS（跨站脚本）攻击。

今天，我们就通过案例来看一下这三个问题，并了解下应对方式。

## SQL 注入能干的事情比你想象的更多

我们应该都听说过 SQL 注入，也可能知道最经典的 SQL 注入的例子，是通过构造 `' or ' 1' = '1` 作为密码实现登录。这种简单的攻击方式，在十几年前可以突破很多后台的登录，但现在很难奏效了。

最近几年，我们的安全意识增强了，都知道使用参数化查询来避免 SQL 注入问题。其中的原理是，使用参数化查询的话，参数只能作为普通数据，不可能作为 SQL 的一部分，以此有效避免 SQL 注入问题。

虽然我们已经开始关注 SQL 注入的问题，但还是有一些认知上的误区，主要表现在以下三个方面：

**第一，认为 SQL 注入问题只可能发生于 Http Get 请求，也就是通过 URL 传入的参数才可能产生注入点。**这是很危险的想法。从注入的难易度上来说，修改 URL 上的 QueryString 和修改 Post 请求体中的数据，没有任何区别，因为黑客是通过工具来注入的，而不是通过修改浏览器上的 URL 来注入的。甚至 Cookie 都可以用来 SQL 注入，任何提供数据的地方都可能成为注入点。

**第二，认为不返回数据的接口，不可能存在注入问题。**其实，黑客完全可以利用 SQL 语句构造出一些不正确的 SQL，导致执行出错。如果服务端直接显示了错误信息，那黑客需要的数据就有可能被带出来，从而达到查询数据的目的。甚至是，即使没有详细的出错信息，黑客也可以通过所谓盲注的方式进行攻击。我后面再具体解释。

**第三，认为 SQL 注入的影响范围，只是通过短路实现突破登录，只需要登录操作加强防范即可。**首先，SQL 注入完全可以实现拖库，也就是下载整个数据库的内容（之后我们会演示），SQL 注入的危害不仅仅是突破后台登录。其次，根据木桶原理，整个站点的安全性

受限于安全级别最低的那块短板。因此，对于安全问题，站点的所有模块必须一视同仁，并不是只加强防范所谓的重点模块。

在日常开发中，虽然我们是使用框架来进行数据访问的，但还可能会因为疏漏而导致注入问题。接下来，我就用一个实际的例子配合专业的 SQL 注入工具 [sqlmap](#)，来测试下 SQL 注入。

首先，在程序启动的时候使用 JdbcTemplate 创建一个 userdata 表（表中只有 ID、用户名、密码三列），并初始化两条用户信息。然后，创建一个不返回任何数据的 Http Post 接口。在实现上，我们通过 SQL 拼接的方式，把传入的用户名入参拼接到 LIKE 子句中实现模糊查询。

 复制代码

```
1 //程序启动时进行表结构和数据初始化
2 @PostConstruct
3 public void init() {
4     //删除表
5     jdbcTemplate.execute("drop table IF EXISTS `userdata`;");
6     //创建表，不包含自增ID、用户名、密码三列
7     jdbcTemplate.execute("create TABLE `userdata` (\n" +
8         "    `id` bigint(20) NOT NULL AUTO_INCREMENT,\n" +
9         "    `name` varchar(255) NOT NULL,\n" +
10        "    `password` varchar(255) NOT NULL,\n" +
11        "    PRIMARY KEY (`id`)\n" +
12        ") ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;");
13     //插入两条测试数据
14     jdbcTemplate.execute("INSERT INTO `userdata` (name,password) VALUES ('test.
15 }
16 @Autowired
17 private JdbcTemplate jdbcTemplate;
18
19 //用户模糊搜索接口
20 @PostMapping("jdbcwrong")
21 public void jdbcwrong(@RequestParam("name") String name) {
22     //采用拼接SQL的方式把姓名参数拼到LIKE子句中
23     log.info("{} ", jdbcTemplate.queryForList("SELECT id,name FROM userdata WHEI
24 }
```

使用 sqlmap 来探索这个接口：

 复制代码

```
1 python sqlmap.py -u http://localhost:45678/sqlinject/jdbcwrong --data name=te:
```

一段时间后，sqlmap 给出了如下结果：

```
[13:09:22] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query - comment)'  
[13:09:22] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query)'  
[13:09:22] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'  
[13:09:32] [INFO] POST parameter 'name' appears to be 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)' injectable  
[13:09:32] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (NULL) - 1 to 20 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (random number) - 1 to 20 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (NULL) - 21 to 40 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (random number) - 21 to 40 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (NULL) - 41 to 60 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (random number) - 41 to 60 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (NULL) - 61 to 80 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (random number) - 61 to 80 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (NULL) - 81 to 100 columns'  
[13:09:32] [INFO] testing 'MySQL UNION query (random number) - 81 to 100 columns'  
POST parameter 'name' is vulnerable. Do you want to keep testing the others (if any)? [y/N]  
sqlmap identified the following injection point(s) with a total of 1079 HTTP(s) requests:  
---  
Parameter: name (POST)  
Type: error-based  
Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)  
Payload: name=test'||(SELECT 0x63657557 WHERE 4419=4419 AND (SELECT 1429 FROM(SELECT COUNT(*),CONCAT(0x71767a7671,(SELECT (ELT(1429=1429,1))) ,0x7176627671,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a))||'  
  
Type: time-based blind  
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)  
Payload: name=test'||(SELECT 0x4757464e WHERE 7724=7724 AND (SELECT 6039 FROM (SELECT(SLEEP(5)))TZNa))||'  
  
[13:10:01] [INFO] the back-end DBMS is MySQL  
[13:10:01] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'  
back-end DBMS: MySQL >= 5.0  
[13:10:01] [WARNING] HTTP error codes detected during run:  
500 (Internal Server Error) - 357 times  
[13:10:01] [INFO] fetched data logged to text files under '/Users/zhuye/.sqlmap/output/localhost'
```

可以看到，这个接口的 name 参数有两种可能的注入方式：一种是报错注入，一种是基于时间的盲注。

接下来，**仅需简单的三步，就可以直接导出整个用户表的内容了。**

第一步，查询当前数据库：

 复制代码

```
1 python sqlmap.py -u http://localhost:45678/sqlinject/jdbcwrong --data name=te:
```

可以得到当前数据库是 common\_mistakes：

 复制代码


```
1 current database: 'common_mistakes'
```

第二步，查询数据库下的表：

```
1 python sqlmap.py -u http://localhost:45678/sqlinject/jdbcwrong --data name=te: 
```

可以看到其中有一个敏感表 userdata:

```
1 Database: common_mistakes
2 [7 tables]
3 +-----+
4 | user          |
5 | common_store  |
6 | hibernate_sequence |
7 | m             |
8 | news          |
9 | r             |
10 | userdata      |
11 +-----+
```

 复制代码

第三步, 查询 userdata 的数据:


```
1 python sqlmap.py -u http://localhost:45678/sqlinject/jdbcwrong --data name=te: 
```

你看, 用户密码信息一览无遗。当然, 你也可以继续查看其他表的数据:

```
1 Database: common_mistakes
2 Table: userdata
3 [2 entries]
4 +-----+-----+-----+
5 | id | name | password |
6 +-----+-----+-----+
7 | 1  | test1 | haha1    |
8 | 2  | test2 | haha2    |
9 +-----+-----+-----+
```

 复制代码

在日志中可以看到, sqlmap 实现拖库的方式是, 让 SQL 执行后的出错信息包含字段内容。注意看下错误日志的第二行, 错误信息中包含 ID 为 2 的用户的密码字段的值 "haha2"。这, 就是报错注入的基本原理:

 复制代码

```
1 [13:22:27.375] [http-nio-45678-exec-10] [ERROR] [o.a.c.c.C.[.[/].[dispatcher:
2 java.sql.SQLIntegrityConstraintViolationException: Duplicate entry 'qbjzqhaha2'
```

既然是这样，我们就实现一个 `ExceptionHandler` 来屏蔽异常，看看能否解决注入问题：

 复制代码

```
1 @ExceptionHandler
2 public void handle(HttpServletRequest req, HandlerMethod method, Exception ex)
3     log.warn(String.format("访问 %s -> %s 出现异常!", req.getRequestURI(), methc
4 }
```

重启程序后重新运行刚才的 `sqlmap` 命令，可以看到报错注入是没戏了，但使用时间盲注还是可以查询整个表的数据：

```
POST parameter 'name' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection point(s) with a total of 73 HTTP(s) requests:
---
Parameter: name (POST)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: name=test' AND (SELECT 1391 FROM (SELECT(SLEEP(5)))Giuy) AND 'XmJO'='XmJO
---
[13:29:47] [INFO] the back-end DBMS is MySQL
[13:29:47] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent potential disruptions
back-end DBMS: MySQL >= 5.0.12
[13:29:47] [INFO] fetching columns for table 'userdata' in database 'common_mistakes'
[13:29:47] [INFO] retrieved:
do you want sqlmap to try to optimize value(s) for DBMS delay responses (option '--time-sec')? [Y/n] Y
[13:30:17] [INFO] adjusting time delay to 1 second due to good response times
3
[13:30:17] [INFO] retrieved: id
[13:30:23] [INFO] retrieved: name
[13:30:34] [INFO] retrieved: password
[13:31:02] [INFO] fetching entries for table 'userdata' in database 'common_mistakes'
[13:31:02] [INFO] fetching number of entries for table 'userdata' in database 'common_mistakes'
[13:31:02] [INFO] retrieved: 2
[13:31:04] [WARNING] (case) time-based comparison requires reset of statistical model, please wait..... (done)
1
[13:31:06] [INFO] retrieved: test1
[13:31:22] [INFO] retrieved: haha1
[13:31:34] [INFO] retrieved: 2
[13:31:37] [INFO] retrieved: test2
[13:31:53] [INFO] retrieved: haha2
Database: common_mistakes
Table: userdata
[2 entries]
+---+-----+
| id | name | password |
+---+-----+
| 1  | test1 | haha1    |
| 2  | test2 | haha2    |
+---+-----+
```

所谓盲注，指的是注入后并不能从服务器得到任何执行结果（甚至是错误信息），只能寄希望服务器对于 SQL 中的真假条件表现出不同的状态。比如，对于布尔盲注来说，可能是“真”可以得到 200 状态码，“假”可以得到 500 错误状态码；或者，“真”可以得到内容输出，“假”得不到任何输出。总之，对于不同的 SQL 注入可以得到不同的输出即可。




在这个案例中，因为接口没有输出，也彻底屏蔽了错误，布尔盲注这招儿行不通了。那么退而求其次的方式，就是时间盲注。也就是说，通过在真假条件中加入 SLEEP，来实现通过判断接口的响应时间，知道条件的结果是真还是假。

不管是什么盲注，都是通过真假两种状态来完成的。你可能会好奇，通过真假两种状态如何实现数据导出？

其实你可以想一下，我们虽然不能直接查询出 password 字段的值，但可以按字符逐一来查，判断第一个字符是否是 a、是否是 b.....，查询到 h 时发现响应变慢了，自然知道这就是真的，得出第一位就是 h。以此类推，可以查询出整个值。

所以，sqlmap 在返回数据的时候，也是一个字符一个字符跳出结果的，并且时间盲注的整个过程会比报错注入慢许多。

你可以引入 [p6spy](#) 工具打印出所有执行的 SQL，观察 sqlmap 构造的一些 SQL，来分析其中原理：


 复制代码

```
1 <dependency>
2   <groupId>com.github.gavlyukovskiy</groupId>
3   <artifactId>p6spy-spring-boot-starter</artifactId>
4   <version>1.6.1</version>
5 </dependency>
```

```
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),5,1))>50,0,1))))nMsM) AND 'konk'='konk%';
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),5,1))>50,0,1))))nMsM) AND 'konk'='konk%';
[13:32:07.024] [http-nio-45678-exec-9] [INFO ] [o.g.t.c.c.sqlinject.SqlInjectController:47 ] - []
[13:32:07.030] [http-nio-45678-exec-10] [INFO ] [p6spy:60 ] - #1581917527030 | took 1ms | statement | connection 444| url
jdbc:mysql://localhost:6657/common_mistakes?characterEncoding=UTF-8&useSSL=false&rewriteBatchedStatements=true
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),5,1))!=50,0,1))))nMsM) AND 'konk'='konk%';
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),5,1))!=50,0,1))))nMsM) AND 'konk'='konk%';
[13:32:07.030] [http-nio-45678-exec-10] [INFO ] [o.g.t.c.c.sqlinject.SqlInjectController:47 ] - []
[13:32:07.037] [http-nio-45678-exec-1] [INFO ] [p6spy:60 ] - #1581917527037 | took 2ms | statement | connection 445| url
jdbc:mysql://localhost:6657/common_mistakes?characterEncoding=UTF-8&useSSL=false&rewriteBatchedStatements=true
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),6,1))>47,0,1))))nMsM) AND 'konk'='konk%';
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),6,1))>47,0,1))))nMsM) AND 'konk'='konk%';
[13:32:07.037] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.c.sqlinject.SqlInjectController:47 ] - []
[13:32:07.043] [http-nio-45678-exec-2] [INFO ] [p6spy:60 ] - #1581917527043 | took 1ms | statement | connection 446| url
jdbc:mysql://localhost:6657/common_mistakes?characterEncoding=UTF-8&useSSL=false&rewriteBatchedStatements=true
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),6,1))>1,0,1))))nMsM) AND 'konk'='konk%';
SELECT id,name FROM userdata WHERE name LIKE '%test' AND (SELECT 7378 FROM (SELECT(SLEEP(1-(IF(ORD(MID((SELECT IFNULL(CAST(password AS NCHAR),0x20)
FROM common_mistakes.userdata ORDER BY id LIMIT 1,1),6,1))>1,0,1))))nMsM) AND 'konk'='konk%';
```

所以说，即使屏蔽错误信息错误码，也不能彻底防止 SQL 注入。真正的解决方式，还是使用参数化查询，让任何外部输入值只可能作为数据来处理。


比如，对于之前那个接口，在 SQL 语句中使用 “?” 作为参数占位符，然后提供参数值。这样修改后，sqlmap 也就无能为力了：

 复制代码

```
1 @PostMapping("jdbcright")
2 public void jdbcright(@RequestParam("name") String name) {
3     log.info("{} ", jdbcTemplate.queryForList("SELECT id,name FROM userdata WHERE ?"))
4 }
```


对于 MyBatis 来说，同样需要使用参数化的方式来写 SQL 语句。在 MyBatis 中，“#{ }”是参数化的方式，“\${ }”只是占位符替换。

比如 LIKE 语句。因为使用 “#{ }” 会为参数带上单引号，导致 LIKE 语法错误，所以一些同学会退而求其次，选择 “\${ }” 的方式，比如：

 复制代码

```
1 @Select("SELECT id,name FROM `userdata` WHERE name LIKE '${name}%'")
2 List<UserData> findByNameWrong(@Param("name") String name);
```

你可以尝试一下，使用 sqlmap 同样可以实现注入。正确的做法是，使用 “#{ }” 来参数化 name 参数，对于 LIKE 操作可以使用 CONCAT 函数来拼接 % 符号：

 复制代码

```
1 @Select("SELECT id,name FROM `userdata` WHERE name LIKE CONCAT('%',#{name},'%')")
2 List<UserData> findByNameRight(@Param("name") String name);
```


又比如 IN 子句。因为涉及多个元素的拼接，一些同学不知道如何处理，也可能会选择使用 “\${ }”。因为使用 “#{ }” 会把输入当做一个字符串来对待：

 复制代码

```
1 <select id="findByNameWrong" resultType="org.geekbang.time.commonmistakes.code">
2     SELECT id,name FROM `userdata` WHERE name in (${names})
3 </select>
```




但是，这样直接把外部传入的内容替换到 IN 内部，同样会有注入漏洞：

 复制代码

```
1 @PostMapping("mybatiswrong2")
2 public List mybatiswrong2(@RequestParam("names") String names) {
3     return userDataMapper.findByNameWrong(names);
4 }
```

你可以使用下面这条命令测试下：

 复制代码

```
1 python sqlmap.py -u http://localhost:45678/sqlinject/mybatiswrong2 --data nam
```

最后可以发现，有 4 种可行的注入方式，分别是布尔盲注、报错注入、时间盲注和联合查询注入：

```
Parameter: names (POST)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: names='test1','test2') AND 4391=4391 AND ('qAIz'='qAIz'

Type: error-based
Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: names='test1','test2') AND (SELECT 6669 FROM(SELECT COUNT(*),CONCAT(0x716a717871,(SELECT (ELT(6669=6669,1))) ,0x7178706b71,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a) AND ('Mnrw'='Mnrw'

Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: names='test1','test2') AND (SELECT 2875 FROM (SELECT(SLEEP(5)))WKuQ) AND ('BfZx'='BfZx'

Type: UNION query
Title: Generic UNION query (NULL) - 3 columns
Payload: names='test1','test2') UNION ALL SELECT CONCAT(0x716a717871,0x4761667779626849517150584654586a61674a58587a62704b4f57726b4e534c474b6a74465a4871,0x7178706b71),NULL-- --'
```

修改方式是，给 MyBatis 传入一个 List，然后使用其 foreach 标签来拼接出 IN 中的内容，并确保 IN 中的每一项都是使用 “#{ }” 来注入参数：

 复制代码

```
1 @PostMapping("mybatisright2")
2 public List mybatisright2(@RequestParam("names") List<String> names) {
3     return userDataMapper.findByNameRight(names);
4 }
5
6 <select id="findByNamesRight" resultType="org.geekbang.time.commonmistakes.codi
7     SELECT id,name FROM `userdata` WHERE name in
8     <foreach collection="names" item="item" open="(" separator="," close=")">
9         #{item}
10    </foreach>
11 </select>
```

修改后这个接口就不会被注入了，你可以自行测试一下。

## 小心动态执行代码时代码注入漏洞

总结下，我们刚刚看到的 SQL 注入漏洞的原因是，黑客把 SQL 攻击代码通过传参混入 SQL 语句中执行。同样，对于任何解释执行的其他语言代码，也可以产生类似的注入漏洞。我们看一个动态执行 JavaScript 代码导致注入漏洞的案例。

现在，我们要对用户名实现动态的规则判断：通过 ScriptEngineManager 获得一个 JavaScript 脚本引擎，使用 Java 代码来动态执行 JavaScript 代码，实现当外部传入的用户名为 admin 的时候返回 1，否则返回 0：

 复制代码

```
1 private ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
2 //获得JavaScript脚本引擎
3 private ScriptEngine jsEngine = scriptEngineManager.getEngineByName("js");
4
5 @GetMapping("wrong")
6 public Object wrong(@RequestParam("name") String name) {
7     try {
8         //通过eval动态执行JavaScript脚本，这里name参数通过字符串拼接方式混入JavaScript
9         return jsEngine.eval(String.format("var name='%s'; name=='admin'?1:0;"));
10    } catch (ScriptException e) {
11        e.printStackTrace();
12    }
13    return null;
14 }
```

这个功能本身没什么问题：

← → ↻ 🏠 ⓘ localhost:45678/codeinject/wrong?name=admin

1

但是，如果我们把传入的用户名修改为这样：

```
1 haha';java.lang.System.exit(0);'
```

就可以达到关闭整个程序的目的。原因是，我们直接把代码和数据拼接在了一起。外部如果构造了一个特殊的用户名先闭合字符串的单引号，再执行一条 `System.exit` 命令的话，可以满足脚本不出错，命令被执行。

解决这个问题有两种方式。

第一种方式和解决 SQL 注入一样，需要把外部传入的条件数据仅仅当做数据来对待。我们可以通过 `SimpleBindings` 来绑定参数初始化 `name` 变量，而不是直接拼接代码：

```
1 @GetMapping("right")
2 public Object right(@RequestParam("name") String name) {
3     try {
4         //外部传入的参数
5         Map<String, Object> parm = new HashMap<>();
6         parm.put("name", name);
7         //name参数作为绑定传给eval方法，而不是拼接JavaScript代码
8         return jsEngine.eval("name=='admin'?1:0;", new SimpleBindings(parm));
9     } catch (ScriptException e) {
10         e.printStackTrace();
11     }
12     return null;
13 }
```

这样就避免了注入问题：

← → ↺ ⬆ ⓘ localhost:45678/codeinject/right?name=haha%27;java.lang.System.exit(0);%27

0

第二种解决方法是，使用 `SecurityManager` 配合 `AccessControlContext`，来构建一个脚本运行的沙箱环境。脚本能执行的所有操作权限，是通过 `setPermissions` 方法精细化设置的：

```
1 @Slf4j
2 public class ScriptingSandbox {
3     private ScriptEngine scriptEngine;
4     private AccessControlContext accessControlContext;
5
6     private SecurityManager securityManager;
7     private static ThreadLocal<Boolean> needCheck = ThreadLocal.withInitial(()
8
9     public ScriptingSandbox(ScriptEngine scriptEngine) throws InstantiationExc
10         this.scriptEngine = scriptEngine;
11         securityManager = new SecurityManager(){
12             //仅在需要的时候检查权限
13             @Override
14             public void checkPermission(Permission perm) {
15                 if (needCheck.get() && accessControlContext != null) {
16                     super.checkPermission(perm, accessControlContext);
17                 }
18             }
19         };
20         //设置执行脚本需要的权限
21         setPermissions(Arrays.asList(
22             new RuntimePermission("getProtectionDomain"),
23             new PropertyPermission("jdk.internal.lambda.dumpProxyClasses",
24             new FilePermission(Shell.class.getProtectionDomain().getPermis
25             new RuntimePermission("createClassLoader"),
26             new RuntimePermission("accessClassInPackage.jdk.internal.org.ol
27             new RuntimePermission("accessClassInPackage.jdk.nashorn.intern
28             new RuntimePermission("accessDeclaredMembers"),
29             new ReflectPermission("suppressAccessChecks")
30         ));
31     }
32     //设置执行上下文的权限
33     public void setPermissions(List<Permission> permissionCollection) {
34         Permissions perms = new Permissions();
35
36         if (permissionCollection != null) {
37             for (Permission p : permissionCollection) {
38                 perms.add(p);
39             }
40         }
41
42         ProtectionDomain domain = new ProtectionDomain(new CodeSource(null, (C
43         accessControlContext = new AccessControlContext(new ProtectionDomain[]
44     }
45
46     public Object eval(final String code) {
47         SecurityManager oldSecurityManager = System.getSecurityManager();
48         System.setSecurityManager(securityManager);
49         needCheck.set(true);
50         try {
51             //在AccessController的保护下执行脚本
```

```

52         return AccessController.doPrivileged((PrivilegedAction<Object>) ()
53             try {
54                 return scriptEngine.eval(code);
55             } catch (ScriptException e) {
56                 e.printStackTrace();
57             }
58             return null;
59     }, accessControlContext);
60
61     } catch (Exception ex) {
62         log.error("抱歉, 无法执行脚本 {}", code, ex);
63     } finally {
64         needCheck.set(false);
65         System.setSecurityManager(oldSecurityManager);
66     }
67     return null;
68 }

```

写一段测试代码，使用刚才定义的 ScriptingSandbox 沙箱工具类来执行脚本：

 复制代码

```

1 @GetMapping("right2")
2 public Object right2(@RequestParam("name") String name) throws InstantiationException {
3     //使用沙箱执行脚本
4     ScriptingSandbox scriptingSandbox = new ScriptingSandbox(jsEngine);
5     return scriptingSandbox.eval(String.format("var name='%s'; name=='admin'?1"));
6 }

```

这次，我们再使用之前的注入脚本调用这个接口：

 复制代码

```

1 http://localhost:45678/codeinject/right2?name=haha%27;java.lang.System.exit(0)

```

可以看到，结果中抛出了 AccessControlException 异常，注入攻击失效了：

 复制代码

```

1 [13:09:36.080] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.c.codeinject.ScriptingSandbox]
2 java.security.AccessControlException: access denied ("java.lang.RuntimePermission"
3     at java.security.AccessControlContext.checkPermission(AccessControlContext.java:437)
4     at java.lang.SecurityManager.checkPermission(SecurityManager.java:585)
5     at org.geekbang.time.commonmistakes.codeanddata.codeinject.ScriptingSandbox$.eval(ScriptingSandbox.scala:10)
6     at java.lang.SecurityManager.checkExit(SecurityManager.java:761)

```


在实际应用中，我们可以考虑同时使用这两种方法，确保代码执行的安全性。

## XSS 必须全方位严防死堵

对于业务开发来说，XSS 的问题同样要引起关注。

XSS 问题的根源在于，原本是让用户传入或输入正常数据的地方，被黑客替换为了 JavaScript 脚本，页面没有经过转义直接显示了这个数据，然后脚本就被执行了。更严重的是，脚本没有经过转义就保存到了数据库中，随后页面加载数据的时候，数据中混入的脚本又当做代码执行了。黑客可以利用这个漏洞来盗取敏感数据，诱骗用户访问钓鱼网站等。

我们写一段代码测试下。首先，服务端定义两个接口，其中 index 接口查询用户名信息返回给 xss 页面，save 接口使用 @RequestParam 注解接收用户名，并创建用户保存到数据库；然后，重定向浏览器到 index 接口：

 复制代码

```
1 @RequestMapping("xss")
2 @Slf4j
3 @Controller
4 public class XssController {
5     @Autowired
6     private UserRepository userRepository;
7     //显示xss页面
8     @GetMapping
9     public String index(ModelMap modelMap) {
10         //查数据库
11         User user = userRepository.findById(1L).orElse(new User());
12         //给View提供Model
13         modelMap.addAttribute("username", user.getName());
14         return "xss";
15     }
16     //保存用户信息
17     @PostMapping
18     public String save(@RequestParam("username") String username, HttpServletRequest request) {
19         User user = new User();
20         user.setId(1L);
21         user.setName(username);
22         userRepository.save(user);
23         //保存完成后重定向到首页
24         return "redirect:/xss/";
25     }
26 }
```



```

27 }
28 //用户类，同时作为DTO和Entity
29 @Entity
30 @Data
31 public class User {
32     @Id
33     private Long id;
34     private String name;

```

我们使用 Thymeleaf 模板引擎来渲染页面。模板代码比较简单，页面加载的时候会在标签显示用户名，用户输入用户名提交后调用 save 接口创建用户：

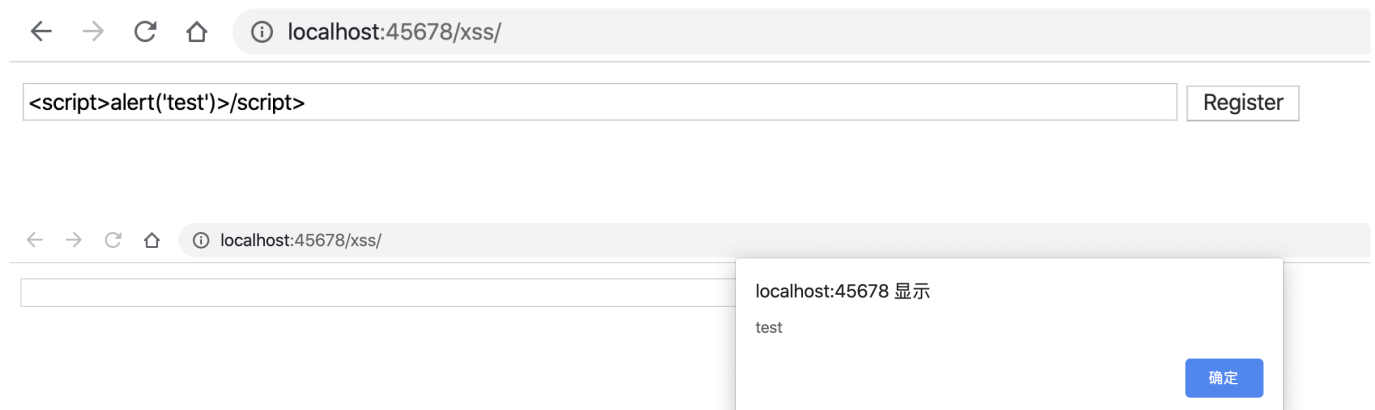
```

1 <div style="font-size: 14px">
2     <form id="myForm" method="post" th:action="@{/xss/}">
3         <label th:utext="${username}"/>
4         <input id="username" name="username" size="100" type="text"/>
5         <button th:text="Register" type="submit"/>
6     </form>
7 </div>

```

复制代码

打开 xss 页面后，在文本框中输入 `<script>alert( 'test' )</script>` 点击 Register 按钮提交，页面会弹出 alert 对话框：



并且，脚本被保存到了数据库：

	id	name
	1	<script>alert('test')</script>

你可能想到了，解决方式就是 HTML 转码。既然是通过 @RequestParam 来获取请求参数，那我们定义一个 @InitBinder 实现数据绑定的时候，对字符串进行转码即可：

 复制代码

```
1 @ControllerAdvice
2 public class SecurityAdvice {
3     @InitBinder
4     protected void initBinder(WebDataBinder binder) {
5         //注册自定义的绑定器
6         binder.registerCustomEditor(String.class, new PropertyEditorSupport() {
7             @Override
8             public String getAsText() {
9                 Object value = getValue();
10                return value != null ? value.toString() : "";
11            }
12            @Override
13            public void setAsText(String text) {
14                //赋值时进行HTML转义
15                setValue(text == null ? null : HtmlUtils.htmlEscape(text));
16            }
17        });
18    }
19 }
```

的确，针对这个场景，这种做法是可行的。数据库中保存了转义后的数据，因此数据会被当做 HTML 显示在页面上，而不是当做脚本执行：

	id	name
	1	&lt;script&gt;alert(&#39;test&#39;)&lt;/script&gt;

← → ↺ ⌂ ⓘ localhost:45678/xss/

<script>alert('test')</script>

Register

但是，这种处理方式犯了一个严重的错误，那就是没有从根儿上来处理安全问题。因为 @InitBinder 是 Spring Web 层面的处理逻辑，如果有代码不通过 @RequestParam 来获取数据，而是直接从 HTTP 请求获取数据的话，这种方式就不会奏效。比如这样：

 复制代码

```
1 user.setName(request.getParameter("username"));
```

更合理的解决方式是，定义一个 servlet Filter，通过 HttpServletRequestWrapper 实现 servlet 层面的统一参数替换：

 复制代码

```
1 //自定义过滤器
2 @Component
3 @Order(Ordered.HIGHEST_PRECEDENCE)
4 public class XssFilter implements Filter {
5     @Override
6     public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws ServletException, IOException {
7         chain.doFilter(new XssRequestWrapper((HttpServletRequest) request), response);
8     }
9 }
10 public class XssRequestWrapper extends HttpServletRequestWrapper {
11
12     public XssRequestWrapper(HttpServletRequest request) {
13         super(request);
14     }
15
16     @Override
17     public String[] getParameterValues(String parameter) {
18         //获取多个参数值的时候对所有参数值应用clean方法逐一清洁
19         return Arrays.stream(super.getParameterValues(parameter)).map(this::clean).toArray(String[]::new);
20     }
21
22     @Override
23     public String getHeader(String name) {
24         //同样清洁请求头
25         return clean(super.getHeader(name));
26     }
27
28     @Override
29     public String getParameter(String parameter) {
30         //获取参数单一值也要处理
31         return clean(super.getParameter(parameter));
32     }
33     //clean方法就是对值进行HTML转义
34     private String clean(String value) {
35         return value != null ? value.replaceAll("<|>|"|'|\\" data-bbox="93 344 910 948"/>
```

```
35         return StringUtils.isEmpty(value)? "" : HtmlUtils.htmlEscape(value);
36     }
37 }
```

这样，我们就可以实现所有请求参数的 HTML 转义了。不过，这种方式还是不够彻底，原因是无法处理通过 @RequestBody 注解提交的 JSON 数据。比如，有这样一个 PUT 接口，直接保存了客户端传入的 JSON User 对象：


 复制代码

```
1 @PutMapping
2 public void put(@RequestBody User user) {
3     userRepository.save(user);
4 }
```

通过 Postman 请求这个接口，保存到数据库中的数据还是没有转义：



我们需要自定义一个 Jackson 反序列化器，来实现反序列化时的字符串的 HTML 转义：

 复制代码

```
1 //注册自定义的Jackson反序列化器
2 @Bean
3 public Module xssModule() {
4     SimpleModule module = new SimpleModule();
5     module.addDeserializer(String.class, new XssJsonDeserializer());
6     return module;
7 }
8
9 public class XssJsonDeserializer extends JsonSerializer<String> {
10     @Override
11     public String deserialize(JsonParser jsonParser, DeserializationContext ct:
12         String value = jsonParser.getValueAsString();
13         if (value != null) {
```

```
14         //对于值进行HTML转义
15         return HtmlUtils.htmlEscape(value);
16     }
17     return value;
18 }
19
20 @Override
21 public Class<String> handledType() {
22     return String.class;
23 }
24 }
```

这样就实现了既能转义 Get/Post 通过请求参数提交的数据，又能转义请求体中直接提交的 JSON 数据。


你可能觉得做到这里，我们的防范已经很全面了，但其实不是。这种只能堵新漏，确保新数据进入数据库之前转义。如果因为之前的漏洞，数据库中已经保存了一些 JavaScript 代码，那么读取的时候同样可能出问题。因此，我们还要实现数据读取的时候也转义。

接下来，我们看一下具体的实现方式。

首先，之前我们处理了 JSON 反序列化问题，那么就需要同样处理序列化，实现数据从数据库中读取的时候转义，否则读出来的 JSON 可能包含 JavaScript 代码。

比如，我们定义这样一个 GET 接口以 JSON 来返回用户信息：

```
1 @GetMapping("user")
2 @ResponseBody
3 public User query() {
4     return userRepository.findById(1L).orElse(new User());
5 }
```

 复制代码

GET

▼

http://localhost:45678/xss/user

Params

Authorization

Headers (9)

Body ●

Pre-request Script

Tests

Settings

Query Params

	KEY	VALUE
	Key	Value

Body

Cookies (1)

Headers (3)

Test Results

Pretty

Raw

Preview

Visualize BETA

JSON ▼

1

{

2

"id": 1,

3

"name": "<script>alert('test')</script>"

4

}

修改之前的 SimpleModule 加入自定义序列化器，并且实现序列化时处理字符串转义：

复制代码

```
1 //注册自定义的Jackson序列器
2 @Bean
3 public Module xssModule() {
4     SimpleModule module = new SimpleModule();
5     module.addDeserializer(String.class, new XssJsonDeserializer());
6     module.addSerializer(String.class, new XssJsonSerializer());
7     return module;
8 }
9
10 public class XssJsonSerializer extends JsonSerializer<String> {
11     @Override
12     public Class<String> handledType() {
13         return String.class;
14     }
15
16     @Override
17     public void serialize(String value, JsonGenerator jsonGenerator, Serialize
18         if (value != null) {
19             //对字符串进行HTML转义
20             jsonGenerator.writeString(HtmlUtils.htmlEscape(value));
21         }
22     }
23 }
```



可以看到，这次读到的 JSON 也转义了：

GET

http://localhost:45678/xss/user

Params

Authorization

Headers (9)

Body ●

Pre-request Script

Tests

Settings

Query Params

KEY	VALUE
Key	Value

Body

Cookies (1)

Headers (3)

Test Results

Pretty

Raw

Preview

Visualize BETA

JSON

1

{

2

"id": 1,

3

"name": "&lt;script&gt;alert(&#39;test&#39;)&lt;/script&gt;"

4

}

其次，我们还需要处理 HTML 模板。对于 Thymeleaf 模板引擎，需要注意的是，使用 `th:utext` 来显示数据是不会进行转义的，需要使用 `th:text`：

1

<label th:text="\$\${username}"/>

复制代码

经过修改后，即使数据库中已经保存了 JavaScript 代码，呈现的时候也只能作为 HTML 显示了。现在，对于进和出两个方向，我们都实现了补漏。

但，所谓百密总有一疏。为了避免疏漏，进一步控制 XSS 可能带来的危害，我们还要考虑一种情况：如果需要在 Cookie 中写入敏感信息的话，我们可以开启 `HttpOnly` 属性。这样 JavaScript 代码就无法读取 Cookie 了，即便页面被 XSS 注入了攻击代码，也无法获得我们的 Cookie。

写段代码测试一下。定义两个接口，其中 `readCookie` 接口读取 Key 为 `test` 的 Cookie，`writeCookie` 接口写入 Cookie，根据参数 `HttpOnly` 确定 Cookie 是否开启 `HttpOnly`：

```

1 //服务端读取Cookie
2 @GetMapping("readCookie")
3 @ResponseBody
4 public String readCookie(@CookieValue("test") String cookieValue) {
5     return cookieValue;
6 }
7 //服务端写入Cookie
8 @GetMapping("writeCookie")
9 @ResponseBody
10 public void writeCookie(@RequestParam("httpOnly") boolean httpOnly, HttpServletRequest
11     Cookie cookie = new Cookie("test", "zhuye");
12     //根据httpOnly入参决定是否开启HttpOnly属性
13     cookie.setHttpOnly(httpOnly);
14     response.addCookie(cookie);
15 }

```

复制代码

可以看到，由于 test 和 \_ga 这两个 Cookie 不是 HttpOnly 的。通过 document.cookie 可以输出这两个 Cookie 的内容：

The screenshot shows a web browser's developer tools. The Network tab is active, and the 'Cookies' sub-tab is selected for the request 'writeCookie?httpOnly=false'. The 'Request Cookies' table lists the following cookies:

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite
JSESSIONID	6C7076B2653A48CB55576D75F9E5CD44	localhost	/	Session	42	✓		
_ga	GA1.1.907472498.1580877998	localhost	/	2022-02-...	29			
grafana_session	00b65787a441a24ca18c915b97704b53	localhost	/	2020-03-...	47	✓		Lax
test	zhuye	localhost	/xss	Session	9			

The 'test' cookie is highlighted with a red box. The 'Response Cookies' table shows the 'test' cookie:

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite
test	zhuye			Session	10			

The Console tab at the bottom shows the output of 'document.cookie' as '< \"test=zhuye; \_ga=GA1.1.907472498.1580877998\"'.

为 test 这个 Cookie 启用了 HttpOnly 属性后，就不能被 document.cookie 读取到了，输出中只有 \_ga 一项：

localhost:45678/xss/writeCookie?httpOnly=true

Network

writeCookie?httpOnly=false  
writeCookie?httpOnly=true

Request Cookies

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly
JSESSIONID	6C7076B2653A48CB55576D75F9E5CD44	localhost	/	Session	42	✓
_ga	GA1.1.907472498.1580877998	localhost	/	2022-02-...	29	
grafana_session	00b65787a441a24ca18c915b97704b53	localhost	/	2020-03-...	47	✓
test	zhuye	localhost	/xss	Session	9	✓

Response Cookies

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly
test	zhuye			Session	20	✓

2 requests 204 B transferred

Console

```
> document.cookie  
< "_ga=GA1.1.907472498.1580877998"
```

但是服务端可以读取到这个 cookie:

localhost:45678/xss/readCookie

zhuye

## 重点回顾

今天，我通过案例，和你具体分析了 SQL 注入和 XSS 攻击这两类注入类安全问题。

在学习 SQL 注入的时候，我们通过 sqlmap 工具看到了几种常用注入方式，这可能改变了我们对 SQL 注入威力的认知：对于 POST 请求、请求没有任何返回数据、请求不会出错的情况下，仍然可以完成注入，并可以导出数据库的所有数据。

对于 SQL 注入来说，避免参数化的查询是最好的堵漏方式；对于 JdbcTemplate 来说，我们可以使用 “?” 作为参数占位符；对于 MyBatis 来说，我们需要使用 “#{ }” 进行参数化

处理。

和 SQL 注入类似的是，脚本引擎动态执行代码，需要确保外部传入的数据只能作为数据来处理，不能和代码拼接在一起，只能作为参数来处理。代码和数据之间需要划出清晰的界限，否则可能产生代码注入问题。同时，我们可以通过设置一个代码的执行沙箱来细化代码的权限，这样即便产生了注入问题，因为权限受限注入攻击也很难发挥威力。

**随后通过学习 XSS 案例，我们认识到处理安全问题需要确保三点。**

第一，要从根本上、从最底层进行堵漏，尽量不要在高层框架层面做，否则堵漏可能不彻底。

第二，堵漏要同时考虑进和出，不仅要确保数据存入数据库的时候进行了转义或过滤，还要在取出数据呈现的时候再次转义，确保万无一失。

第三，除了直接堵漏外，我们还可以通过一些额外的手段限制漏洞的威力。比如，为 Cookie 设置 HttpOnly 属性，来防止数据被脚本读取；又比如，尽可能限制字段的最大保存长度，即使出现漏洞，也会因为长度问题限制黑客构造复杂攻击脚本的能力。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

## 思考与讨论

1. 在讨论 SQL 注入案例时，最后那次测试我们看到 sqlmap 返回了 4 种注入方式。其中，布尔盲注、时间盲注和报错注入，我都介绍过了。你知道联合查询注入，是什么吗？
2. 在讨论 XSS 的时候，对于 Thymeleaf 模板引擎，我们知道如何让文本进行 HTML 转义显示。FreeMarker 也是 Java 中很常用的模板引擎，你知道如何处理转义吗？

你还遇到过其他类型的注入问题吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

# 6月-7月课表抢先看

## 充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 28 | 安全兜底: 涉及钱时, 必须考虑防刷、限量和防重

下一篇 30 | 如何正确保存和传输敏感数据?

### 精选留言 (2)

写留言



Summer 空城

2020-05-23

老师, 您好, 请教一个问题, 微服务中一个模块跟多个模块rpc交互的时候, 参数比较多的情况下是把其他模块的pojo复制过来, 还是提供一个jar存放多个模块交互的pojo, 供多个模块引用么? 这两种方式感觉都不太好, 老师您遇到这种问题是怎样处理的呢, 麻烦老师指点下, 谢谢老师o(^o^o)

展开

作者回复: 一般而言, 不会使用你提到的两种方式

把其他模块的pojo复制过来, 复制粘贴显然不是推荐的方式; 提供一个jar存放多个模块交互的pojo, 不建议把多个模块交互的pojo混在一个jar中。

一般有2种做法：

1、如果技术栈统一，那么可以把微服务拆为API + 实现两个jar（模块），前者包含接口 + POJO，后者的服务类实现前者中的接口，服务的开发者把前者打包为API包提供给使用者即可，比如Spring Cloud的Feign可以接口可以共享出去为微服务API包加入私服

2、如果技术栈不统一，比如不是所有的使用者都使用Spring Cloud，那么或许提供API包不是最好的方式，这个时候可以升级为提供SDK，SDK内部既包含API定义也包含Client调用Server的实现（比如通过HttpClient来），这样微服务使用者不用局限于使用微服务服务端相同的协议



2



**13963865700**

2020-05-23

对于xss攻击防范，ESAPI的建议是在前端根据变量所处的位置（html、js）采用不同的编码方式进行转义，前端的开发成本较高；文中提到的方式是在后端采用过滤器统一转义存储，可以一劳永逸。老师在实际生产项目中采用的是哪种方式，转义存储对输入内容进行了修改，会不会产生什么副作用？

展开 ∨

作者回复: 要从业务角度看输入的内容是不是没有可能有脚本或特殊符号，如果是这样的话转义没有什么问题的，实际项目中就是采用文中说的所有方式组合起来

