

07 | SQL注入：明明设置了强密码，为什么还会被别人登录？

2019-12-23 何为舟

安全攻防技能30讲

[进入课程 >](#)



讲述：何为舟

时长 15:14 大小 13.96M



你好，我是何为舟。

在上一讲中，我们介绍了 XSS 攻击。今天，我们来介绍另外一种常见的 Web 攻击：SQL 注入。

在讲正文之前，让我们先来看一个案例。某天，当你在查看应用的管理后台时，发现有很多异常的操作。接着，你很快反应过来了，这应该是黑客成功登录了管理员账户。于是，你立刻找到管理员，责问他是不是设置了弱密码。管理员很无辜地表示，自己的密码非常复杂，不可能泄漏，但是为了安全起见，他还是立即修改了当前的密码。奇怪的是，第二天，黑客还是能够继续登录管理员账号。问题来了，黑客究竟是怎么做到的呢？你觉得这里面的问题究竟出在哪里呢？你可以先自己思考一下，然后跟着我开始今天的学习！

SQL 注入攻击是如何产生的？

在上一讲中，我们讲了，XSS 是黑客通过篡改 HTML 代码，来插入并执行恶意脚本的一种攻击。其实，SQL 注入和 XSS 攻击很类似，都是黑客通过篡改代码逻辑发起的攻击。那么，不同的点是什么？SQL 注入到底是什么呢？


通常来说，我们会将应用的用户信息存储在数据库中。每次用户登录时，都会执行一个相应的 SQL 语句。这时，黑客会通过构造一些恶意的输入参数，在应用拼接 SQL 语句的时候，去篡改正常的 SQL 语意，从而执行黑客所控制的 SQL 查询功能。这个过程，就相当于黑客“注入”了一段 SQL 代码到应用中。这就是我们常说的 **SQL 注入**。

这么说可能还是有点理论，不够具体。接下来，我就以几个简单而又经典的示例，来给你介绍两种主要的 SQL 注入方式。

1. 修改 WHERE 语句


我们先来看一个例子。现在有一个简单的登录页面，需要用户输入 Username 和 Password 这两个变量来完成登录。具体的 Web 后台代码如下所示：

```
1 uName = getRequestString("username");
2 uPass = getRequestString("password");
3
4 sql = 'SELECT * FROM Users WHERE Username = "' + uName + '" AND Password = "' + uPass + '"'
```

 复制代码

当用户提交一个表单（假设 Username 为 admin，Password 为 123456）时，Web 将执行下面这行代码：

```
1 SELECT * FROM Users WHERE Username ="admin" AND Password ="123456"
```

 复制代码

用户名密码如果正确的话，这句 SQL 就能够返回对应的用户信息；如果错误的话，不会返回任何信息。因此，只要返回的行数 ≥ 1 ，就说明验证通过，用户可以成功登录。

所以，当用户正常地输入自己的用户名和密码时，自然就可以成功登录应用。那黑客想要在不知道密码的情况下登录应用，他又会输入什么呢？他会输入 " or ""="。这时，应用的数据库就会执行下面这行代码：

 复制代码

```
1 SELECT * FROM Users WHERE Username ="" AND Password ="" or ""=""
```

我们可以看到，WHERE 语句后面的判断是通过 or 进行拼接的，其中""="的结果是 true。那么，当有一个 or 是 true 的时候，最终结果就一定是 true 了。因此，这个 WHERE 语句是恒为真的，所以，数据库将返回全部的数据。

这样一来，我们就能解答文章开头的问题了，也就是说，黑客只需要在登录页面中输入 " or ""="，就可以在不知道密码的情况下，成功登录后台了。而这，也就是所谓的“万能密码”。而这个“万能密码”，其实就是通过修改 WHERE 语句，改变数据库的返回结果，实现无密码登录。

2. 执行任意语句

除此之外，大部分的数据库都支持多语句执行。因此，黑客除了修改原本的 WHERE 语句之外，也可以在原语句的后面，插入额外的 SQL 语句，来实现任意的增删改查操作。在实际工作中，MySQL 是最常用的数据库，我们就以它为例，来介绍一下，任意语句是如何执行的。

在 MySQL 中，实现任意语句执行最简单的方法，就是利用分号将原本的 SQL 语句进行分割。这样，我们就可以一次执行多个语句了。比如，下面这个语句在执行的时候会先插入一个行，然后再返回 Users 表中全部的数据。

 复制代码

```
1 INSERT INTO Users (Username, Password) VALUES("test","000000"); SELECT * FROM I
```

接下来，我们来看一个具体的例子。在用户完成登录后，应用通常会通过 userId 来获取对应的用户信息。其 Web 后台的代码如下所示：

```
1 uid = getRequestString("userId");
2 sql = "SELECT * FROM Users WHERE UserId = " + uid;
```

[复制代码](#)

在这种情况下，黑客只要在传入的 `userId` 参数中加入一个分号，就可以执行任意的 SQL 语句了。比如，黑客想“删库跑路”的话，就令 `userId` 为 `1;DROP TABLE Users`，那么，后台实际执行的 SQL 就会变成下面这行代码，而数据库中所有的用户信息就都会被删除。

```
1 SELECT * FROM Users WHERE UserId = 1; DROP TABLE Users
```

[复制代码](#)

SQL 注入的“姿势”还有很多（比如：[没有回显的盲注](#)、[基于 INSERT 语句的注入](#)等等），它们的原理都是一样的，都是通过更改 SQL 的语义来执行黑客设定的 SQL 语句。如果你有兴趣，可以通过我前面给出的链接去进一步了解。

通过 SQL 注入攻击，黑客能做什么？

通过上面对 SQL 注入的简单介绍，我们已经知道，SQL 注入会令 Web 后台执行非常规的 SQL 语句，从而导致各种各样的问题。那么通过 SQL 注入攻击，黑客究竟能够干些什么呢？下面我们就——来看。

1. 绕过验证

在上面的内容中，我们已经介绍过，“`or ""=""`”作为万能密码，可以让黑客在不知道密码的情况下，通过登录认证。因此，SQL 注入最直接的利用方式，就是绕过验证，也就相当于身份认证被破解了。


2. 任意篡改数据

除了绕过验证，我们在任意语句执行的部分中讲到，SQL 注入漏洞导致黑客可以执行任意的 SQL 语句。因此，通过插入 DML 类的 SQL 语句（INSERT、UPDATE、DELETE、TRUNCATE、DROP 等），黑客就可以对表数据甚至表结构进行更改，这样数据的完整性就会受到损害。比如上面例子中，黑客通过插入 `DROP TABLE Users`，删除数据库中全部的用户。

3. 窃取数据

在 XSS 漏洞中，黑客可以通过窃取 Cookie 和“钓鱼”获得用户的隐私数据。那么，在 SQL 注入中，黑客会怎么来获取这些隐私数据呢？

在各类安全事件中，我们经常听到“拖库”这个词。所谓“拖库”，就是指黑客通过类似 SQL 注入的手段，获取到数据库中的全部数据（如用户名、密码、手机号等隐私数据）。最简单的，黑客利用 UNION 关键词，将 SQL 语句拼接成下面这行代码之后，就可以直接获取全部的用户信息了。

 复制代码

```
1 SELECT * FROM Users WHERE UserId = 1 UNION SELECT * FROM Users
```

4. 消耗资源

通过 [第 1 讲](#) 对 CIA 三元组的学习，我们知道，除了获取数据之外，影响服务可用性也是黑客的目标之一。

SQL 注入破坏可用性十分简单，可以通过完全消耗服务器的资源来实现。比如，在 Web 后台中，黑客可以利用 WHILE 打造死循环操作，或者定义存储过程，触发一个无限迭代等等。在这些情况下，数据库服务器因为 CPU 被迅速打满，持续 100%，而无法及时响应其他请求。

总结来说，通过 SQL 注入攻击，黑客可以绕过验证登录后台，非法篡改数据库中的数据；还能执行任意的 SQL 语句，盗取用户的隐私数据影响公司业务等等。所以，我认为，SQL 注入相当于让黑客直接和服务端的数据库进行了交互。正如我们一直所说的，应用的本质是数据，黑客控制了数据库，也就相当于控制了整个应用。

如何进行 SQL 注入防护？

在认识到 SQL 注入的危害之后，我们知道，一个简单的 SQL 查询逻辑，能够带来巨大的安全隐患。因此，我们应该做到在开发过程中就避免出现 SQL 注入漏洞。那具体应该怎么做呢？接下来，我会为你介绍 3 种常见的防护方法，它们分别是：使用 PreparedStatement、使用存储过程和验证输入。接下来，我们——来看。

1. 使用 PreparedStatement

通过**合理地**使用 PreparedStatement，我们就能够避免 99.99% 的 SQL 注入问题。你肯定很好奇，我为什么会这么说。接下来，让我们一起看一下它的实现过程。

当数据库在处理一个 SQL 命令的时候，大致可以分为两个步骤：

将 SQL 语句解析成数据库可使用的指令集。我们在使用 EXPLAIN 关键字分析 SQL 语句，就是干的这个事情；

将变量代入指令集，开始实际执行。之所以在批量处理 SQL 的时候能够提升性能，就是因为这样做避免了重复解析 SQL 的过程。

那么 PreparedStatement 为什么能够避免 SQL 注入的问题呢？

这是因为，SQL 注入是在解析的过程中生效的，用户的输入会影响 SQL 解析的结果。因此，我们可以通过使用 PreparedStatement，将 SQL 语句的解析和实际执行过程分开，只在执行的过程中代入用户的操作。这样一来，无论黑客提交的参数怎么变化，数据库都不会去执行额外的逻辑，也就避免了 SQL 注入的发生。

在 Java 中，我们可以通过执行下面的代码将解析和执行分开：


```
1 String sql = "SELECT * FROM Users WHERE UserId = ?";
2 PreparedStatement statement = connection.prepareStatement(sql);
3 statement.setInt(1, userId);
4 ResultSet results = statement.executeQuery();
5
```

 复制代码

为了实现相似的效果，在 PHP 中，我们可以使用 PDO (PHP Data Objects)；在 C# 中，我们可以使用 OleDbCommand 等等。

这里有一点需要你注意，前面我们说了，通过合理地使用 PreparedStatement 就能解决 99.99% 的 SQL 注入问题，那到底怎么做才算“合理地”使用呢？

PreparedStatement 为 SQL 语句的解析和执行提供了不同的“方法”，你需要分开来调用。但是，如果你在使用 PreparedStatement 的时候，还是通过字符串拼接来构造 SQL 语句，那仍然是将解析和执行放在了一块，也就不会产生相应的防护效果了。我这里给你展示了一个错误案例，你可以和上面的代码进行对比。


 复制代码

```
1 String sql = "SELECT * FROM Users WHERE UserId = " + userId;
2 PreparedStatement statement = connection.prepareStatement(sql);
3 ResultSet results = statement.executeQuery();
```

2. 使用存储过程

接下来，我们说一说，如何使用 [存储过程](#) 来防止 SQL 注入。实际上，它的原理和使用 PreparedStatement 类似，都是通过将 SQL 语句的解析和执行过程分开，来实现防护。区别在于，存储过程防注入是将解析 SQL 的过程，由数据库驱动转移到了数据库本身。

还是上述的例子，使用存储过程，我们可以这样来实现：

 复制代码

```
1 delimiter $$      # 将语句的结束符号从分号 ; 临时改为两个 $$ (可以是自定义)
2 CREATE PROCEDURE select_user(IN p_id INTEGER)
3 BEGIN
4     SELECT * FROM Users WHERE UserId = p_id;
5 END$$
6 delimiter;        # 将语句的结束符号恢复为分号
7
8 call select_user(1);
```

3. 验证输入

在上一节课中，我们讲过，**防护的核心原则是，一切用户输入皆不可信**。因此，SQL 注入的防护手段和 XSS 其实也是相通的，主要的不同在于：

SQL 注入的攻击发生在输入的时候，因此，我们只能在输入的时候去进行防护和验证；大部分数据库不提供针对 SQL 的编码，因为那会改变原有的语意，所以 SQL 注入没有编码的保护方案。

因此，对所有输入进行验证或者过滤操作，能够很大程度上避免 SQL 注入的出现。比如，在通过 `userId` 获取 Users 相关信息的示例中，我们可以确认 `userId` 必然是一个整数。因此，我们只需要对 `userId` 参数，进行一个整型转化（比如，Java 中的 `Integer.parseInt`，PHP 的 `intval`），就可以实现防护了。

当然，部分场景下，用户输入的参数会比较复杂。我们以用户发出的评论为例，其内容完全由用户定义，应用无法预判它的格式。这种情况下，应用只能通过对部分关键字符进行过滤，来避免 SQL 注入的发生。比如，在 MySQL 中，需要注意的关键词有 `" % ' \ _`。

这里我简单地总结一下，在实际使用这些防护方法时的注意点。对于验证输入来说，尤其是在复杂场景下的验证输入措施，其防护效果是最弱的。因此，避免 SQL 注入的防护方法，首要选择仍然是 `PreparedStatement` 或者存储过程。

总结

好了，这一节内容差不多了，下面我来带你总结回顾一下，你要掌握的重点内容。

SQL 注入就是黑客通过相关漏洞，篡改 SQL 语句的攻击。通过 SQL 注入，黑客既可以影响正常的 SQL 执行结果，从而绕过验证，也可以执行额外的 SQL 语句，对数据的机密性、完整性和可用性都产生影响。

为了避免 SQL 注入的出现，我们需要正确地使用 `PreparedStatement` 方法或者存储过程，尽量避免在 SQL 语句中出现字符串拼接的操作。除此之外，SQL 注入的防护也可以和 XSS 一样，对用户的输入进行验证、检测并过滤 SQL 中的关键词，从而避免原有语句被篡改。

今天的内容比较多，为了方便你记忆，我总结了一个知识脑图，你可以通过它来对今天的重要内容进行复习巩固。



思考题

好了，今天的内容差不多了，我们来看一道思考题。

假设有下面这样一个语句：

```
1 SELECT Username FROM Users WHERE UserId = 1
```

复制代码

你已经知道，WHERE 语句中存在着 SQL 注入的点。那么，我们怎么才能获取到除了 Username 之外的其他字段呢？这里我给你一个小提示，你可以先了解一下“[盲注](#)”这个概念，之后再思考这个问题。

欢迎留言和我分享你的思考和疑惑，也欢迎你把文章分享给你的朋友。我们下一讲再见！

点击查看 

来参加打卡，攻克 工作中 80% 的安全问题



PC端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 06 | XSS：当你“被发送”了一条微博时，到底发生了什么？


精选留言 (3)

 写留言



Cy23


2019-12-23

注册的时候看input的name都有什么，基本上就了解个大概有什么字段，然后尝试猜常用的字段，
或者查询显示的语句里注入查询表中所有字段名，然后替换字段名到显示输出的地方查看，
突然想起原来有个资源下载的网站，原来是免费的，后来收费了，有天无意将页面上下...
展开 



rocedu

2019-12-23

要是有个课题程配套虚拟机就更好了
展开 





ban

2019-12-23

现在应该很多大部分开发都使用 PreparedStatement 了吧

作者回复: 我也希望是这样的。

