链滴

sql注入分析

作者：guobing
原文链接：https://ld246.com/article/1470835857951
来源网站：链滴

##1. 什么是sql注入
> SQL注入攻击指的是通过**构建特殊的输入作为参数**传入Web应用程序，而这些输入大都是SQL法里的一些组合，通过让**原SQL改变了语义，达到欺骗服务器执行恶意的SQL命令**。其主要原因程序没有细致地过滤用户输入的数据，致使非法数据侵入系统。

##2. SQL注入实例

很多Web开发者没有意识到SQL查询是可以被篡改的，从而把SQL查询当作可信任的命令。殊不知，QL查询是可以绕开访问控制，从而绕过身份验证和权限检查的。更有甚者，有可能通过SQL查询去运主机系统级的命令。

下面将通过一些真实的例子来详细讲解SQL注入的方式。

考虑以下简单的登录表单：
```
<form action="/login" method="POST">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" value="登陆" /></p>
</form>
```
我们的处理里面的SQL可能是这样的：
```
username:=r.Form.Get("username")
password:=r.Form.Get("password")
sql:="SELECT * FROM user WHERE username='"+username+"' AND password='"+password+"
"
```
如果用户的输入的用户名如下，密码任意

`myuser' or 'foo' = 'foo' --`，这里的`--`很重要，相当于把后面的内容都注释掉了
那么我们的SQL变成了如下所示：
```
SELECT * FROM user WHERE username='myuser' or 'foo'=='foo' --'' AND password='xxx'
```
在SQL里面--是注释标记，所以查询语句会在此中断。这就让攻击者在不知道任何合法用户名和密码情况下成功登录了。

对于MSSQL还有更加危险的一种SQL注入，就是控制系统，下面这个可怕的例子将演示如何在某些版的MSSQL数据库上执行系统命令。
```
sql:="SELECT * FROM products WHERE name LIKE '%"+prod+"%'"
Db.Exec(sql)
```
如果攻击提交`a%' exec master..xp_cmdshell 'net user test testpass /ADD' --`作为变量 prod的值那么sql将会变成
```
sql:="SELECT * FROM products WHERE name LIKE '%a%' exec master..xp_cmdshell 'net user t
st testpass /ADD'--%'"
```
MSSQL服务器会执行这条SQL语句，包括它后面那个用于向系统添加新用户的命令。如果这个程序是sa运行而 MSSQLSERVER服务又有足够的权限的话，攻击者就可以获得一个系统帐号来访问主机了。

虽然以上的例子是针对某一特定的数据库系统的，但是这并不代表不能对其它数据库系统实施类似的

击。针对这种安全漏洞，只要使用不同方法，各种数据库都有可能遭殃。

再举个例子
后台需要根据前端传过来的参数查询用户信息。正常情况下我们要执行的sql语句是这样的
`SELECT uid,username FROM user WHERE username='plhwin'
`如果前端的参数传成这样`username=plhwin';SHOW TABLES-- hack `,我们执行sql的时候，sql变成
SELECT uid,username FROM user WHERE username='plhwin';SHOW TABLES-- hack'
`
,
注意:
> 在MySQL中，最后连续的两个减号表示忽略此SQL减号后面的语句，目前几乎所有SQL注入实例都直接采用两个减号结尾，但是实际测试，5.6.12版本的MySQL要求两个减号后面必须要有空格才能正注入，而浏览器是会自动删除掉URL尾部空格的，所以我们的注入会在两个减号后面统一添加任意一字符或单词，本篇文章的SQL注入实例统一以 -- hack 结尾。

经过上面的SQL注入后，原本想要执行查询会员详情的SQL语句，此时还额外执行了 SHOW TABLES; 语句，这显然不是开发者的本意
更可怕的事情还在后面，如果注入参数写成这样`plhwin';DROP TABLE user-- hack`,那你发现执行一查询操作后整个user表不见了。悲剧啊，重大事故。。所以数据库的权限一定要设置合理。很重要

另一个例子
```
select * from tb_usertable where name='"+u+"'and password='"+p+"'"
--如果我们把[' or '1' = '1]作为password传入进来.用户名随意,看看会成为什么?
 select * from tb_usertable where name='"+HEHE+"'and password='1'or'1'='1'
-- 这样构造成的SQL语句中因为'1'='1'肯定成立,所以可以任何通过验证
```
##3. 如何发现sql注入
1. 错误提示。所以提示信息不能太详细
2. 盲注

##4. 如何预防SQL注入
也许你会说攻击者要知道数据库结构的信息才能实施SQL注入攻击。确实如此，但没人能保证攻击者定拿不到这些信息，一旦他们拿到了，数据库就存在泄露的危险。如果你在用开放源代码的软件包来问数据库，比如论坛程序，攻击者就很容易得到相关的代码。如果这些代码设计不良的话，风险就更了。目前Discuz、phpwind、phpcms等这些流行的开源程序都有被SQL注入攻击的先例。

这些攻击总是发生在安全性不高的代码上。所以，永远不要信任外界输入的数据，特别是来自于用户数据，包括选择框、表单隐藏域和 cookie。就如上面的第一个例子那样，就算是正常的查询也有可造成灾难。

SQL注入攻击的危害这么大，那么该如何来防治呢?下面这些建议或许对防治SQL注入有一定的帮助。

> * 严格限制Web应用的数据库的`操作权限`，给此用户提供仅仅能够满足其工作的最低权限，从而大限度的减少注入攻击对数据库的危害。
>* 检查输入的数据是否具有所期望的`数据格式`，严格限制变量的类型，例如使用regexp包进行一匹配处理，或者使用strconv包对字符串转化成其他基本类型的数据进行判断。
对进入数据库的特殊字符（'"\尖括号&*;等）进行转义处理，或编码转换。Go 的text/template包里的HTMLEscapeString函数可以对字符串进行转义处理。
>* 所有的查询语句建议使用数据库提供的`参数化查询接口`，参数化的语句使用参数而不是将用户输变量嵌入到SQL语句中，即不要直接拼接SQL语句。例如使用database/sql里面的查询函数Prepare Query，或者Exec(query string, args ...interface{})。
>* 在应用发布之前建议使用专业的`SQL注入检测工具`进行检测，以及时修补被发现的SQL注入漏洞网上有很多这方面的开源工具，例如sqlmap、SQLninja等。
>* `避免网站打印出SQL错误信息`，比如类型错误、字段不匹配等，把代码里的SQL语句暴露出来，

防止攻击者利用这些错误信息进行SQL注入。
>* sql预编译

##5. 我们项目中是怎么防止sql注入的
首页我们用的是Groovy语言，Groovy提供了很好的sql操作，在`Groovy.sql.Sql`下面。就一般sql注最多的查询类来说，Groovy sql,它提供一个方法`db.rows(sql, args)`,方法底层是这么实现的
```
public List<GroovyRowResult> rows(String sql, List<Object> params, int offset, int maxRows, Closure metaClosure) throws SQLException {
        Sql.AbstractQueryCommand command = this.createPreparedQueryCommand(sql, param);
        command.setMaxRows(offset + maxRows);
        List var7;
        try {
            var7 = this.asList(sql, command.execute(), offset, maxRows, metaClosure);
        } finally {
            command.closeResources();
        }
        return var7;
    }
```
看到了一个很重要的方法`createPreparedQueryCommand`,这和preparedStatement是类似的。执了预编译。我们自己写sql也是参数用问好代替，支持预编译，例如：
`select * from ih_answer_reply where openid = ? and status = ? and type = ? `
所以，我们是通过预编译的方式来避免的。
> 那么`PreparedStatement`是怎么避免sql注入的呢?
 之所以PreparedStatement能防止注入，是因为它把单引号转义了，变成了\'，这样一来，就无法截SQL语句，进而无法拼接SQL语句，基本上没有办法注入了。

但是利用预编译的方式能解决所有的sql注入吗，答案是不一定的。看下面。
一个简单的sql`select * from goods where min_name like '儿童%'`正常情况下是没问题的，那如我的参数是`%儿童%`
整个意思就变成`select * from goods where min_name like '%儿童%%'`,这种情况下是不会转义的虽然此种SQL注入危害不大，但这种查询会耗尽系统资源，从而演化成拒绝服务攻击。

还有一种，我们自己写了一个方法,过滤掉了所有的特殊字符
```
static String transferSQLInjection(String str){
        str.replaceAll(".*([';]+|(--)+).*", " ")
}
```

##6. 总结
通过上面的示例我们可以知道，SQL注入是危害相当大的安全漏洞。所以对于我们平常编写的Web应，应该对于每一个小细节都要非常重视，细节决定命运，生活如此，编写Web应用也是这样。