**修订记录Revision record**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期  Date | 修订版本Revision version | 修改描述  change Description | 作者  Author | 审核 |
| 2016-04-20 | 0.1 | 初稿完成 | s90006809 |  |
| 2016-40-26 | 0.2 | 添加一个漏洞案例；按评审专家的修改意见进行修改。 | s90006809 |  |
| 2016-40-28 | 1.0 | 补充工具部分 | 安全工具组 |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 背景

经对近两年ICSL发现的web类问题统计，SQL注入问题分别为2014年23个（占比4%）、2015年61个（占比8%），从统计数据看，SQL注入问题的数量及占比都呈上升趋势。另外2015年CESC也发现2个SQL注入问题。

对这86个SQL注入问题进行分析，发现SQL注入问题的产生原因主要包括：使用外部输入数据直接拼接SQL语句、对外部输入数据校验不充分、mybatis使用不当（本质还是使用sql拼接）。SQL语句拼接又分为：字段值拼接、表名/字段名拼接、in中的条件值拼接。从问题产生原因来看，这些SQL注入问题绝大部分是很容易识别的。产品在开发过程中，增加开发人员的SQL注入防范意识；测试阶段，使用有效的代码静态扫描工具对代码进行扫描，都可以有效消除该类问题。

本文档针对近两年ICSL、CESC测出的SQL注入漏洞进行分析，并给出解决方案、排查指导，建议各产品线对现有产品做排查，并对问题整改。

# 漏洞案例

本节将对几个典型的SQL注入漏洞进行分析。

## \*\*产品动态SQL拼接导致SQL注入漏洞技术分析报告

|  |  |  |  |
| --- | --- | --- | --- |
| \*\*产品动态SQL拼接导致SQL注入漏洞技术分析报告 | | | |
| **安全问题号** | DTS2014042700550 | **日期** | 2014-7-27 |
| **严重级别** | 严重 | | |
| **产品简介** | | | |
|  | | | |
| **问题所在模块功能、使用场景说明** | | | |
| 产品支持日程信息删除操作，执行删除操作时存在sql注入漏洞，可导致所有的用户的日程信息全部被删除。 | | | |
| **问题代码片段：**  如下代码存在SQL注入的漏洞：  http://dts.huawei.com/net/dts/fckeditor/download.ashx?Path=SBxcQn4A7qNWoSTyQQWKCerHIZuzctCcGbDY6Fs2Gsnu5FH1PZNOXoXoxqOOLbYE | | | |
| **漏洞验证流程** | | | |
| 创建两个联系人，并在联系人上创建两个不同的生日日期，这时会在日历中看到两个日期的生日日程。  C:\Users\s90006809\Desktop\20140427040222.jpg  使用drozer工具，发送：run app.broadcast.send --action com.huawei.account\_delete\_birthday --extra string delete\_account\_type "1'or'1'='1"，执行后所有的日程信息被删除。 | | | |
| **问题详细技术分析** | | | |
| **安全问题原因分析：**   1. 使用不可信息数据进行动态SQL语句拼接，不可信数据位进行任何校验，最终导致实际执行的数据库操作与预期不同。 2. 该案例中，实际预期的操作是根据执行的accountType进行日程信息清理，恶意用户通过对accountType信息进行特殊构造，比如 "1' or '1'='1" ，这样动态拼接SQL语句后，执行删除操作时，会忽略accountType这一限制条件，直接删除所有的日程信息。 | | | |
| **对业务的影响** | | | |
| 利用该漏洞，攻击这可清空所有用户的日程信息。 | | | |
| **解决方案** | | | |
| 1、执行删除操作时，采用参数化查询的方式  2、对不可信数据进行严格的校验，比如accountType的值只能为字母或数字，且限制其长度等。 | | | |

## \*\*产品不可信数据校验不严导致SQL注入漏洞技术分析报告

|  |  |  |  |
| --- | --- | --- | --- |
| \*\*产品不可信数据校验不严导致SQL注入漏洞技术分析报告 | | | |
| **安全问题号** | DTS2014071707490 | **日期** | 2014-7-17 |
| **严重级别** | 低 | | |
| **产品简介** | | | |
|  | | | |
| **问题所在模块功能、使用场景说明** | | | |
| **用户登录\*\*系统后，打开多媒体参数维护界面，点击Search按钮，服务器端会根据用户指定的查询条件进行查询，并按照orderStr参数对查询结果进行排序。orderStr参数直接动态拼接SQL语句，使用前对该信息已经进行过参数校验，但是校验不严格，仍然存在SQL注入漏洞。** | | | |
| **问题代码片段：**  如下代码存在SQL注入的漏洞： | | | |
| **漏洞验证流程** | | | |
| 服务端正确执行了注入语句，返回结果按PARAMNAME列排序，    数据库侧查询的排序结果可以佐证：    由于该漏洞点在order by子句中，需利用嵌套查询、通过布尔条件来逐步获取数据库中的信息，一次能获取的信息量较少。 | | | |
| **问题详细技术分析** | | | |
| **安全问题原因分析**   1. 进行动态SQL语句拼接前对外部输入进行了校验，但是由于检验规则存在漏洞，用户的恶意构造可绕过校验规则。 2. 该案例中，使用使用正则表达式对外部输入进行校验，不允许包含空格字符，如空格、制表符、换页符等，但是恶意用户构造时，可使用/\*\*/替代空格进行注入。 | | | |
| **对业务的影响** | | | |
| 利用该漏洞，会造成数据库中的数据泄露。该漏洞被利用的限制条件较多，利用难度较大。 | | | |
| **解决方案** | | | |
| 1、使用参数化查询；  2、使用白名单校验的方式对不可信数据进行校验，对有效输入内容进行严格限制。 | | | |

## \*\*产品MyBatis使用不当导致SQL注入漏洞技术分析报告

|  |  |  |  |
| --- | --- | --- | --- |
| \*\*产品MyBatis使用不当导致SQL注入漏洞技术分析报告 | | | |
| **安全问题号** | DTS2015061901453 | **日期** | 2015-6-19 |
| **严重级别** | 严重 | | |
| **产品简介** | | | |
|  | | | |
| **问题所在模块功能、使用场景说明** | | | |
| **\*\*系统丢MyBatis框架使用不当，对外部输入的参数使用$拼接，且外部输入为进行校验，导致存在大量SQL注入漏洞。** | | | |
| **问题代码片段：**  如下代码存在SQL注入的漏洞：  http://dts.huawei.com/net/dts/fckeditor/download.ashx?Path=OZMh6Mt9duuGYxoeG5G8KZOcvL26k12fWB4c3hoFhase9uf0rmMCYeFDm7WIO9tW | | | |
| **漏洞验证流程** | | | |
| 数据库表R\_VIP\_NETCELL\_NE\_L\_74内容C:\Users\s90006809\Desktop\20150619103613.jpg  以ENODEBNAME进行排序，即decode(1,1, ENODEBNAME, RRCSETUPREQCNT)  C:\Users\s90006809\Desktop\20150619103626.jpg  以RRCSETUPREQCNT进行排序，即decode(1,2, ENODEBNAME, RRCSETUPREQCNT)  C:\Users\s90006809\Desktop\20150619103636.jpg | | | |
| **问题详细技术分析** | | | |
| **安全问题分析：**   1. MyBatis 是apache 的一个开源项目，一个O/R Mapping 解决方案，底层通过对SQL语句进行预编译，在一定程度上可以规避SQL注入。 2. 产品使用MyBatis框架不当，对外部输入的数据使用$进行拼接，且未对外部输入进行校验，导致SQL注入漏洞。 3. MyBatis框架中，$指定参数，执行SQL操作等同于动态拼接SQL语句；使用#指定参数，执行SQL语句等同于使用参数化查询，可防止SQL注入问题 | | | |
| **对业务的影响** | | | |
| 利用该漏洞，数据库中的信息存在被泄露的风险。 | | | |
| **解决方案** | | | |
| 使用#指定参数 | | | |

## \*\*产品对参数化查询使用不当造成SQL注入漏洞技术分析报告

|  |  |  |  |
| --- | --- | --- | --- |
| \*\*产品对参数化查询使用不当造成SQL注入漏洞技术分析报告 | | | |
| **安全问题号** | DTS2015061900656 | **日期** | 2015-6-19 |
| **严重级别** | 严重 | | |
| **产品简介** | | | |
|  | | | |
| **问题所在模块功能、使用场景说明** | | | |
|  | | | |
| **问题代码片段：**  如下代码存在SQL注入的漏洞： | | | |
| **漏洞验证流程** | | | |
|  | | | |
| **问题详细技术分析** | | | |
| **安全问题原因分析：**   1. 函数queryInfo中使用了参数化查询执行查询造作，不会存在SQL注入问题。 2. 但在调用queryInfo函数前，使用外部输入参数动态拼接了SQL语句。 3. 这样的操作过程，参数化查询就失去了防SQL注入的功能。 | | | |
| **对业务的影响** | | | |
|  | | | |
| **解决方案** | | | |
| 禁止使用不可信数据进行SQL语句动态拼接。 | | | |

# 根因分析

## 使用不可信数据动态拼接SQL语句

代码中，执行数据库操作时，直接用外部输入等不可信数据动态拼接SQL语句，当攻击者输入恶意构造的数据，动态拼接的SQL语句会执行与预期不符的操作，最终会造成数据泄露、信息篡改、数据被恶意清除等后果。

不可信数据包括：**用户输入参数、cookie中的信息、请求头中的信息**等。

## 对外部输入数据校验不彻底

实际代码中，开发人员会使用黑名单的形式对外部数据进行校验，校验后的数据认为是可信数据，然后使用这样的数据动态拼接SQL语句。使用黑名单对外部输入数据进行校验，往往由于**考虑不周全**、**忽略不同DB之间的特殊性**等原因造成疏漏，恶意用户精心构造的数据可绕过校验规则。

## 数据库操作框架使用不当

常见的数据操作框架MyBaits、Hibernate等本身具有防SQL注入的功能，但如果使用不当，仍然会造成SQL注入问题。对框架使用不当本质还是动态拼接SQL语句。

比如MyBaits中使用$指定动态参数；Hibernate中直接拼接SQL语句，未使用:param的形式指定参数。

# 推荐解决方案

对于SQL注入的防护措施主要有使用参数化查询、参数校验、编码和使用数据库框架等。在实际项目中，如果未使用数据库框架，**推荐优先使用参数化查询**，对于不能使用参数化查询的可对参数进行严格校验，如果确实允许输入特殊字符，则需要对不可信数据进行编码。

## 对于条件中的参数 优选使用参数化查询

参数化查询（Parameterized Query 或 Parameterized Statement）是访问数据库时，在需要填入数值或数据的地方，使用参数 (Parameter) 来给值。在使用参数化查询的情况下，数据库服务器不会将参数的内容视为SQL指令的一部份来处理，而是在**数据库完成SQL指令的编译后，才套用参数运行**，因此就算参数中含有指令，也不会被数据库运行。

**但是在使用参数化查询之前，已经动态拼接SQL语句，则无法防止SQL注入漏洞。**

下面的代码是参数化查询的一个具体示例。

|  |
| --- |
| *PreparedStatement stmt =* ***null***  *ResultSet rs**=* ***null***  ***try***  *{*  *String userName = ctx.getAuthenticatedUserName(); //this is a constant*  *String itemName = request.getParameter("itemName");*  ***// ...Ensure that the length of userName and itemName is legitimate***  ***// ...***  ***String sqlString = "SELECT \* FROM t\_item WHERE owner=? AND itemName=?";***  ***stmt = connection.prepareStatement(sqlString);***  ***stmt.setString(1, userName);***  ***stmt.setString(2, itemName);***  *rs = stmt.executeQuery();*  *// ... result set handling*  *}*  ***catch*** *(SQLException se)*  *{*  *// ... logging and error handling*  *}* |

使用参数化查询，在SQL语句中使用占位符表示需在运行时确定的参数值。参数化查询使得SQL查询的语义逻辑被预先定义，而实际的查询参数值则等到程序运行时再确定。参数化查询使得数据库能够区分SQL语句中语义逻辑和数据参数，以确保用户输入无法改变预期的SQL查询语义逻辑。在Java中，可以使用java.sql.PreparedStatement来对数据库发起参数化查询。在上述的示例中，如果一个攻击者将itemName输入为name' OR 'a' = 'a，这个参数化查询将免受攻击，会查找一个itemName匹配name' OR 'a' = 'a这个字符串的条目。

使用参数化查询时应注意，如果正在调用的数据库功能时在存储过程或函数实现中使用了动态SQL拼接，仍然会存在SQL注入风险。下面介绍一下存储过程中的sql注入。

|  |
| --- |
| *// ... result set handling*  *CallableStatement = null*  *ResultSet results = null;*  ***try***  *{*  *String userName = ctx.getAuthenticatedUserName(); //this is a constant*  *String itemName = request.getParameter("itemName");*  *cs = connection.prepareCall("{call sp\_queryItem(?,?)}");*  *cs.setString(1, userName);*  *cs.setString(2, itemName);*  *results = cs.executeQuery();*  *// ... result set handling*  *}*  ***catch*** *(SQLException se)*  *{*  *// ... logging and error handling*  *}*  *//.SQL存储过程*  ***CREATE******PROCEDURE*** *sp\_queryItem*  *@userName* ***varchar****(50),*  *@itemName* ***varchar****(50)*  ***AS***  ***BEGIN***  ***DECLARE*** *@sql nvarchar(500);*  ***SET @sql = 'SELECT \* FROM t\_item***  ***WHERE owner = ''' + @userName + '''***  ***AND itemName = ''' + @itemName + '''';***  ***EXEC(@sql);***  ***END***  *GO* |

上面的额存储过程中，通过拼接参数值来构建查询字符串，和在应用程序代码中拼接参数一样，同样是有SQL注入风险的。存储过程的正确写法如下：

|  |
| --- |
| ***CREATE******PROCEDURE*** *sp\_queryItem*  *@userName* ***varchar****(50),*  *@itemName* ***varchar****(50)*  ***AS***  ***BEGIN***  ***SELECT*** *\** ***FROM*** *t\_item*  ***WHERE*** *userName = @userName*  ***AND*** *itemName = @itemName;*  ***END***  *GO* |

**参数化查询不适用的场景**包括：

(1) 动态指定表名称或字段名称；

(2) where条件中的in、like限制添加。

另外，对于**where中条件动态组合的场景**，开发人员可能会认为使用参数化查询比较复杂，直接动态拼接SQL语句。这样的场景往往更容易造成SQL注入，因为这种场景开发人员往往也不会对参数进行严格校验。如果产品中存在多种类似场景，推荐使用MyBatis框架，该框架对这种场景提供了完善的解决方案。

## 对于不适用参数化查询的场景推荐对不可信数据进行严格校验

数据校验是除了参数化查询外的另一种功能强大的防护措施。对于不可信数据进行严格的校验，可有效降低SQL注入漏洞的产生。数据校验可采用白名单校验和黑名单校验。白名单校验是“接受已知好的数据”，黑名单校验是“拒绝已知坏的数据”。黑名单校验是一种较弱的校验方式，因为潜在的不合法数据可能是一个不受约束的无限集合。进行不可行数据校验时，建议采用白名单的校验方式。

使用白名单校验时，可以考虑以下要点：

**数据类型**：比如数字类型、正负数等；

**数据大小**：字符串可判断长度，数字可判断其大小

**数据访问**：数字类型，判断该数据是否在其预期的范围内；

**数据内容**：比如手机号、邮箱地址等是否符合标准的格式

常用的校验方式是使用正则表达式。

对于前面提到的不适用参数化查询的两种场景，可使用参数校验的方式进行防护。

1. 动态指定表名称或字段名称

表名称或字段名称数量有限，对该类信息进行校验时，可以使用穷举所有可能值的白名单校验规则；另外也可以采取用户间接指定表名称、字段名称的方式，这样可以不用对表名、字段名进行校验。

1. where中in、like中条件

该场景下，需要对用户输入进行严格的校验。如果校验规则不能完全拦截有攻击性的字符时，需对输入参数进行编码。

**参数校验存在漏洞**的场景：

(1) 当允许用户输入易造成SQL注入的特殊字符时(如’ –等)，符合校验规则的数据任然可造成SQL注入。

## 对于无法通过参数校验防止SQL注入时 可对参数进行编码

对参数进行编码，可以防止改变SQL语句的逻辑，保证信息被正确的处理，也是一种有效的防SQL注入的方案。

前面已经提到，即使是使用白名单校验的方式，有时发送给数据库的数据仍然是不安全的，尤其是动态SQL语句使用了该信息，这时使用输出编码是一种有效的措施。例如，O’Boyle是一个有效的名字，白名单校验允许该信息，如果使用该值动态拼接一个SQL查询，可能会破坏SQL语句查询逻辑，造成一个数据库异常。

对数据进行编码时，需考虑不同类型数据库之间的差别。推荐使用OWASP ESAPI提供的标准方法encodeForSQL(Codec codec, String input)，该方法需要制定数据库类型，目前该方法支持oracle，Mysql，DB2三种数据库类型。

**该方案的缺点是目前成熟的CBB中，编码并没有覆盖所有类型的数据库**。

## 新起项目推荐使用数据库操作框架

常见的数据操作框架MyBaits、Hibernate等。这些框架本身提供一些防SQL注入的方案，开发人员只要使用正确，可彻底杜绝sql注入问题。例如MyBatis框架是在配置文件中对所有的数据库操作进行集中配置，只要有经验的人员集中审查一下这些配置文件，就可以解决所有的SQL注入问题。

MyBatis中，SQL映射允许在SQL语句中通过#指定动态参数，例如：

|  |
| --- |
| *<select id="getItems" parameterClass="MyClass" resultClass="Item">*  *SELECT \* FROM t\_item WHERE owner = #userName# AND itemName = #itemName#*  *</select>* |

#号括起来的userName和itemName两个参数指示MyBaits在创建参数化查询时将它们替换成占位符：

|  |
| --- |
| *String sqlString = "SELECT \* FROM t\_item WHERE owner=? AND itemName=?";*  *PreparedStatement stmt = connection.prepareStatement(sqlString);*  *stmt.setString(1, myClassObj.getUserName());*  *stmt.setString(2, myClassObj.getItemName());*  *ResultSet rs = stmt.executeQuery();*  *// ... convert results set to Item objects* |

然而，MyBaits也允许使用$符号指示使用某个参数来直接拼接SQL语句，这种做法是有SQL注入漏洞的：

|  |
| --- |
| *<select id="getItems" parameterClass="MyClass" resultClass="items">*  *SELECT \* FROM t\_item WHERE owner = #userName# AND itemName = '$itemName$'*  *</select>* |

MyBaits将会为以上SQL映射执行类似下面的代码：

|  |
| --- |
| *String sqlString = "SELECT \* FROM t\_item WHERE owner=? AND itemName='" + myClassObj.getItemName() + "'";*  *PreparedStatement stmt = connection.prepareStatement(sqlString);*  *stmt.setString(1, myClassObj.getUserName());*  *ResultSet rs = stmt.executeQuery();*  *// ... convert results set to Item objects* |

在这里，攻击者可以利用itemName参数发起SQL注入攻击。

Hibernate框架中也支持参数化查询，该框架下可支持基于位置的参数化查询、基于名称的参数化查询和原生的参数化查询。

|  |
| --- |
| *//基于位置的参数化查询*  *String userName = ctx.getAuthenticatedUserName(); //this is a constant*  *String itemName = request.getParameter("itemName");*  *Query hqlQuery = session.createQuery("from Item as item where item.owner = ? and item.itemName = ?");*  *hqlQuery.setString(1, userName);*  *hqlQuery.setString(2, itemName);*  *List<Item> rs = (List<Item>) hqlQuery.list();*  *//基于名称的参数化查询*  *String userName = ctx.getAuthenticatedUserName(); //this is a constant*  *String itemName = request.getParameter("itemName");*  *Query hqlQuery = session.createQuery("from Item as item where item.owner = :owner and item.itemName = :itemName");*  *hqlQuery.setString("owner", userName);*  *hqlQuery.setString("itemName", itemName);*  *List<Item> rs = (List<Item>) hqlQuery.list();*  *//原生参数化查询*  *String userName = ctx.getAuthenticatedUserName(); //this is a constant*  *String itemName = request.getParameter("itemName");*  *Query sqlQuery = session.createSQLQuery("select \* from t\_item where owner = ? and itemName = ?");*  *sqlQuery.setString(0, owner);*  *sqlQuery.setString(1, itemName);*  *List<Item> rs = (List<Item>) sqlQuery.list();* |

# 排查范围及工具

**排查范围：**所有涉及Web应用的自研产品（此次排查不涉及第三方系统）。

**排查模块：**Web应用系统中所有涉及数据库操作的模块。

**排查内容：**排查所有涉及数据库的操作。

## SQL注入问题工具排查

工具只能作为辅助手段，任何工具都无法完全排查SQL注入的问题。

### 白盒工具检查：

1. 使用CodeDEX工具进行扫描，重点排查是否存在下列规则告警：

SQL Injection

SQL Injection: Hibernate

SQL Injection: JDO

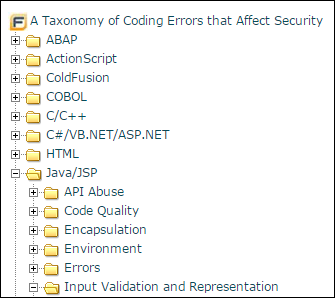
SQL Injection: MyBatis Mapper

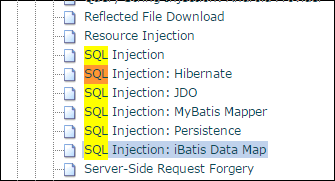
SQL Injection: Persistence

SQL Injection: iBatis Data Map

具体的告警说明和案例可以访问下面链接：

<http://www.hpenterprisesecurity.com/vulncat/zh_CN/vulncat/index.html>

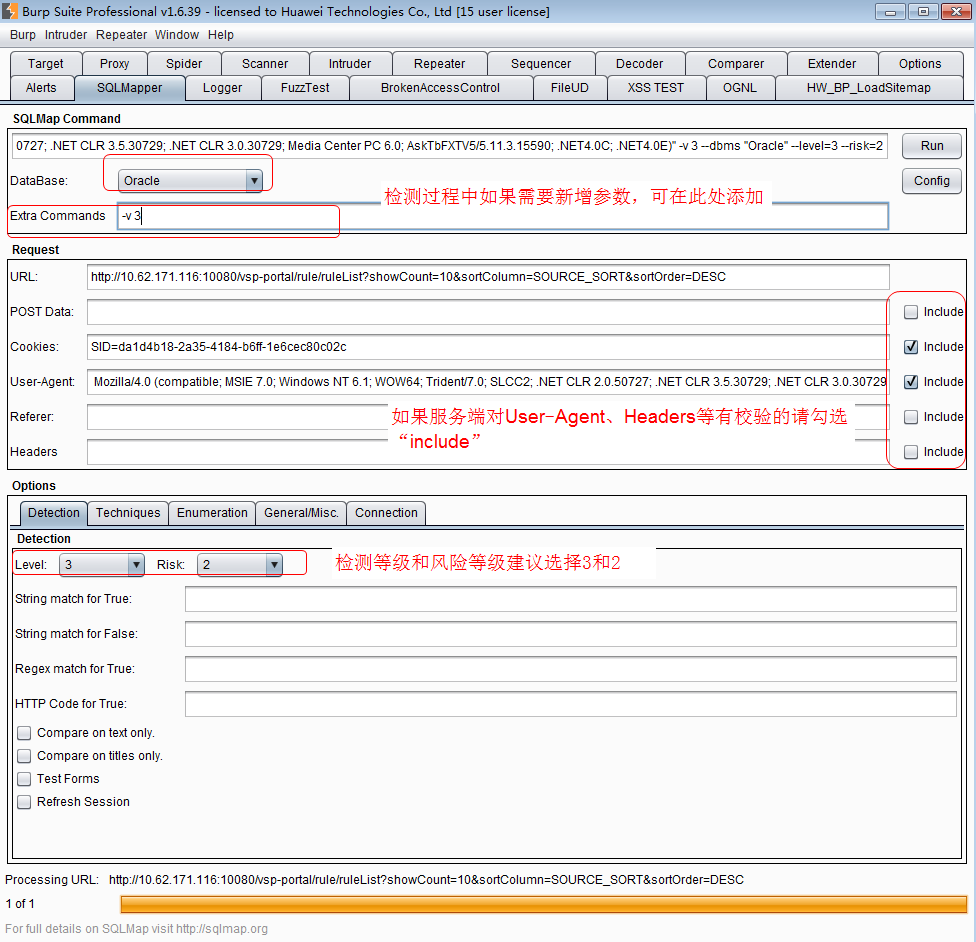




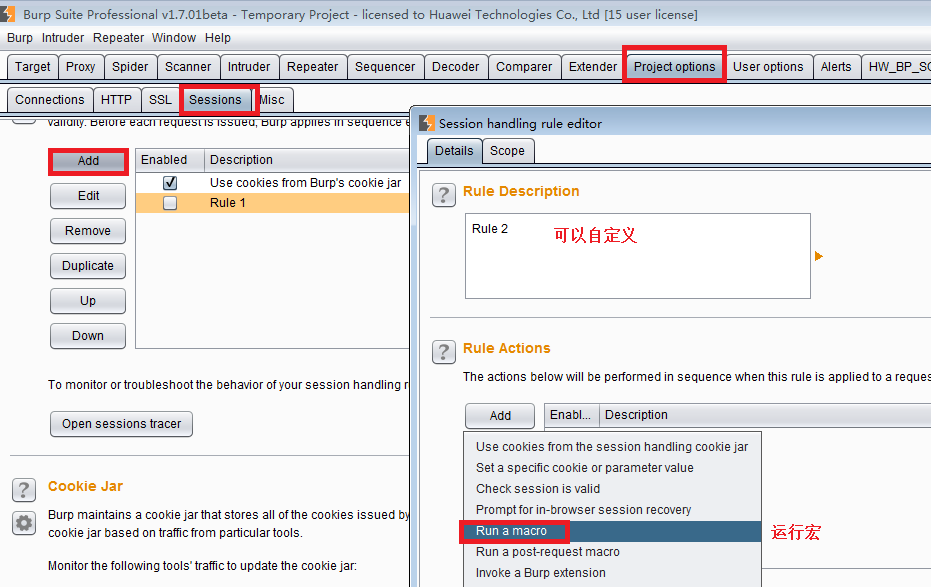
1. 使用BME平台的产品，必须要使用BME\_CI进行源码扫描，具体工具获取及使用方法详见：<http://3ms.huawei.com/hi/group/5255/blog_1744329.html?mapId=2503143&for_statistic_from=my_group_blog>

### 黑盒工具排查：

1. 使用AppScan时候建议补充人工页面爬取手段，保证测试页面的全面性，测试时可以使用默认SQL注入策略；
2. Burp suite已经集成SqlMap，可以进行批量SQl注入测试，检测等级和风险等级建议选择3和2，若系统对User-Agent、Headers有校验请勾选“Include”；



使用时需要保证会话有效，建议对登录进行录制



具体录制方法请见：

http://3ms.huawei.com/hi/group/1007713/file\_9026071.html

# 排查指导

## 排查是否存在动态SQL拼接问题

* 根据**SQL、createStatement**等关键词在代码中检索，查找是否存在动态拼接SQL语句问题；
* 检查存储过程中是否在SQL语句拼接的用法，参考关键词：SQL Server中的**EXEC(@sql)**、MySQL中的**concat**、Oracle中的|| 等；

## 排查与数据库操作相关的不可信参数是否存在被绕过问题

不同数据库中，具有特殊含义的字符不同，对不可信数据进行校验之前，开发人员首先要清楚针对使用的数据库，哪些字符具有特殊含义。

* 如果采用**黑名单校验**，检查是否存在被绕过的风险，绕过方式一般包括：使用CHR()/CHAR()绕过特殊字符校验、使用/\*\*/替代空格等；
* 如果采用白名单校验，白名单校验规则是否包含可造成SQL注入的特殊字符，比如’等；

## 排查是否使用成熟CBB的防SQL注入编码

* 如果产品代码中使用了输出编码的方式防止SQL注入，检查编码方式是否是自研代码实现，建议采用业界成熟CBB中编码函数，如ESAPI中的encodeForSQL()进行编码。

## 排查数据库操作框架使用是否正确

* 对于使用MyBatis框架，必须使用#指定动态参数。检查所有数据库相关的xml配置文件，排查是否存在使用$指定参数
* 对于使用Hibernate框架，推荐使用原生参数化查询或基于名称的参数化查询，对于所有动态拼接SQL语句的场景都要进行审核。

# 结果反馈

## 排查完成后请各产品如实反馈排查过程中发现的共性问题，各BU的排查结果统一反馈到BU安全办公室接口人处汇总，具体反馈格式请参考以下表格：



# 技术支持

漏洞排查过程中有技术问题及漏洞修复问题可咨询安全能力中心，联系人如下：

孙加奉 90006809 陈群华 00308701 吴伟90005155