计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2020,29(8):144−151 [doi: 10.15888/j.cnki.csa.007524]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于模型的 Web 应用二阶 SQL 注入测试用例集生成①

尤　枫, 王维扬, 尚　颖

(北京化工大学 信息科学与技术学院, 北京 100029)
通讯作者: 尤　枫, E-mail: bjbj1234@126.com

摘　要: SQL 注入漏洞一直以来都是威胁 Web 应用安全的主要问题之一, 其中二阶 SQL 注入漏洞相较于一阶 SQL 注入更加隐蔽且威胁更大, 对其检测通常依赖于测试人员的先验知识与经验. 目前, 在缺乏源码信息的黑盒测试场景下, 还没有针对该漏洞的有效检测手段. 利用基于模型的测试用例生成思想, 提出了一种基于客户端行为模型的测试用例集生成方法 (CBMTG), 用于生成检测 Web 应用二阶 SQL 注入漏洞的测试用例集. 首先通过初始测试用例集的执行建立迁移与 SQL 语句的映射关系; 然后通过 SQL 语句的字段分析建立迁移之间的拓扑关系; 最后通过拓扑关系来指导最终的测试用例集生成. 实验结果表明, 本文方法优于当前主流的二阶 SQL 注入漏洞检测方法.

关键词: web 应用测试; 模型; 基于模型的测试; 二阶 SQL 注入; 测试用例集生成

引用格式: 尤枫,王维扬,尚颖.基于模型的 Web 应用二阶 SQL 注入测试用例集生成.计算机系统应用,2020,29(8):144–151. http://www.c-s-a.org.cn/1003-3254/7524.html

Model Based Web Application Second-Order SQL Injection Test Suite Generation

YOU Feng, WANG Wei-Yang, SHANG Ying

(College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: SQL injection vulnerability has been the one of the most problems that threaten Web application security. Among them, second-order SQL injection vulnerabilities are more subtle and destructive than the first-order one, and the detection usually depends on the tester's prior knowledge and experience. At present, in the Black-Box Testing scenario, there is no effective detection method for the second-order vulnerability yet. Utilizing the idea of model-based test case generation, in this study, a Test suite Generation method based on a Client Behavior Model (CBMTG) is proposed to get a test suite capable of detecting second-order SQL injection vulnerabilities in Web applications. In the CBMTG, firstly, the mapping relationship between transitions and SQL statements is established through the execution of the initial test suite. Then, the topological relationship between transitions is established through the field analysis of the SQL statements. Finally, the final test suite is generated under the guidance of the topological relationship. The experimental results show that the method in this study performs better in most Web application than the state-of-the-art second-order SQL injection vulnerability detection methods.

Key words: Web application testing; model; model based testing; second-order SQL injection; test case generation

1　引言

在 21 世纪, 互联网上各种 Web 应用随处可见, 它们在为我们提供便利服务的同时也带来了不同程度的安全隐患. 相较于传统应用, Web 应用因其开发快捷,

① 基金项目: 国家自然科学基金 (61672085)
Foundation item: National Natural Science Foundation of China (61672085)
收稿时间: 2019-12-21; 修改时间: 2020-01-19; 采用时间: 2020-02-11; csa 在线出版时间: 2020-07-29

部署方便, 易于使用等优点被众多企业所青睐, 但也因此具有更多的安全问题. 2013 年的一篇报导指出在 2012 年约有 22% 的网站遭受了安全攻击[1]. SQL 注入漏洞是一种常见 Web 安全漏洞, 在最近几年更是成了影响 Web 应用安全的主要漏洞之一[2,3]. 虽然一些脚本语言 (如 PHP, Java) 通过提供参数化的方法来避免这种漏洞, 但是由于一些历史以及教育上的因素[4], 这些漏洞还在持续的增加.

SQL 注入漏洞可分为两类, 一阶 SQL 注入漏洞和以二阶 SQL 注入漏洞为代表的多阶 SQL 注入漏洞[5]. 二阶 SQL 注入漏洞相比于一阶 SQL 注入漏洞, 其注入路径更加曲折, 隐蔽性远高于一阶 SQL 注入漏洞, 这使得其更加难以检测, 且由于恶意数据会被持久性存储, 所以其造成的影响更加恶劣[6].

2004 年二阶代码注入攻击概念被首次提出[7]. 在之后的几年里, 尽管对一阶代码注入漏洞的相关研究得到了许多成果, 但是对二阶代码注入漏洞的研究却进展缓慢. 2010 年, Bau 等对当时主流的黑盒测试工具进行评估, 发现它们并不能检测出程序中的二阶代码注入漏洞[8]. 目前针对二阶 SQL 注入漏洞的检测主要是采用白盒与灰盒的测试方法, 即基于程序的源码信息来检测漏洞, 如文献 [9–11] 中提出的方法. 文献 [9] 使用对程序源码进行静态分析, 获取污点传播路径, 将这样一条数据注入点到触发点的路径认定为程序的漏洞. 文献 [10] 在文献 [9] 的基础上将程序源码的语义分析与数据的存储状态分析相结合, 以此来找到二阶代码注入漏洞的污点传播路径, 并将这种路径认定为二阶代码注入漏洞. 在文献 [11] 中, 闫璐等针对二阶 SQL 注入漏洞提出了一种静态分析与动态测试相结合的检测方法, 该方法通过静态分析找到源码中可能存在二阶 SQL 注入漏洞的 SQL 语句和列名, 然后通过提交 HTTP 请求的方法验证漏洞. 该方法有效地利用了程序内部信息来提高动态测试的准确性, 同时也利用了动态测试的特性降低了静态分析的误报率. 这种基于静态分析的漏洞检测方法, 往往依赖于静态分析工具设计人员的先验知识, 这就导致在面对具有简单内部逻辑的应用时, 静态分析往往能表现出很好的效果, 但是在复杂的 Web 应用上的表现却不尽如人意. 如今的 Web 应用大多使用面向对象的设计思想, 大量使用第三方框架以及参数化的数据库操作, 使得现有的白盒与灰盒测试方法很难对这些特性进行分析.

黑盒测试是一种在任何 Web 应用上都适用的测试方法, 其针对大规模代码单元的测试效率要远高于白盒测试, 且在某些场景下测试人员只能采用黑盒的测试方法. 但由于对其研究的复杂性, 目前对该领域的研究成果较少, 还没有一种能够可靠且高效检测二阶 SQL 注入漏洞的方法. 虽然文献 [12] 提出了一种基于页面爬虫的二阶代码注入漏洞检测方法, 该方法对被测网站进行两次爬取, 第一次用于获取页面 URL 并设置不同类型锚点, 第二次则只针对具有存储锚点的页面进行爬取, 来达到触发漏洞的目的. 但该方法并未有效地利用二阶 SQL 注入漏洞的特点, 作者虽然表明该方法能够检测出二阶 SQL 注入漏洞, 但是在漏洞检测效果方面却并未给出更多的实验结果. 随着模型语言 (如 UML)、模型驱动技术 (MDA) 和形式化验证技术的逐步成熟, 以及以测试为中心的软件开发技术与方法的兴起和应用, 基于模型的软件测试在学术界和工业界得到越来越多的重视, 已成为自动化测试的一个重要研究方向. 其中基于模型的 Web 应用软件的测试方法, 目前在国内外已有一些研究与进展[13], 通过 Web 应用模型来生成测试用例是十分高效的且有较好的预期.

本文利用 Web 应用的前端模型, 通过对二阶 SQL 注入漏洞与前端模型之间关系的深入分析, 提出了一种基于客户端行为模型的测试用例集生成方法 (CBMTG), 该方法能够高效地生成检测二阶 SQL 注入漏洞的测试用例集.

2 相关技术

2.1 基于 CBM 的测试用例

在 Web 2.0 时代, Ajax 技术被大量应用到 Web 应用中, 而针对 Web 应用漏洞的攻击是可能发生在这种 Ajax 请求之中的. 传统的 Web 应用模型无法准确的对这种 Web 应用中的动态行为进行建模, 在本课题组之前的相关研究中, 以 Web 应用的 DOM 结构为基准提出了一种 Web 应用的客户端行为模型 (CBM), 该模型能够准确的描述 Web 应用的客户端行为[14].

客户端的行为模型表示为一个 4 元组<S, I, O, T>, 其中 S 表示一个有限的状态集合, I 表示一个输入变量的有限集合, O 表示输出变量的有限集合, T 是一个迁移的有限集合. S 中的每一个成员都是一个状态, 用 URL 以及其对应的 DOM 结构来表示, I 和 O 中的每一个成员都是一个变量, T 中的成员则表示从一个状态

到另一个状态的迁移. T 中的每一个迁移 t 都用一个 5 元组<*src*, *event*, *cond*, *act*, *trgt*>表示, 其中 *src*, *trgt*∈*S*, *src* 表示起始状态, *trgt* 表示目标状态, *event* 表示用户在 *src* 状态所触发的事件, *cond* 表示在触发事件 *event* 时进行当前的状态转移必须要满足的条件, *act* 则表示触发事件之后 web 应用所产生的 DOM 操作等行为. 在迁移具有输入时, *event* 还可以表示为 *event*(*inputlist*), 表示当前 *event* 需要输入一组变量 *inputlist*.

对 Web 应用 CBM 模型的建立在文献 [14] 中有详细描述, 如图 1 所示展示了一个简单的 CBM 模型, 其中每一个节点表示一个 S 中的状态, 有向边 t_i 表示 T 中的一个迁移, 为了表述方便图中隐去了 t_i 的细节.

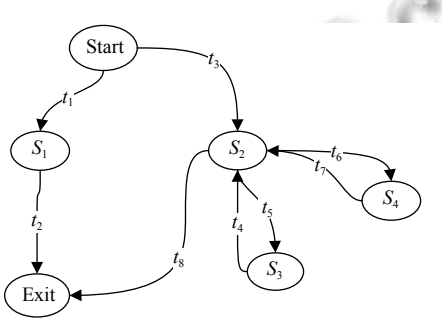

图 1 CBM 模型

在针对 Web 应用的自动化测试领域, 测试用例通常是指一系列的用户操作的集合, 而 CBM 中的每一个迁移, 都对应于一次用户操作, 故如果一个 Web 应用的 CBM 模型为 M=<S, I, O, T>, 则针对该 Web 应用的测试用例可以表示为有序集合 C=[t_1, t_2, t_3,…, t_n], 其中 t_i∈T 且满足 t_{i-1} 是 t_i 的一个直接前驱迁移.

在本文的研究中, 为了降低生成测试用例复杂程度, 规定 CBM 一定包含两个状态节点 start 和 exit, 分别表示 Web 应用的起始状态和终止状态. Start 状态一般是指 Web 应用的主页或者登录页面等; exit 状态可以表示用户登出之后的页面, 也可能表示一个虚拟状态, 指用户已经停止操作. 故在本文中, 测试用例是指一个在 CBM 中使得状态从 start 转移到 exit 的一个迁移集合, 并将其称为从 start 到 exit 的一条路径. 例如, 在图 1 中, t_3, t_5, t_4, t_8 这 4 条迁移构成的从 start 到 exit 的路径表示一个测试用例.

2.2 Web 应用二阶 SQL 注入攻击

在一阶 SQL 注入的情景下, 用户输入的恶意变量被直接拼接到 SQL 查询语句中并更改正常的 SQL 语句逻辑. 而在二阶 SQL 注入情形下, 注入过程被分为两个阶段, 首先用户输入的恶意变量会先被存入数据库中, 然后在用户进行第二次请求时, 之前存入数据库或文件系统的恶意变量被取出, 并被用于构造另一条 SQL 语句, 此时该 SQL 语句的正常逻辑被更改. 如代码 1 和代码 2 展示了一个 Web 应用中添加与更新用户信息的代码段中所包含的二阶 SQL 注入漏洞.

代码 1. addUser.php

```
1. $link=new mysqli("localhost", "root", "root");
2. $query="INSERT INTO Users VALUES(‘?’, ‘?’, ‘?’)";
3. cmd=cmd=1ink->prepare($qurey);
4. cmd−>bindparam("sss",cmd−>bindparam("sss",usemame,$password,
$email);
5. $cmd->execute();
```

代码 2. updateUser.php

```
6. $query="SELECT * FROM Users WHERE username=‘?’ AND
password=‘?’ ";
7. result=mysqlquery(result=mysqlquery(query);
8. row=mysqlfetcharray(row=mysqlfetcharray(result);
9. usemame=usemame=row["username"];
10. email=email=row["email"];
11. password=password=_POST["newpwd"];
12. $query="UPDATE Users SET username=‘$username’,
password=‘$password’, email=‘$emal’ WHERE usemame=
‘$username’";
13. rs=mysqlquery(rs=mysqlquery(query);
```

在 addUser.php 文件中采用的是预编译语句来将用户信息插入数据库中, 而非动态构造 SQL 语句. 由于 DBMS 会通过占位符来编译 SQL 语句, 在接受用户输入时编译已经完成, 恶意数据无法改变 SQL 语句的执行逻辑, 所以这种参数化方法能够有效的防御一阶 SQL 注入攻击. 如果将 email 输入为“123’ WHERE 1=(updatexml (1, concat (0x5e24,(select password from admin limit 1), 0x5e24), 1)); -- ”, 该恶意数据则会被存入数据库的“users”表中. 在 updateUser.php 文件中, 6-8 行执行数据库查询语句从 users 表中取出用户信息并保存到内存中, 在第 12 行利用内存中的用户信息动态构建更新用户信息的 SQL 语句, 由于之前在数据库中 email 存入的是恶意数据, 此时构建的 SQL 语句为“UPDATE Users SET username=‘XXX’,password=‘XXX’, email=‘123’ WHERE l=(updatexml(1, concat (0x5e24, (select password from admin limit 1), 0x5e24), 1)); --‘WHERE username=‘XXX’”, 该条语句会通过错误信息

暴露 admin 的密码, 而非更新用户信息.

3 方法框架

二阶 SQL 注入漏洞的触发至少需要 3 条 SQL 语句的执行, 分别对应“存储”、“取出”、“使用”这 3 个过程, 在 2.2 节的示例中, 2, 6 和 12 行分别对应了这 3 条 SQL 语句. 事实上这个过程也是用户输入的数据到达程序注入点过程, 从“存储”到“取出”, 数据在数据库中; 而从“取出”到“使用”, 数据则是在程序的运行内存中. 在黑盒条件下, 内存中的数据流向是未知的, 故本文只要求最终的 SQL 语句执行满足“存储”然后“取出”的顺序, 该执行顺序也是触发二阶 SQL 注入漏洞的必要条件. 为生成具有特定迁移执行顺序的测试用例集, 使得 SQL 语句的执行顺序满足触发漏洞的必要条件, 本文提出基于客户端行为模型的 Web 应用二阶 SQL 注入测试用例集生成方法 (CBMTG), 该方法的框架如图 2 所示. CBMTG 主要由 3 部分组成, 分别为初始测试用例集生成, 拓扑关系图 (Topo 图) 构建和最终的测试用例集生成.

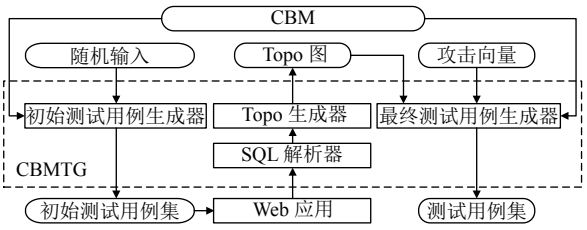

图 2 CBMTG 框架

CBMTG 首先通过对一个初始测试用例集的执行来建立迁移与 SQL 语句之间的映射关系, 然后对迁移对应的 SQL 语句进行字段分析以确定迁移之间的依赖关系, 并以此为依据建立迁移之间的 Topo 图, 最终以 Topo 图为指导生成最终的测试用例集.

4 二阶 SQL 注入测试用例集生成

4.1 初始测试用例集

在 Web 应用中, 用户进行的每一次操作都能对应到一个后台 DBMS 所执行的一个 SQL 语句集合. 对 CBM 中的每一个迁移同样与 SQL 语句之间存在对应关系. 本文对这种关系给出如下定义.

定义 1. 对 CBM 中 T 的每一个迁移 t, 存在一个单射函数 f, 使得 $f(t)=SQLs$. 其中 $SQLs$ 表示执行迁移 t 时 DBMS 所有可能执行的不同结构的 SQL 语句集合.

CBMTG 通过动态执行迁移 t 来确定其对应的 $SQLs$. 为了保证 CBM 上的每一个迁移都能建立映射关系 f, CBMTG 以 CBM 的迁移全覆盖为目标来生成初始的测试用例集. 并且, 为了获取到具有正常逻辑的 SQL 语句以及保证程序的正常运行, 初始测试用例中的输入采用随机输入.

单个测试用例的生成过程是一个迁移选择的过程, 即从 CBM 的 start 节点开始选择迁移直到达到 exit 节点. 迁移选择采用贪心算法, 即对一个节点的射出迁移集合, 从中优先选择之前被选择次数最少的迁移, 如果生成的测试用例中所有迁移都在之前生成的测试用例中出现过, 则舍弃该测试用例重新生成, 直到 CBM 中所有迁移都被选择.

贪心算法是一种求可行解的搜索算法, 只要求达到近似最优解, 相较于其他求全局最优解的启发式搜索算法, 如遗传算法、粒子群算法等, 其更加简单高效. 生成初始测试用例集用以覆盖到 CBM 中所有迁移的问题有多个可行解, 本文仅需找到其中之一即可, 所以该问题可以通过贪心算法求解. 在求解过程中, 将问题分解为针对当前节点的射出迁移选择问题, 使得每一次局部迁移选择都能够更加接近迁移的全覆盖.

算法 1 是初始测试用例集的生成算法, 该算法输入为 CBM 模型, 并根据迁移条件以及迁移的覆盖情况选择迁移, 输出为初始测试用例集.

算法 1. 初始测试用例生成

输入: CBM<S,T>
输出: 测试用例集 C

```
1. Array outgoTran[][]//store all outgoing for each state
2. Array outgoState[]//store all outgoing for each trans
3. Array C=φ;
4. while not allcovered(T) do
5.    Array transList=φ;
6.    nextNode='start';
7.    while nextNode!='exit'do
8.       nextTrans=selectByTimes(outgoTran[nextNode]);
9.       transList.push(nextTrans);
10.      T[nextTrans].coveredTimes+=1;
11.      nextNode=outgoState[nextTrans];
12.   end while
13.   C.push(transList);
14. end while
15. return C.
```

4.2 迁移依赖关系

在获取到迁移对应的 SQL 语句集合后, 由于同一个迁移可能被多次执行, 所以该集合中可能存在大量冗余的 SQL 语句, 而通过 SQL 语句的解析树可以判断这些冗余的 SQL 语句. SQL 作为一种结构化的语言, 对其解析树进行分析是一种常见的研究手段, 如文献 [15] 利用解析树来检测 SQL 注入攻击. 本文采用文献 [15] 对 SQL 解析树的表示方式, 将具有相同解析树结构的 SQL 语句判定为相同的 SQL 语句, 并以此来去除冗余, 即对所有具有相同解析树结构的 SQL 语句只保留其中之一, 最终得到迁移对应的 *SQLs*. 所以 *SQLs* 中每一个 SQL 语句都不相同. 图 3 展示了一个 SQL 解析树的例子. 在不考虑 literal 节点的子节点情况下, 只要其他节点相同, 则两个 SQL 解析树结构相同.

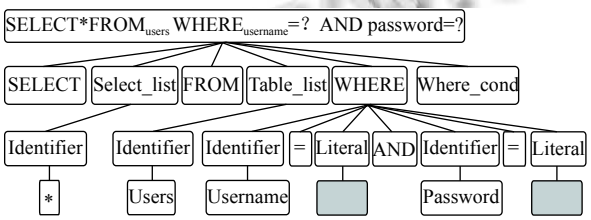

图 3 SQL 解析树

在本文中, SQL 语句为触发二阶 SQL 注入漏洞而需满足的执行顺序表示为 SQL 语句之间的依赖关系, 该依赖关系可通过对其进行字段分析得到. SQL 语句之间的依赖关系定义如下.

定义 2. 设 sql_1 为操作字段包含 a 的 select 类型 SQL 语句, 如果 a 为长字符串类型, 且存在另一个 insert 或 update 类型 SQL 语句 sql_2 以 a 作为操作字段, 则称 sql_1 依赖于 sql_2.

在测试用例集中, 迁移是对 Web 应用进行操作的最小单位, 因为迁移与 *SQLs* 之间存在单射关系, 即一个迁移对应多个 SQL 语句, 本文在此基础上建立迁移之间的依赖关系, 使得当所有迁移之间的依赖关系被满足时, 所有 SQL 语句之间的依赖关系被满足, 本文定义迁移之间的依赖关系如下.

定义 3. 设迁移 $f(t_1)$ 中存在 SQL 语句 sql_1, $f(t_2)$ 中存在 SQL 语句 sql_2, 使得 sql_1 依赖于 sql_2, 则称 t_1 依赖于 t_2.

例如, 迁移 t_1 对应 *SQLs* 为{select name from user where id='*userInput*', update schoolinfo set info='*userInput*'}, 迁移 t_2 对应 *SQLs* 为{insert into user (name) values('*userInput*'), select info from schoolinfo}, 在该例中迁移 t_1 依赖于 t_2, 同时 t_2 依赖于 t_1.

在定义 2 中, 由于数据库中非字符串类型以及短字符串字段无法存储恶意数据, 所以无需考虑这些类型的数据库字段, 字段信息通过查询 information_schema.columns 来获取.

4.3 Topo 图生成

CBM 中迁移之间的依赖关系以 Topo 图的形式来表示, Topo 图定义如下.

定义 4. Topo 图表示为一个二元组<N, E>, 其中 $N \subseteq T$, E 迁移之间依赖关系的集合, E 中每一个元素用二元组<t_{head}, t_{tail}>表示, t_{head}, $t_{tail} \in N$.

其中的二元组<t_{head}, t_{tail}>表示 t_{tail} 依赖于 t_{head}. 如图 4 所示是图 1 的 CBM 所相对应的 Topo 图, 图中的节点表示迁移, 边表示依赖关系. 可以看出 CBM 中的 t_2, t_5, t_6, t_7, t_8 都不在 Topo 图中, 这是因为这些迁移都未与其他迁移产生依赖关系. 事实上 Topo 图中的每一对依赖关系都表示为了触发二阶 SQL 注入漏洞迁移之间应满足的执行顺序.

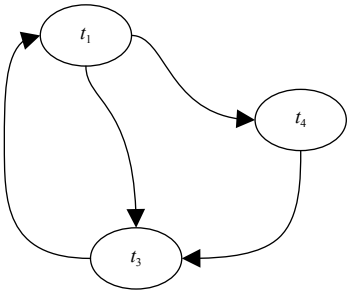

图 4 Topo 图

算法 2 是 Topo 图生成算法的伪代码. 在第 5 行, 使用开源工具 SQL-parser[16] 来进行 SQL 语句的解析, 获取其属性字段以及操作类型等 (该工具同样被用于 4.2 节的 SQL 语句去冗余操作中). 因为一个迁移可能对应多个不同类型的 SQL 语句, 故本文通过 storageAttr 来记录 insert 和 update 操作的字段, usageAttr 来记录 select 操作的字段, 然后通过求两者的交集来求得迁移之间的依赖关系, 其对应算法中的 16 行. 在 12 行中先将所有的迁移都放入 Topo 图中, 在 14–20 行判断它们的依赖关系, 最终在 21 行从 Topo 图中删除不存在任何依赖关系的迁移.

算法 2. Topo 图生成

输入: 迁移以及其对应 SQL 集合 *SQLs*[][]
输出: 拓扑图 Topo

```
1. Topo=obj();
2. foreach sqls in transSql do
3.    topoNode=obj();
4.    for sql in sqls do
5.       buf=parseSqlToObj(sql);
6.       if buf.type==‘insert’ || buf.type==‘update’ then
7.          topoNode.storageAttr.push(buf.attr);
8.       else if buf.type==‘select’ then
9.          topoNode.usageAttr.push(buf.attr);
10.      end if
11.   end for
12.   Topo.addNode(topoNode);
13. end for
14. foreach node1 in Topo.nodes do
15.   foreach node2 in Topo.nodes do
16.   if isIntersect(node1.storageAttr,node2.usageAttr) then
17.      Topo.addEdge(node1,node2)
18.   end if
19.   end for
20. end for
21. Topo.delAloneNode();
22. return Topo.
```

4.4 测试用例集生成

为满足触发二阶 SQL 注入漏洞的必要条件, 即使得迁移满足 Topo 图中的依赖关系, 最终的测试用例集需满足: 对 Topo 图中的任意一对依赖关系<t_1, t_2>, 在测试用例集中总能找到这两个迁移使得 t_1 在 t_2 之前执行. 最终的测试用例集需要迁移之间满足依赖关系, 本文同样采用贪心算法来选择迁移, 同时为了保证在最坏的情况, 即两个全覆盖迁移的测试用例集仅能满足一对迁移依赖关系下有解, 在 4.1 节算法的基础上增加了迁移依赖关系的判断, 在迁移选择时优先选择被依赖的迁移, 以此来找到最终的测试用例集, 算法流程如图 5 所示. 最终的测试用例集是一个测试用例的有序集合, 因为数据库中的数据是持久存储的, 即各个测试用例的执行情况通过数据库相互影响, 所以迁移之间只需要在测试用例集上满足执行顺序即可.

为了触发漏洞, 将测试用例上的所有输入都替换为攻击向量. 一般情况下, Web 应用通常会对用户输入进行过滤、进化等操作, 这使得输入的恶意数据被应用识别出来而根本不会被执行, 或者被破坏以致于达不到攻击的目的, 所以需要使用更加隐蔽的攻击向量. 目前对 SQL 注入攻击的攻击向量进行伪装的方法一般分为如下几种: 大小写混合、关键词替换、更改编码形式、使用注释、等价函数与命令、特殊符号、HTTP 参数控制, 以及这几种方式的组合[17,18]. 在本文的研究中, 为了便于攻击向量的自动生成, 攻击向量的生成规则以巴科斯范式 (BNF) 的形式进行表示.

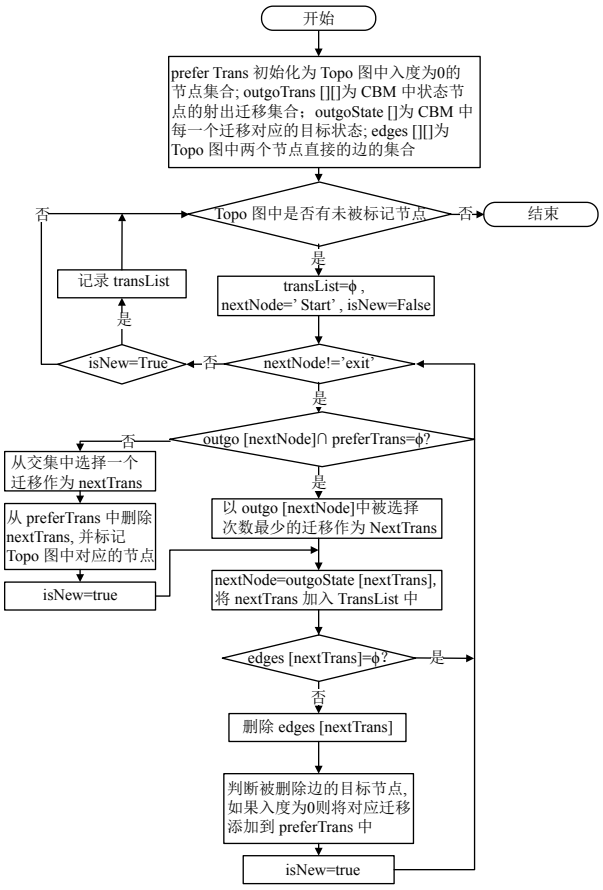

图 5 迁移选择流程图

5 实验评估

为了全方位评估 CBMTG 在检测二阶 SQL 注入漏洞的能力, 本文提出如下两个研究问题:

(1) CBMTG 是否能够有效地检测出 Web 应用中的二阶 SQL 注入漏洞, 其与当前现有的方法相比效果如何?

(2) CBMTG 使用的 Topo 图是否能有效地指导针对二阶 SQL 注入漏洞的测试用例集生成?

5.1 实验对象与环境配置

本文对如表 1 所示的 4 个开源 Web 应用进行了实验, 这些 Web 应用均使用 PHP 作为后端脚本, 并且

以 MySQL 作为数据库. 这些应用都可以从 Sourceforge 上获取得到.

表 1 被测程序

程序	程序大小 (Byte)	CBM模型大小<S,T>
SchoolMate-1.3	5085 323	<38,75>
addressbook-9.0.0.1	3898 299	<20,30>
phpaaCMS-1.0.0	1843 117	<37,69>
Webchess-1.0.0	484 743	<14,36>

注: S表示模型中节点数量, T表示模型中迁移的数量.

实验环境如下: CPU: Intel(R) Core(TM) i5-3 470 @3.20Ghz; 内存: 8 GB DDR3; 操作系统: Windows10; 浏览器: Chromever.78; 测试用例执行工具: Python3.6.4+ selenium3.0.1.

5.2 实验结果

5.2.1 漏洞检测能力

为了评估文中方法的漏洞检测能力, 本文将 CBMTG 与文献 [11] 中的灰盒方法以及 RIPS 工具进行了对比, 这两种方法是目前检测二阶 SQL 注入漏洞的主流方法. RIPS 是目前仅有的能检测二阶 SQL 注入漏洞的代码审计工具, 它来自于文献 [9,10] 中的白盒方法. 实验结果如表 2 所示.

表 2 漏洞检测结果

程序	RIPS		文献[11]	CBMTG
	ALL	TP	TP	TP
Schoolmate	144	1	1	1
Webchess	24	1	0	**2**
phpaaCMS	3	0	—	0
addressbook	7	0	—	**4**

注: ALL表示RIPS报告的所有漏洞, TP表示真实存在的漏洞

从表 2 中可以看出, CBMTG 能比 RIPS 以及文献 [11] 的方法检测出更多的二阶 SQL 注入漏洞. RIPS 使用静态分析的方法来检测漏洞, 可以看出其报告的漏洞要远远多于真实存在的漏洞. 经过对程序源码的分析发现, RIPS 将大量的主键数据的存取视为二阶注入漏洞. 同样从结果中可以看出 RIPS 在 phpaaCMS 和 addressbook 上检测效果并不理想, 这是因为相比于另外两个应用, phpaaCMS 和 addressbook 的源码具有更加复杂结构, 严重干扰了 RIPS 的静态分析能力. 而文献 [11] 的方法同样涉及源码的静态分析过程, 由于文献中并未说明针对复杂源码结构 (比如“类”、“命名空间”) 的分析方法, 所以本文没有得到对 phpaaCMS 以及 addressbook 的检测结果. 本文的 CBMTG 是一种黑盒的检测方法, 所以复杂的程序结构并不会影响方法的检测结果. 我们发现 CBMTG 并未从 phpaaCMS 检测出二阶 SQL 注入漏洞, 这是由于从数据库中查询出的恶意数据并未被用于构建另一个 SQL 语句, 虽然存在数据的“存储”到“取出”的过程, 但是事实上该应用中并不存在二阶注入漏洞.

5.2.2 CBMTG 有效性

为了评估迁移 Topo 图对测试用例生成的指导作用, 本文在 Schoolmate、Webchess、addressbook 等 3 个应用上将 CBMTG 生成的测试用例集与随机生成的测试用例集进行对比试验. 其中随机生成的测试用例集保持与 CBMTG 生成测试用例集同样的大小, 且每组随机试验都进行了 20 次, 取平均值作为最终检测出漏洞个数.

实验结果如图 6 所示. 横轴上括号中的数字表示单个测试用例集中的测试用例数量. 从图 6 中可以看出 CBMTG 优于随机的测试用例生成方法, 说明 CBMTG 中使用的 Topo 图确实能够有效地指导测试用例的生成. 同时结合测试用例数量后发现, 由于针对 Schoolmate 的测试用例集更加庞大, 导致随机测试用例集在该应用上的表现与 CBMTG 的测试用例集更加接近, 因为庞大的测试用例集能够使得迁移之间进行更多的组合, 这就导致了二阶 SQL 注入漏洞检测概率的提高.

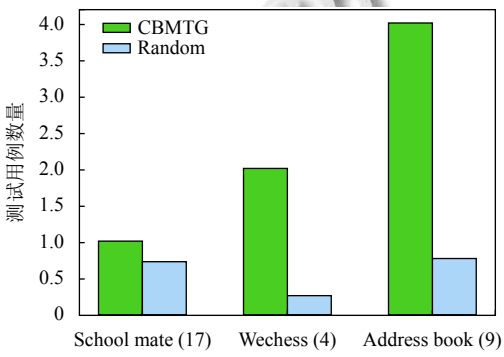

图 6 CBMTG 与随机测试用例集对比

6 结论与展望

在 Web 应用中, 二阶 SQL 注入漏洞比一阶 SQL 注入漏洞更加难以检测, 在黑盒情况下, 目前还没有一种有效的方法来针对该漏洞生成测试用例. 本文利用基于模型的测试用例生成思想, 提出一种基于客户端行为模型 (CBM) 的测试用例集生成方法, 利用客户端行为模型 (CBM) 先建立迁移与 SQL 语句之间的映射关系,

然后获取迁移之间的拓扑关系, 以此来指导测试用例集生成. 实验结果表明, 本文提出方法所生成的测试用例集能够有效地检测出 Web 应用中的二阶 SQL 注入漏洞. 在后续进一步的研究中, 作者将考虑在现有测试用例集的基础上去除不相关迁移以达到提高漏洞检测效率的目的, 同时探索将本文方法拓展到其他类型的二阶注入漏洞检测领域.

参考文献

1 Symantec. Internet threat report: 2012 trends, volume 18. Apr. 2013.

2 OWASP. OWASP top ten. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project.

3 CWE. 2019 CWE top 25 most dangerous software errors. http://cwe.mitre.org/top25/index.html.

4 Taylor C, Sakharkar S. ');DROP TABLE textbooks;--: An Argument for SQL injection coverage in database textbooks. Proceedings of the 50th ACM Technical Symposium on Computer Science Education. Minneapolis, MN, USA. 2019. 191–197.

5 Halfond WG, Viegas J, Orso A. A classification of SQL-injection attacks and countermeasures. Proceedings of IEEE International Symposium on Secure Software Engineering. Arlington, TX, USA. 2006. 13–15.

6 Sharma C, Jain SC. Analysis and classification of SQL injection vulnerabilities and attacks on web applications. Proceedings of 2014 International Conference on Advances in Engineering & Technology Research. Unnao, India. 2014. 1–6.

7 nccgroup. Second-order code injection attacks. https://www.nccgroup.trust/uk/our-research/second-order-code-injection-attacks/. (2013-08-23).

8 Bau J, Bursztein E, Gupta D, *et al*. State of the art: Automated black-box web application vulnerability testing. Proceedings of 2010 IEEE Symposium on Security and Privacy. Berkeley, CA, USA. 2010. 332–345.

9 Dahse J, Holz T. Simulation of built-in php features for precise static code analysis. Proceedings of the 21st Annual Network and Distributed System Security Symposium. San Diego, CA, USA. 2014. 23–26.

10 Dahse J, Holz T. Static detection of second-order vulnerabilities in web applications. Proceedings of the 23rd USENIX Security Symposium. San Diego, CA, USA. 2014. 989–1003.

11 Yan L, Li XH, Feng RT, *et al*. Detection method of the second-order SQL injection in Web applications. Proceedings of the 3rd International Workshop on Structured Object-Oriented Formal Language and Method. Queenstown, New Zealand. 2013. 154–165.

12 Liu M, Wang B. A Web second-order vulnerabilities detection method. IEEE Access, 2018, 6: 70983–70988. [doi: 10.1109/ACCESS.2018.2881070]

13 Javed H, Minhas N M, Abbas A, *et al*. Model based testing for web applications: A literature survey presented. Journal of Software, 2016, 11(4): 347–361.

14 Wang W, Guo J, Li Z, *et al*. Behavior model construction for client-side of modern Web applications. Tsinghua Science and Technology. [doi: 10.26599/TST.2019.9010043]

15 Buehrer G, Weide BW, Sivilotti PAG. Using parse tree validation to prevent SQL injection attacks. Proceedings of the 5th International Workshop on Software Engineering and Middleware. Lisbon, Portugal. 2005. 106–113.

16 A validating SQL lexer and parser with a focus on MySQL dialect. https://github.com/phpmyadmin/sql-parser.

17 Tian W, Yang JF, Xu J, *et al*. Attack model based penetration test for SQL injection vulnerability. Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops. Izmir, Turkey. 2012. 589–594.

18 Appelt D, Nguyen CD, Briand LC, *et al*. Automated testing for SQL injection vulnerabilities: An input mutation approach. Proceedings of 2014 International Symposium on Software Testing and Analysis. San Jose, CA, USA. 2014. 259–269.