Comparison of SQL Injection Detection and Prevention Tools based on Attack Type and Deployment Requirements

Atefeh Tajpour
Center for Advanced Software Engineering
University Technology Malaysia
Kuala Lumpur, Malaysia
tajpour81sn@yahoo.com

Maslin Masrom
Razak School of UTM in Engineering
University Technology Malaysia
Kuala Lumpur, Malaysia
maslin@ic.utm.my

Mohammad Jojor Zadeh Shooshtari
Center for Advanced Software Engineering
University Technology Malaysia
Kuala Lumpur, Malaysia
mohammadjorg@yahoo.com

Hossein Rouhani zeidanloo
Center for Advanced Software Engineering
University Technology Malaysia
Kuala Lumpur, Malaysia
h_rouhani@hotmail.com

I. INTRODUCTION

Web applications are often vulnerable to attacks, which can give attackers easily access to the application's underlying database. SQL injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer intended.

SQL Injection Attacks (SQLIAs) have known as one of the most common threats to the security of database-driven applications. So there is not enough assurance for confidentiality and integrity of this information. SQLIA is a class of code injection attacks that take advantage of lack of user input validation. In fact, attackers can shape their illegitimate input as parts of final query string which operate by databases. Financial web applications or secret information systems could be the victims of this vulnerability because attackers by abusing this vulnerability can threat their authority, integrity and confidentiality. So, developers addressed some defensive coding practices to eliminate this vulnerability but they are not sufficient.

II. DEFINITION OF SQLIA

SQL injection is a type of attack which the attacker adds Structured Query Language code to input box of a web form to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to a web application's underlying database and destroy functionality or confidentiality.
III. SQL INJECTION ATTACK TYPES

There are different methods of attacks that depending on the goal of attacker are performed together or sequentially. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query. Now the following classification of SQLIAs in accordance to the Halfond, Viegas, and Orso researches \[4, 11\] will be presented.

**Tautologies:** This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use WHERE clause.

"SELECT * FROM employee WHERE userid = '112' and password ='aaa' OR '1'='1""
As the tautology statement (1=1) has been added to the query statement so it is always true.

**Illegal/Logically Incorrect Queries:** when a query is rejected, an error message is returned from the database including useful debugging information. This error messages help attacker to find vulnerable parameters in the application and consequently database of the application. In fact attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose. In this example attacker makes a type mismatch error by injecting the following text into the pin input field:

1) Original URL:  
http://www.arch.polimi.it/eventi/?id_nav=8864  
2) SQL Injection:  
http://www.arch.polimi.it/eventi/?id_nav=8864'  
3) Error message showed:  
SELECT name FROM Employee WHERE id =8864'
From the message error we can find out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

**Union Query:** By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application.  
Suppose for our examples that the query executed from the server is the following:  
SELECT Name, Phone FROM Users WHERE Id=$id  
By injecting the following Id value:

\$id=1 UNION ALL SELECT creditCardNumber,1 FROM CreditCarTable
We will have the following query:

SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1 FROM CreditCarTable which will join the result of the original query with all the credit card users.

**Piggy-backed Queries:** In this type of attack, intruders exploit database by the query delimiter, such as ";", to append extra query to the original query. With a successful attack database receives and execute a multiple distinct queries. Normally the first query is legitimate query, whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker inject " 0; drop table user " into the pin input field instead of logical value. Then the application would produce the query:

SELECT info FROM users WHERE login='doe' AND pin=0; drop table users

Because of ";" character, database accepts both queries and executes them. The second query is illegitimate and can drop users table from the database. It is noticeable that some databases do not need special separation character in multiple distinct queries, so for detecting this type of attack, scanning for a special character is not impressive solution.

**Stored Procedure:** Stored procedure is a part of database that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, so, this part is as inject able as web application forms. Depend on specific stored procedure on the database there are different ways to attack. In the following example, attacker exploits parameterized stored procedure.

CREATE PROCEDURE DBO.isAuthenticated  
@userName varchar2, @pass varchar2, @pin int  
AS  
EXEC("SELECT accounts FROM users  
WHERE login='" +@userName+ "' and pass='" +@password+ "+' and pin=" +@pin);  
GO
For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder input "' ; SHUTDOWN; - -" for username or password. Then the stored procedure generates the following query:

SELECT accounts FROM users WHERE login='doe' AND pass=' ' ; SHUTDOWN; -- AND pin=

After that, this type of attack works as piggy-back attack. The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down. So, it is considerable that stored procedures are as vulnerable as web application code.

**Inference:** By this type of attack, intruders change the behaviour of a database or application. There are two well-known attack techniques that are based on inference: blind-injection and timing attacks.
Blind Injection: Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be more difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements. Consider two possible injections into the login field:

```sql
SELECT accounts FROM users WHERE login='doe' and l=0 -- AND pass= AND pin=0
SELECT accounts FROM users WHERE login='doe' and l=1 -- AND pass= AND pin=0
```

If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submit the first query and receives an error message because of "l=0". So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection.

Timing Attacks: A timing attack lets an attacker gather information from a database by observing timing delays in the database's responses. This technique by using if-then statement cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. This attack is similar to blind injection and attacker can then measure the time the page takes to load to determine if the injected statement is true. This technique uses an if-then statement for injecting queries. WAITFOR is a keyword along the branches, which causes the database to delay its response by a specified time.

For example, in the following query:

```sql
declare @s varchar(8000) select @s = db_name() if (ascii(substring(@s, 1, 1)) & (power(2, 0))) > 0 waitfor delay '0:0:5'
```

Database will pause for five seconds if the first bit of the first byte of the name of the current database is 1. Then code is then injected to generate a delay in response time when the condition is true. Also, attacker can ask a series of other questions about this character. As these examples show, the information is extracted from the database using a vulnerable parameter.

Alternate Encodings: In this technique, attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII, and Unicode. Because by this way they can escape from developer’s filter which scan input queries for special known "bad character". For example attacker use `char (44)` instead of single quote that is a bad character. This technique with join to other attack techniques could be strong, because it can target different layers in the application so developers need to be familiar to all of them to provide an effective defensive coding to prevent the alternate encoding attacks. By this technique, different attacks could be hidden in alternate encodings successfully.

In the following example the `pin` field is injected with this string: "0; exec (0x73587574 64 5f77 6e)," and the result query is:

```sql
SELECT accounts FROM users WHERE login=" AND pin=0; exec (char(0x73587574646f776e))
```

This example use the `char()` function and ASCII hexadecimal encoding. The `char()` function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the `shutdown` command by database when it is executed.

IV. SQL INJECTION DETECTION AND PREVENTION TOOLS

Although developers deploy defensive coding or OS hardening but they are not enough to stop SQLIAs to web applications so researchers have proposed some tools to assist developers. It is noticeable that there are more approaches that have not implemented as a tool yet. This paper emphasizes on tool not technique.

Huang and colleagues [18] propose WAVES, a black-box technique for testing web applications for SQL injection vulnerabilities. The tool identify all points a web application that can be used to inject SQLIAs. It builds attacks that target these points and monitors the application how response to the attacks by utilize machine learning.

JDBC-Checker [12, 13] was not developed with the intent of detecting and preventing general SQLIAs, but can be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. As most of the SQLIAs depend on input and detects attacks by comparing it against the structure of the actual query issued. CANDID’s natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

CANDID [2, 7] modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID’s natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

In SQL Guard [10] and SQL Check [5] queries are checked at runtime based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. In two approaches developer should to modify code to use a
special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

AMNESIA combines static analysis and runtime monitoring [16, 17]. In static phase, it builds models of the different types of queries which an application can legally generate at each point of access to the database. Queries are intercepted before they are sent to the database and are checked against the statically built models, in dynamic phase. Queries that violate the model are prevented from accessing to the database. The primary limitation of this tool is that its success is dependent on the accuracy of its static analysis for building query models.

WebSSARI [15] use static analysis to check taint flows against preconditions for sensitive functions. It works based on sanitized input that has passed through a predefined set of filters. The limitation of approach is adequate preconditions for sensitive functions cannot be accurately expressed so some filters may be omitted.

SecuriFly [14] is another tool that was implemented for java. Despite of other tool, chases string instead of character for taint information. SecurityFly tries to sanitize query strings that have been generated using tainted input but unfortunately injection in numeric fields cannot stop by this approach. Difficulty of identifying all sources of user input is the main limitation of this approach.

Positive tainting [1] not only focuses on positive tainting rather than negative tainting but also it is automated and does need developer intervention. Moreover this approach benefits from syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string.

IDS [6] use an Intrusion Detection System (IDS) to detect SQLIAs, based on a machine learning technique. The technique builds models of the typical queries and then at runtime, queries that do not match the model would be identified as attack. This tool detects attacks successfully but it depends on training seriously. Else, many false positives and false negatives would be generated.

Another approach in this category is SQL-IDS [8] which focus on writing specifications for the web application that describe the intended structure of SQL statements that are produced by the application, and in automatically monitoring the execution of these SQL statements for violations with respect to these specifications.

SQLPrevent [11] is consists of an HTTP request interceptor. The original data flow is modified when SQLPrevent is deployed into a web server. The HTTP requests are saved into the current thread-local storage. Then, SQL interceptor intercepts the SQL statements that are made by web application and pass them to the SQLIA detector module. Consequently, HTTP request from thread-local storage is fetched and examined to determine whether it contains an SQLIA. The malicious SQL statement would be prevented to be sent to database, if it is suspicious to SQLIA.

Swaddler [3], analyzes the internal state of a web application. It works based on both single and multiple variables and shows an impressive way against complex attacks to web applications. First the approach describes the normal values for the application’s state variables in critical points of the application’s components. Then, during the detection phase, it monitors the application’s execution to identify abnormal states.

V. EVALUATION

In this section, the SQL injection detection or prevention tools presented in Section IV would be compared. In fact the attack types which each tool is able to address are considered as well as deployment requirements.

A. Comparison of SQL Injection Detection/Prevention Tools Based on Attack Types

Proposed tools were compared to assess whether it was capable of addressing the different attack types presented in Section III. It is noticeable that this comparison is based on the articles not empirically experience.

Tables 1 summarize the results of this comparison. The symbol “•” is used for tool that can successfully stop all attacks of that type. The symbol “-” is used for tool that is not able to stop attacks of that type. The symbol “°” refers to tool that stop the attack type only partially because of natural limitations of the underlying approach.

As the table shows the stored procedure is a critical attack which is difficult for some tools to stop it. It is consisting of queries that can execute on the database. However, most of tools consider only the queries that generate within application. So, this type of attack make serious problem for some tools.

Table 1 Comparison of SQLI Detection/Prevention Tools with Respect to Attack Types
B. Comparison of SQL Injection Detection/Prevention Tools Based on Deployment Requirement

Each tool with respect to the following criteria was evaluated: (1) Does the tool require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the tool? (3) What is the degree of automation of the prevention aspect of the tool? (4) What infrastructure (not including the tool itself) is needed to successfully use the tool? The results of this classification are summarized in Table 2.

Table 2. Comparison of Tools Based on Deployment Requirements

<table>
<thead>
<tr>
<th>No</th>
<th>Tool</th>
<th>Modify Code base</th>
<th>Detection</th>
<th>Prevention</th>
<th>Additional Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AIDS [16]</td>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>IDS [6]</td>
<td>No</td>
<td>Auto</td>
<td>Generate report</td>
<td>IDS system Training set</td>
</tr>
<tr>
<td>3</td>
<td>JDBC Checker [12]</td>
<td>No</td>
<td>Auto</td>
<td>Code suggestion</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>Security [14]</td>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>None</td>
</tr>
<tr>
<td>5</td>
<td>SQLCHECK [5]</td>
<td>Yes</td>
<td>Semi Auto</td>
<td>Auto</td>
<td>Key management</td>
</tr>
<tr>
<td>6</td>
<td>SQLGuard [10]</td>
<td>Yes</td>
<td>Semi Auto</td>
<td>Auto</td>
<td>None</td>
</tr>
<tr>
<td>7</td>
<td>WAVES [18]</td>
<td>No</td>
<td>Auto</td>
<td>Generate report</td>
<td>None</td>
</tr>
<tr>
<td>8</td>
<td>WEBSSARY [13]</td>
<td>No</td>
<td>Auto</td>
<td>Semi auto</td>
<td>None</td>
</tr>
<tr>
<td>9</td>
<td>CANDID [7]</td>
<td>No</td>
<td>Auto</td>
<td>Generate report</td>
<td>None</td>
</tr>
<tr>
<td>10</td>
<td>SQL_IDS [8]</td>
<td>No</td>
<td>Auto</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td>11</td>
<td>Swaddler [5]</td>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>Training</td>
</tr>
<tr>
<td>12</td>
<td>Positive Tainting [1]</td>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>None</td>
</tr>
<tr>
<td>13</td>
<td>SQLPrevent [11]</td>
<td>No</td>
<td>Auto</td>
<td>Auto</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 2 determines the degree of automation of tool in detection or prevention of attacks and also it shows that which tool needs to modify the source code of application. Moreover, additional infrastructure that is required for each tool is illustrated.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a survey and comparison of current tools for detecting and preventing SQLIAs. We first identified the various types of SQLIAs. Then we investigated SQL injection detection and prevention tools. After that we compared these tools in terms of their ability to stop SQLIA. Regarding the results some current tools ability should be improved for stopping SQLI attacks. Moreover, we compared these tools in deployment requirements that bring complexity of usage for users.

In our future work we will compare the tools in deployment requirements and present a complete evaluation for different aspects of the SQL Injection detection and/or prevention tools. In fact we will consider effectiveness, efficiency, stability, flexibility and performance to prove the strength and weakness of the tool.

VII. REFERENCES


