2016

Detecting SQLIA using execution plans

Sriram Nagarajan
Iowa State University

Detecting SQLIA using execution plans

by

Sriram Nagarajan

A thesis submitted to graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Johnny S. Wong, Co-major Professor
Samik Basu, Co-major Professor
Yong Guan
Ahmed Kamal

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my parents, Nagarajan and Krishnaveni, without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and support during the writing of this work.

CHAPTER 1. INTRODUCTION

An application that is accessed by users over a network such as the Internet or an Intranet.is called a Web Application [1]. Organizations use web applications to make their data available over the network. Attackers exploit the vulnerabilities in web applications to access sensitive data stored by the organization. SQLIA, SQL Injection Attack is a common vulnerability that prevails in web application.

SQLIA classified under Injection attacks in OWASP [2], usually occurs in web applications due to insufficient input validations. In this type of attack, the attacker submits SQL commands as input to the web application, which when executed in the backend database can cause harmful transaction. The attacker can either access sensitive information from the database or challenge the integrity of the data stored in the database.

OWASP, Open Web Application Security Project, considers SQLIA as one of the most serious security threats in web applications. It also considers SQLIA as top ten web application vulnerabilities of 2013[2]. FireHost, a secure cloud hosting company reports that there has been an estimated 69 percent increase of SQL Injection Attacks in the year 2012.[3]

SQL Injection Attack Example

A simple SQLIA is explained here. A web application is developed to view details of the Employees, stored in a database. A web application developer uses SQL query to access the records from database. The following is the query written by the developer:

Query: Select * from Employee where EmployeeID = {0}; {0} represents the user input.

A valid user enters the employee ID in this field. The query below represents the status of the query written by developer after a valid user input is passed

Valid Query: Select * from Employee where EmployeeID = 1234

An attacker trying to inject SQLIA, enters SQL token in the user input field, which when executed in the back ground causes harmful transactions. The query below represents the status of the attacker query. The attacker has used SQL keyword OR in the user input field.

Attacker Query: Select * from Employee Where EmployeeID = 6547 OR 'a'='a'.

By including the OR keyword the attacker has successfully gained access to the employee records.

Classification of SQLIA

Detecting SQLIA has been a great challenge to the researchers. Good programming practices such as defensive programming and complex input validations can be used to prevent SQLIA in some cases. Such programming practices take a lot of time and become very complex for larger systems. Also attackers try to get around these programming practices by finding new exploits in the web application [4][5][6].

There have been a lot of approaches proposed by the researchers to detect and prevent SQLIA. The efficiency of these approaches is calculated by the range of SQL injection attacks detected and also by the simplicity of the approach. There are various forms of SQLIA and they are classified into different groups based on the intent of attackers. The following are the classification of SQLIA

1. Tautology based attacks
2. Illegal/Logically Incorrect Queries
3. Union query
4. Piggy backed query

5. Stored Procedures
6. Inference
7. Alternate Encoding
8. Second order injection

An ideal SQL Injection detection algorithm should detect all the types of SQL injection attack mentioned above.

Execution plans

Execution plans available in SQL server tells us how a query is executed in the database. It is analogous to blue print for constructing a building. Execution plans are generated by the query optimizer, component responsible for calculating an optimal way to execute a query. Execution plans are used by the Database administrators to troubleshoot poorly performing query. Execution plans are widely used in query optimization.

There are two types of execution plans: Actual and Estimated. Estimated execution plans are generated from the view of an optimizer. Actual execution plan are generated when the query is executed. Actual and Estimated execution plans can be different in some cases as Estimated execution plans are predicted by the query optimizer.

Properties of an Execution plan:

- An execution plan contains all the objects involved in the query. The following are the significant objects that can be found in an Execution Plan.
    - Database name
    - Table name
    - Name of the Columns retrieved from the table
    - Name of the Columns used in the where clause of a query

- Defines the nature of the query. An execution plan helps us in identifying the type of DML/ DDL query that is executed.
- The where clause in an execution plan is computed/ optimized.

Example:

Query: Select * from Employees Where EmployeeID =1234 or 1=1 is optimized in execution plan as Select * from Employees.

Execution plans can be easily obtained in SQL server. It can be viewed in three formats: Graphical, Text and XML. Understanding an execution plan is easy and less time consuming.

Proposed Solution

This thesis presents an approach for detecting SQLIA using Execution Plans in SQL server. As execution plans are available only in SQL Server, we assume that our web application uses SQL server as its backend databases. Even though execution plans are available only in the SQL server, the concept of query optimizer constructing an optimal plan to execute a query exists in other databases as well. Oracle uses explain plan to optimize queries. However this thesis explains how an execution plan can be used for detecting SQLIA.

The proposed approach monitors for the structural changes in the query written in web application to detect SQLIA. The changes in structure of a query are monitored using execution plans. Two execution plans are generated. One execution plan is generated from the query written by the web developer. This execution plan represents the structure of the intended query. The other execution plan is generated when the query written by the developer is executed. This execution plan represents the structure of the executed query. These two execution plans are compared for structural changes in the query. Changes in the structure of the queries are reported as SQLIA.

The objects available in the execution plans define the structure of a query. When two execution plans are compared, the occurrence of objects in each execution plans are compared. A query has changed its structure if

1. An object is not present in the execution plan.
2. A new object is included in the execution plan.
3. Number of occurrence of an object is changed in the execution plan.

The approach is evaluated against all types of SQLIA. A Windows Application tool is developed to compare two queries using execution plan approach. The approach is compared with other tools that are available for detecting SQLIA.

Contributions

The main contributions of this thesis can be summarized as follows:

1. Detection algorithm for SQLIA.

   The approach proposes a new detection algorithm to detect SQLIA using execution plan in SQL server.

2. Evaluating the algorithm against all the types of SQLIA.

   The proposed approach is evaluated against each type of SQLIA. Explanation of how execution plan is used to detect each of this approach is explained.

3. Comparing with other approaches.

   The proposed approach is compared with other approaches available in detecting SQLIA. The efficiency of the algorithms are discussed based on the range of SQLIA detected, simplicity of the algorithm and false positive/ false negative rates.

4. An implementable framework.

   Two applications are developed.

a) A windows application that compares two queries using execution plans in C#.
b) Web application architecture with IDS installed. The IDS includes the implementation of execution plan approach.

Roadmap

The rest of this work is organized as follows: In Chapter 2 we present a review of related work in this research area. Then we present our approach to detect SQLIA in Chapter 3. Chapter 4 describes the implementation of the detection algorithm, and compares our approach with other detection approaches. Finally, we conclude with a discussion and opportunities for future development in Chapter 5.

CHAPTER 2. RELATED WORK

With the advent of such a harmful attack, researchers have discovered various detection and prevention techniques for SQLIA. There has been a lot of effort spent in understanding the attack and its effect in web applications. Based on the understanding of the attack, SQLIA was classified into various categories based on the intent of the attack. Understanding the classes of SQLIA is significant, in crafting an algorithm to detect it. Various algorithms have been proposed to detect and prevent SQLIA. The underlying technique used in these algorithms determines the effectiveness of detecting SQLIA. This section discusses the classification of SQLIA and the techniques used by existing algorithm in detecting and preventing SQLIA.

Classification of SQLIA

Tautology based SQLIA

The aim of this attack is to inject code in one or more conditional statements so that it always evaluates to true. This type of attack is mostly used in authentication pages

Example:

The query used from [7][8][9][10] is to retrieve the records of an employee whose ID is 1234. The attacker using tautology based attacks retrieves information of all employees.

Intended query: Select * from Employees where EmpID = 1234 **Attacker query:** Select * from Employees where EmpID = 1234 or ‘a’=’a’

Illegal/Logically Incorrect Queries

This attack is based on the fact that error messages generated by an application can often reveal significant information to attackers. The attacker injects statements that cause syntax, type

conversion and logical errors in the database. The attacker uses the error message retrieved from the database to get significant information about the database.

Example:

The query used in this example [7][9][10][11] is for an ATM login page. The user enters pin information as input to the query.

Intended query: Select * from users where pin = 1234
Attacker query: Select * from users where pin=
convert(int,(select top 1 name from sysobjects where xtype='u'))

Assuming that CreditCards is the first table in our database, the attacker query would return the following error message.

"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."

Union based SQIA

In this method, the UNION keyword is used to change the dataset returned for a given query. Attackers join the injected query to the original query by using the UNION keyword.

Example:

The query used from [7][9][10][12] is to retrieve employee name from the database.

Intended query: Select name from Employees where EmployeeID=1234
Attacker query: Select name from Employees where EmployeeID=1234 UNION
Select name from SYSOBJECTS.

The attacker query retrieves the name of all the tables available in the database.

Piggy-Backed Queries

In this type of attack, the attackers inject additional queries to the original queries. The intentions of the attacker are not to modify the original query but to include new queries that "piggy-backs" on the original query. This results in SQL server executing multiple unintended queries.

Example:

The query used in this example [7][9][10][11] is to retrieve employee name from the database.

Intended query: Select name from Employees where EmployeeID=1234
Attacker query: Select name from Employees where EmployeeID=1234;
DROP TABLE Employee;

The attacker query has an additional drop statement in it. When SQL server executes the attacker query, the Employees table will be dropped.

Stored Procedures

One of the most common misconceptions is that using stored procedures in web application can avoid SQLIA. Stored procedures are as vulnerable as normal queries [18][24]. In addition, as stored procedures are written in special scripting languages, they contain additional vulnerabilities such as buffer flows, which allows attackers to un arbitrary code on the server [9].

The following example from [7][8][9][10][11][13][14][15][16] is a stored procedure that is used to authenticate a user. The user provides username, password and pin for authorization.

```
CREATE PROCEDURE DBO.isAuthenticated @userName varchar(20), @pass varchar(10),
@pin int
```

AS

EXEC("SELECT accounts FROM users WHERE login='" +@userName+ "' and pass='" +@password+ "' and pin=" +@pin);

GO

The attacker injects SHUTDOWN in either the username or password field and the query is changed to

Attacker query: SELECT accounts FROM user WHERE login='abc' AND pass=' ';

SHUTDOWN; -- AND pin=

Inference

In this type of attack, the attackers inject the query to understand the behavior of the web site. These types of attack are carried on secure websites, where the error messages from databases are blocked. Attackers deduce that certain parameters in the web site are vulnerable, by carefully the noting the changes in the behavior of the web site. There are two types of Inference attacks:

Blind SQL Injection Attack:

In this technique, the attacker asks server true/false questions to understand the behavior of a web page. The web page acts normally, when the injected statements evaluates to true. The web page differs significantly from normal behavior, when the injected statements evaluate to true. The following example explains Blind SQLIA.

Example:

The query used in this example is used to authenticate a user. The user provides username, password and pin for authorization.

Intended query: SELECT account from users WHERE login='abc' and password = ' ' and pin =0

The attacker wants to identify injectable parameters. In our example we assume that the attacker is trying to inject the login field. The following are the two possible injections into the login field representing true/ false. The first being

"legalUser' and 1=0 - -" and the second, *"legalUser' and 1=1 - -"*.

Attack Query: SELECT accounts FROM users WHERE login='legalUser' and 1=0 -- ' AND pass='' AND pin=0

SELECT accounts FROM users WHERE login='legalUser' and 1=1 -- ' AND pass='' AND pin=0.

There are two possible scenarios:

- Login input field is validated:

In this scenario, both the injections mentioned above will return login error messages. Thus attacker would know that login field is not vulnerable.

- Login input field is not validated:

In this scenario, the first injection which always evaluates to false will always result in login error messages from the application. But the second injection which always evaluates to true will not return login error messages. Thus attacker identifies that login field is vulnerable to attacks.

Timing Attacks:

Timing attacks is based on inference. Here the attacker uses the time taken by a query to execute in the server, to gain information about the database. A timing attack is performed by forming an injected if/then statement. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute. By measuring the increase or decrease in response time that attacker can infer which branch was taken in SQL injection.

Example:

The query used in this example[10][17][18] is to is used to authenticate a user. The user provides username, password and pin for authorization.

Intended query: SELECT account from users WHERE login='abc' and password = ' ' and pin =0

Attacker query: SELECT accounts FROM users WHERE login='abc' and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ' AND pass='' AND pin=0.

In the attacker query, substring keyword is used to extract the first character of the first table's name. The attacker in this is asking if the ASCII value of the character is greater than or less than the value of x. If the value is greater than x, the attacker observes a 5 second delay in the response of the database. The attacker then uses binary search strategy to identify the value of the first character.

Alternate Encodings

This type of attack is to overcome the defensive coding practices. Alternate encoding is used in conjunction with other attacks. Defensive coding practices scans for certain known "bad characters" to detect SQLIA. To evade this method the injected query string is encoded using hexadecimal, ASCII and Unicode etc.

Example:

The query used in this example [7][10][11] is to is used to authenticate a user. The user provides username, password and pin for authorization.

Intended query: SELECT account from users WHERE login='abc' and password = ' ' and pin =0

Attacker query: SELECT accounts FROM users WHERE login='legalUser';
exec(char(0x73687574646f776e)) -- AND pass='' AND pin=

The attacker query encodes the string value SHUTDOWN before injecting it in the query.

Second Order Injection

In this type of attack the injected query is stored as data in the database and is executed when the data is requested in the search query. This is a two level attack. First the attacker inserts a harmful query into the database as data.

Example:

Attacker query: Insert into Users (name,age,profile) Values('abc',32,''';
DELETE FROM USERS;--')

When this query [10] is executed, the attacker has successfully inserted the query string "DELETE FROM USERS" into the Users table.

The stored query string does not cause any problems until it is retrieved by another query. Once the harmful stored query string is retrieved, the SL server considers it as separate query statement and executes it.

Intended query: Select * from Users where profile = {0}

Attacker query: Select * from Users where profile=''; DELETE FROM USERS;

Existing Detection and Prevention techniques of SQLIA

The following are the existing techniques that are used in detecting and preventing SQLIA. These techniques range from development best practices to fully automated frameworks.

Defensive coding practices

SQL Injection is caused by lack of input validation. There are various defensive coding practices [7][10] to prevent SQLIA.

a) Input type checking: Developers need to check the input provided in the web application. The checks can be avoiding character in the numeric field, providing length constraints etc.
b) Encoding of inputs: The Meta characters can be specially encoded and interpreted by database as normal characters.
c) Positive pattern matching: A safer way to check inputs. Developers identify good inputs as opposed to bad inputs. Developers specify all the forms of legal input.
d) Identification of all input sources: Identifying all the input sources to the application is significant. When used to construct a query, these input sources are used by attackers to insert SQLIA.

Black Box Testing

WAVES [19], proposed by Huang and colleagues is a black box testing technique for testing web applications for SQLIA. A web crawler is used to identify the points in web application which can be used to inject SQLIA. Using a specific list of patterns and attack techniques, WAVES attack the target points identified. WAVES then monitors the applications response to the attacks and uses machine learning techniques to improve its attack methodology.

Static Code Checkers

JDBC-Checker [20][21] checks for type correctness of dynamically generated SQL queries. It is able to detect improper type checking of input. This technique was not developed to detect and prevent SQLIA, but can be used to prevent attacks that take advantage of type mismatches in a dynamically generated query string. The disadvantage of this technique is that most of the attacks consist of syntactically and type correct queries.

Combined Static and Dynamic Analysis

AMNESIA [22][23], uses static analysis and run time monitoring to detect SQLIA. During the static analysis phase, AMNESIA builds a model to represent the structure of the query. Before queries are sent to the database, they are validated against the model built during the static analysis phase. Queries that violate the models are considered as attack queries.

SQL Guard [24] and SQL-Checker [25] also builds a model and check if queries conform to the designed model. The model is built based on a grammar that only accepts legal queries. In SQL Guard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQL Check, developer specifies the model independently. In both these approaches a secret key is used to delimit the user input. Thus the security of these approaches depends on the ability of attacker identifying the key.

Taint Based Approaches

Information flow analysis is used by WebSSARI [26] to detect input validation errors. Static analysis is used to check taint flows against preconditions for sensitive functions. The points that have not met the preconditions are detected and filters are automatically added to the application to satisfy the preconditions. Another approach that uses information flow analysis to detect vulnerabilities in software was proposed by Livshits and Lam. The basic approach is to detect when tainted input is used to construct a SQL query.

Dynamic taint analysis are proposed by researches as well. Approaches proposed by Nguyen- Tuong [27] and colleagues and Pietraszek and Berghe [28] are similar and use a context sensitive analysis to detect and reject queries, if untrusted input has been used to create certain types of SQL tokens. These approaches modify a PHP interpreter to track precise per-character taint information. A similar approach for java was proposed by Haldar and Colleagues [29] and Securifly [30]. These approaches track taint information on a per-string basis.

New Query Development Paradigms

SQL DOM [31] and Safe Query Objects [32] provide a safe and reliable approach to access databases. A type checked API is used in query building process. Within the API, systematic ding best practices are applied. By changing the environment in which the queries are built, these techniques eliminate the coding practices that make most SQLIA possible.

Intrusion Detection Systems

Intrusion detection system was used to detect SQLIA by Valeur and colleagues [33]. The IDS is built based on a machine learning technique that is trained using a set of typical application

queries. Models of queries are built and monitored during run time to identify queries that do not satisfy the model.

Proxy filters

Security Gateway [34] is a proxy filtering system that enforces input validation rules on the data flowing to a web application. Developers use Security Policy Descriptor Language (SPDL) to provide constraints and specify transformations to be applied to application parameters as they flow from web page to the application server.

Instruction Set Randomization

SQLrand [35] is an approach based on instruction set randomization. It allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter de-randomizes the keywords by intercepting the queries to the database. When an attacker injects SQL code it would not adhere to the randomized instruction set. Table 2.1 compares the detection based techniques with respect to attack types.

Table 2.1: Comparison of detection based techniques with respect to attack types

Technique	Taut.	Illegal/ Incorrect	Piggy-Back	Union	Stored Proc	Infer	Alt. Encodings
AMNESIA	•	•	•	•	×	•	•
CSSE	•	•	•	•	×	•	×
IDS	○	○	○	○	○	○	○
Java Dynamic Tainting	–	–	–	–	–	–	–

Table 2.1 continued

SQL Guard	•	•	•	•	×	•	•
SQLrand	•	×	•	•	×	•	×
Tautology Checker	•	×	×	×	×	×	×
Web App Hardening	•	•	•	•	×	•	×

• Technique can successfully detect the types of attack.

◦ does not provide guarantee of completeness.

- Technique that addresses the attack type considered only partially because of intrinsic limitations of the underlying approach.

× Technique cannot detect the type of attack.

CHAPTER 3. THE SYSTEM METHODOLOGY

Chapter 2 discussed the existing approaches for detecting and preventing SQLIA. One of the most commonly used approaches to detect SQLIA, is to build a model based on the structure of the query used in the web application. When the same query is executed, the query is validated against the model to detect SQLIA. The solution to detect SQLIA proposed in this thesis follows the approach mentioned above. Execution plans from SQL server are used to represent structure of the query i.e. are used as models.

Execution Plans

Execution plans from SQL server, tells us how a query will be executed. They are used extensively by the database administrators in optimizing SQL queries. The following are the properties of the Execution Plans that is used in detecting SQLIA:

- An execution plan contains all the objects involved in the query. The following are the significant objects that can be found in an Execution Plan.
    - Database name
    - Table name
    - Name of the Columns retrieved from the table
    - Name of the Columns used in the where clause of a query
- Defines the nature of the query. An execution plan helps us in identifying the type of DML/ DDL query that is executed.
- The where clause in an execution plan is computed/ optimized.

Execution plan represents the structure of the query. Detecting SQLIA using execution plan is based on the fact that, when an attacker inserts SQL tokens as user inputs to inject SQLIA,

the structure of the original query changes. As the structure of the original query changes, the execution plan of the original query can no longer be used to execute the query. Hence a new execution plan, representing the structure of the modified original query, needs to be generated, to execute the query.

Execution plans can be used to represent the model of input queries. Original Execution Plan acts as a model representing legal query structure. When the same query is executed, the execution plan generated could be compared with the Execution Plan of the original queries. Variations in the execution plan can be used to detect SQLIA.

Assumptions

The following are the assumptions in the Execution Plan approach in detecting SQL Injection attack.

- Execution plans are for SQL server. Hence we assume that our under lying database in SQL server. But this technique can be extended to web applications that use other databases as well. This will be discussed in detail in future work section.
- We do not assume that Execution Plan for a query will be identical every time. Execution Plan depends on lot of factors like table structure, indexing etc. Execution plan for the same query with same table structure will be different if the indexing defined in the table is different during the execution time of the query. We assume that the objects listed in the execution plan will not change for the same query. We do not compare two Execution Plans; we compare the objects involved the Execution Plans.
- We rely in the integrity of the Execution Plans retrieved from the SQL Server.

Detecting SQLIA

The following section illustrates an example of how an SQLIA is detected using Execution Plans. The query mentioned below takes an Employee ID as input and retrieves the detail of the employee from the database.

Developer Query: Select * from Employee Where EmployeeID = {0}(represents user input).

Below is an example of the query executed by a valid user.

Valid Query: Select * from Employee Where EmployeeID = 1234

Now an attacker tries to insert SQL token in user input to inject SQLIA. The query mentioned below represents an attacker query, who is trying to use 'OR' SQL Token.

Attacker query: Select * from Employee Where EmployeeID = 1234 OR 'a'='a'.

To detect such attacks, Execution Plans can be used. The following figures represent the Execution Plan of the above three queries:

a) Developer Query: When generating execution plans for the developer query, the input of the query can be any random values representing a legal state of the query. The value of the user input does not affect the Execution Plan. In the example mentioned above, the user input can be any positive integer as it represents Employee ID. The following Execution Plan is generated considering user input as integer value 0.

Select * from Employee Where EmployeeID = 0

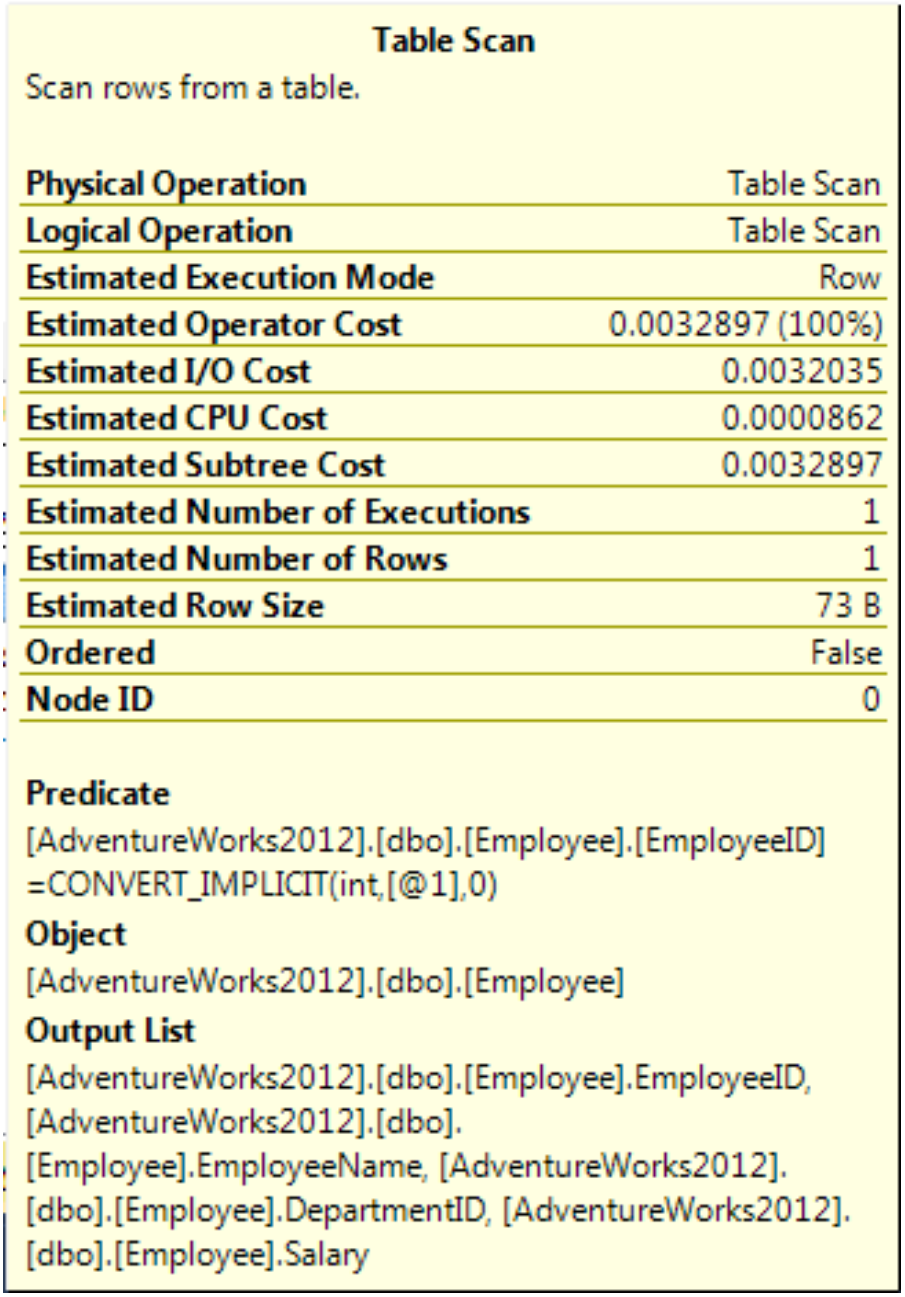

Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Estimated Operator Cost	0.0032897 (100%)
Estimated I/O Cost	0.0032035
Estimated CPU Cost	0.0000862
Estimated Subtree Cost	0.0032897
Estimated Number of Executions	1
Estimated Number of Rows	1
Estimated Row Size	73 B
Ordered	False
Node ID	0

Predicate

[AdventureWorks2012].[dbo].[Employee].[EmployeeID] =CONVERT_IMPLICIT(int,[@1],0)

Object

[AdventureWorks2012].[dbo].[Employee]

Output List

[AdventureWorks2012].[dbo].[Employee].EmployeeID, [AdventureWorks2012].[dbo].[Employee].EmployeeName, [AdventureWorks2012].[dbo].[Employee].DepartmentID, [AdventureWorks2012].[dbo].[Employee].Salary

Figure 3.1: Graphical Execution Plan for query (Select * from Employee Where EmployeeID = 0)

b) Valid query:

```
Select * from Employee Where EmployeeID = 1234
```

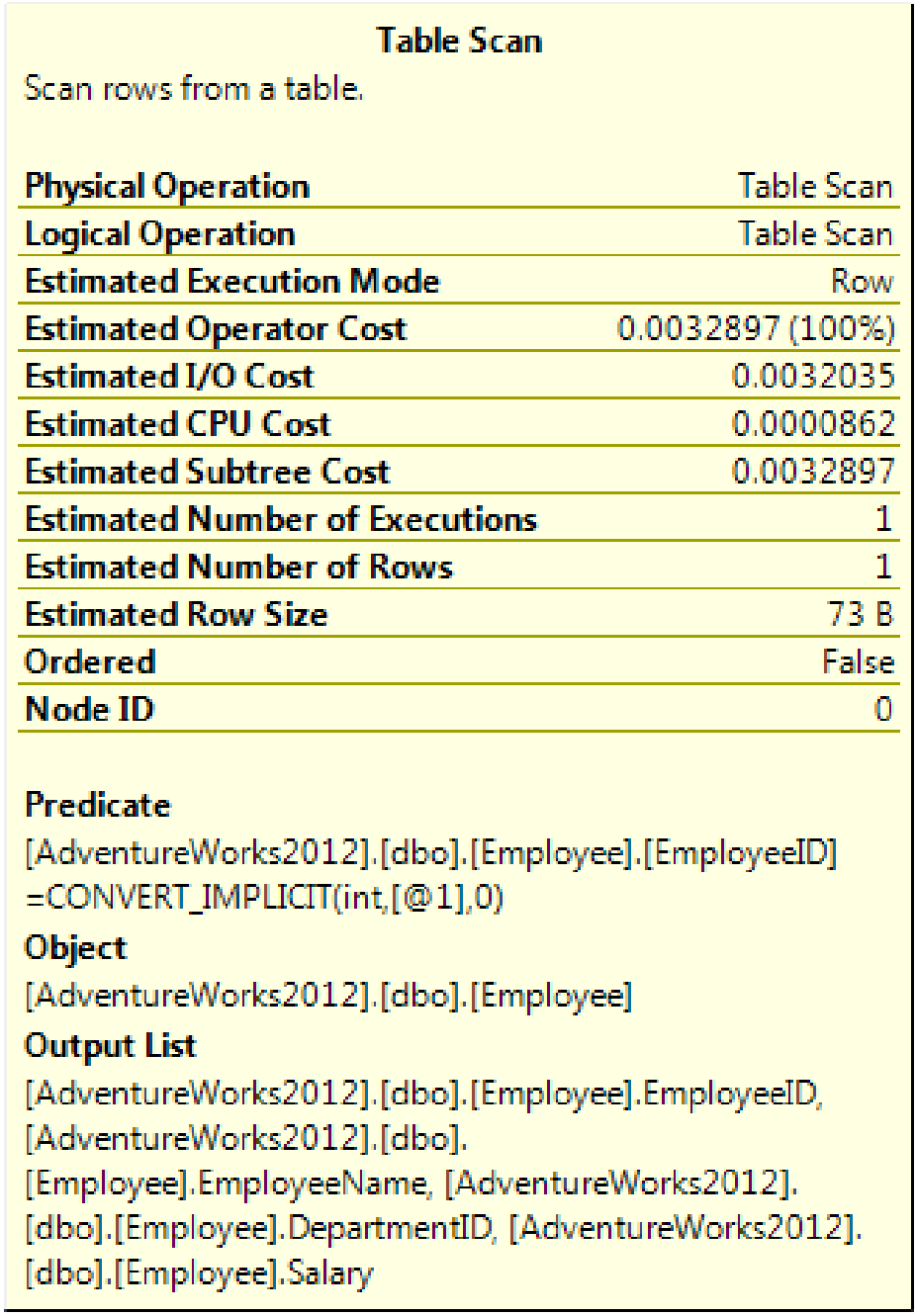

Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Estimated Operator Cost	0.0032897 (100%)
Estimated I/O Cost	0.0032035
Estimated CPU Cost	0.0000862
Estimated Subtree Cost	0.0032897
Estimated Number of Executions	1
Estimated Number of Rows	1
Estimated Row Size	73 B
Ordered	False
Node ID	0

Predicate

[AdventureWorks2012].[dbo].[Employee].[EmployeeID] =CONVERT_IMPLICIT(int,[@1],0)

Object

[AdventureWorks2012].[dbo].[Employee]

Output List

[AdventureWorks2012].[dbo].[Employee].EmployeeID, [AdventureWorks2012].[dbo].[Employee].EmployeeName, [AdventureWorks2012].[dbo].[Employee].DepartmentID, [AdventureWorks2012].[dbo].[Employee].Salary

Figure 3.2: Graphical Execution Plan for query (Select * from Employee Where EmployeeID = 1234)

c) Attacker query:

```
Select * from Employees Where EmployeeID = 1234 or 'a' = 'a'
```

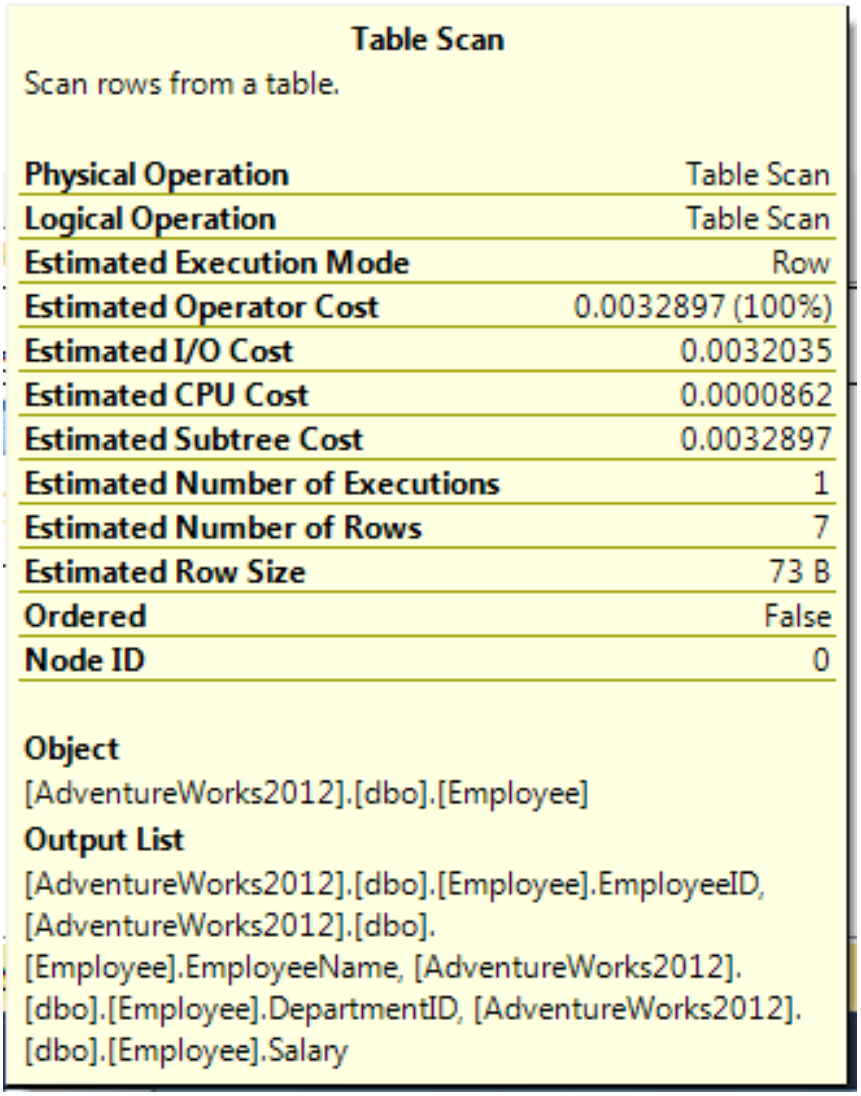

Figure 3.3: Graphical Execution Plan for query (Select * from Employee Where EmployeeID = 1234 or 'a' = 'a')

The Execution Plan of a query in general contains

- Database name
- Table name
- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

From Figures 3.1, 3.2 and 3.3 we could see that Execution Plan for our queries contains the above mentioned objects. The Object section in the Figures represents the database and table name. The Output List section represents the name of the columns retrieved by the query. The Predicate section represents the name of the columns used in the Where clause of the query.

By comparing the execution plans of the three queries we can identify that Execution Plans of developer (Figure 3.1) and valid query (Figure 3.2) are identical but Execution Plan of the attacker query (Figure 3.3) is different from the developer query. The Execution Plan of the attacker query does not have the Predicate section. This is because Execution Plans as discussed earlier, computes the Where clause section before executing the query. In the attacker query, the condition in the Where clause *Where EmployeeID = 1234 OR 'a' = 'a',* always renders to true and hence it is removed while generating Execution Plan. The original query is changed to

```
Select * from Employees;
```

Ignoring the where clause. The change in the Execution Plan represents a SQLIA.

Stages involved in detecting SQLIA

The following sections represent the stages involved in detecting SQLIA using Execution Plan approach.

Stage 1: Identifying the input source

In this stage, the web application code is analyzed to find the input sources that can cause SQLIA. Identification of all input sources is part of defensive coding practices, where the developer/ analyzer identify the places which are used by the attacker to exploit the application to SQLIA.

Stage 2: Building the model

In this stage, a model will be built to each query identified in stage 1. Execution plans of the query will be used to build the model. This model represents the structure of the query expected by the developer. This plan can be generated either manually by the developer or can be generated automatically. As the models developed are different for each query, a table structure is used to tag each query with its corresponding model.

Stage 3: Detecting Structural Changes

This stage represents the execution of the query. Each time the query is executed, its Execution Plan is retrieved from the database. This Execution Plan is compared with the Execution Plan obtained in the stage 2. Difference in the Execution Plans is reported as SQLIA.

The stages involved are explained in Figure 3.4.

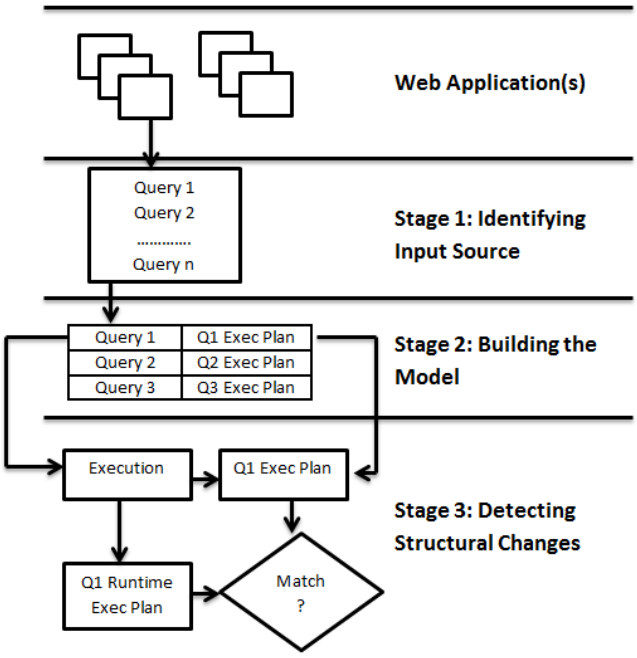

Figure 3.4: Stages involved in detecting SQLIA using Execution Plan Approach.

Execution Plan approach against SQLIA

The following section explains how Execution Plan approach is used to detect each type of SQLIA. The types of SQLIA were discussed in Chapter 2. As discussed earlier, Execution Plan approach detects the changes in the structure of the query to report SQLIA.

Tautology based SQLIA

Attack: The aim of this attack is to inject code in one or more conditional statements so that it always evaluates to true. The attacker uses SQL keywords to make conditional statements to be always evaluated to true. The use of WHERE clause present in the SQL query becomes void.

Example:

The query used in this example is to retrieve the records of an employee whose ID is 1234. The attacker using tautology based attacks retrieves information of all employees.

Intended query: Select * from Employees where EmpID = 1234 **Attacker query:** Select * from Employees where EmpID = 1234 or ‘a’=’a’

Detection: The following property of the Execution Plans is used in detecting tautology based SQLIA.

The where clause in an execution plan is computed/ optimized.

As per the property mentioned above, the attacker query in the example changes its structure to

Attacker query: Select * from Employee;

As the where clause is always true, it is removed from the query structure.

Illegal/Logically Incorrect Queries

Attack: The attacker injects statements that cause syntax, type conversion and logical errors in the database. The attacker uses the error message retrieved from the database to get significant information about the database.

Example:

The query used in this example is for an ATM login page. The user enters pin information as input to the query.

Intended query: Select * from users where pin = 1234 **Attacker query:** Select * from users where pin= convert(int,(select top 1 name from sysobjects where xtype='u'))

Assuming that CreditCards is the first table in our database, the attacker query would return the following error message.

"Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."

Detection: The following property of Execution Plans is used in detecting Illegal/ Incorrect queries.

An execution plan contains all the objects involved in the query.

The following are the significant objects that can be found in an Execution Plan.

- Database name
- Table name
- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

Based on the property mentioned above, insertion of new table names to the original query can be detected. In the example discussed, the attacker has injected new table, SYSTABLES to the query. Using the Execution Plan insertion of a new table to the query can be detected.

Union based SQIA

Attack: Attackers join the injected query to the original query by using the property of the UNION keyword.

Example:

The query used in this example is to retrieve employee name from the database.

Intended query: Select name from Employees where EmployeeID=1234 **Attacker query:** Select name from Employees where EmployeeID=1234 UNION Select name from SYSOBJECTS.

The attacker query retrieves the name of all the tables available in the database.

Detection: The following property of Execution Plans is used in detecting UNION based SQLIA.

An execution plan contains all the objects involved in the query.

The following are the significant objects that can be found in an Execution Plan.

- Database name
- Table name
- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

Based on the property mentioned above, insertion of new table names to the original query can be detected. In the example discussed, the attacker has injected new table, SYSOBJECTS to the query. Using the Execution Plan insertion of a new table to the query can be detected.

Piggy-Backed Queries

Attack: The intentions of the attacker are to include new queries that "piggy-backs" on the original query. This results in SQL server executing multiple unintended queries.

Example:

The query used in this example is to retrieve employee name from the database.

Intended query: Select name from Employees where EmployeeID=1234 **Attacker query:** Select name from Employees where EmployeeID=1234; DROP TABLE Employee;

The attacker query has an additional drop statement in it. When SQL server executes the attacker query, the Employees table will be dropped.

Detection: The following properties of Execution Plans are used in detecting Piggy-backed queries.

An execution plan contains all the objects involved in the query.

The following are the significant objects that can be found in an Execution Plan.

- Database name
- Table name
- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

Defines the nature of the query

An execution plan helps us in identifying the type of DML/ DDL query that is executed.

Based on the properties mentioned above, insertion of new table names to the original query can be detected. Also the type of DML/ DDL statements executed by the query can be identified. In

the example discussed, the attacker has injected Employee table to the original query. Only one occurrence of Employee table is allowed. Also a new DML/ DDL statement DROP is included.

Stored Procedures

***Attack*:** SQLIA carried out in stored procedure calls.

The following is a stored procedure that is used to authenticate a user. The user provides username, password and pin for authorization.

```
CREATE PROCEDURE DBO.isAuthenticated
@userName varchar(20), @pass varchar(10), @pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" +@userName+ "' and pass='" +@password+
"' and pin=" +@pin);
GO
```

The attacker injects SHUTDOWN in either the username or password field and the query is changed to

Attacker query: SELECT accounts FROM user WHERE login='abc' AND pass=' ';
SHUTDOWN; -- AND pin=

This sends shutdown signal to the database.

Detection:

An Execution Plan of a stored procedure is similar to that of a simple query statement. Query optimizer looks Stored Procedures as collection of statements, when generating Execution Plans. Thus by comparing Execution plans of compiled stored procedure and run time stored procedure, SQLIA can be detected.

In the above example, the attacker has inserted a new statement SHUTDOWN, which can be identified using Execution Plans.

Inference

Attack:

In this type of attack, the attackers inject the query to understand the behavior of the web site. There are two types of Inference attacks.

Blind SQLIA:

In this technique, the attacker asks server true/false questions to understand the behavior of a web page.

Example:

The query used in this example is used to authenticate a user. The user provides username, password and pin for authorization.

Intended query: SELECT account from users WHERE login='abc' and password = ' ' and pin =0

The attacker wants to identify injectable parameters. In our example we assume that the attacker is trying to inject the login field. The following are the two possible injections into the login field representing true/ false. The first being

"legalUser' and 1=0 - -" and the second, "legalUser' and 1=1 - -".

Attack Query: SELECT accounts FROM users WHERE login=’legalUser’ and 1=0 -- ’ AND pass=’’ AND pin=0

SELECT accounts FROM users WHERE login=’legalUser’ and 1=1 -- ’ AND pass=’’ AND pin=0.

Example:

The query used in this example is used to authenticate a user. The user provides username, password and pin for authorization.

Timing attacks:

Intended query: SELECT account from users WHERE login=’abc’ and password = ‘ ’ and pin =0

Attacker query: SELECT accounts FROM users WHERE login=’abc’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ’ AND pass=’’ AND pin=0.

In the attacker query, substring keyword is used to extract the first character of the first table’s name. The attacker in this is asking if the ASCII value of the character is greater than or less than the value of x. If the value is greater than x, the attacker observes a 5 second delay in the response of the database. The attacker then uses binary search strategy to identify the value of the first character.

Detection:

The following property of Execution Plans is used in detecting UNION based SQLIA.

An execution plan contains all the objects involved in the query.

The following are the significant objects that can be found in an Execution Plan.

- Database name
- Table name

- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

Based on the property mentioned above, insertion of new table names to the original query can be detected. In the example discussed, the attacker has injected new table, SYSOBJECTS to the query. Using the Execution Plan insertion of a new table to the query can be detected. Also insertion of new keywords such as WAITFOR can be identified in Execution Plans.

Alternate Encodings

Attack: This type of attack is to overcome the defensive coding practices. Alternate encoding is used in conjunction with other attacks. Defensive coding practices scans for certain known "bad characters" to detect SQLIA. To evade this method the injected query string is encoded using hexadecimal, ASCII and Unicode etc.

Example:

The query used in this example is used to authenticate a user. The user provides username, password and pin for authorization.

Intended query: SELECT account from users WHERE login='abc' and password = ' ' and pin =0

Attacker query: SELECT accounts FROM users WHERE login='legalUser';
exec(char(0x73687574646f776e)) -- AND pass='' AND pin=

The attacker query encodes the string value SHUTDOWN before injecting it in the query.

Detection: The encoded characters are decoded before Query optimizer tries to generate Execution Plans. Thus by observing Execution Plans, SQLIA can be detected.

Second Order Injection

Attack:

In this type of attack the injected query is stored as data in the database and is executed when the data is requested in the search query. This is a two level attack. First the attacker inserts an harmful query into the database as data.

Example:

Attacker query: Insert into Users (name,age,profile) Values('abc',32,''';DELETE FROM USERS;--')

When this query is executed, the attacker has successfully inserted the query string "DELETE FROM USERS" into the Users table. The stored query string does not cause any problems until it is retrieved by another query. Once the harmful stored query string is retrieved, the SL server considers it as separate query statement and executes it.

Intended query: Select * from Users where profile = {0} **Attacker query:** Select * from Users where profile=''; DELETE FROM USERS;

Detection: The following properties of Execution Plans are used in detecting Second Order Injection SQLIA.

An execution plan contains all the objects involved in the query.

The following are the significant objects that can be found in an Execution Plan.

- Database name
- Table name
- Name of the Columns retrieved from the table
- Name of the Columns used in the where clause of a query

Defines the nature of the query.

An execution plan helps us in identifying the type of DML/ DDL query that is executed

Based on the properties mentioned above, insertion of new table names to the original query can be detected. Also the type of DML/ DDL statements executed by the query can be identified. In the example discussed, the attacker has injected USERS table to the original query. Also a new DML/ DDL statement DROP is included.

To summarize this section, Execution Plans can be used to detect all the classes of SQLIA. The properties of Execution Plans help in identifying the structural changes of the original query. A SQLIA is detected when the structure of the query changes.

The Table 3.1 describes the Execution Plan approach of detecting SQLIA. It also relates the class of SQLIA with the property of Execution Plans that is used for detection.

Table 3.1: Execution Plan approach of detecting SQLIA

Attack Type	**Description**	**Does attack query changes structure?**	**Property(s) Of Exec Plan used in Detection**
Tautology	Inject code in one or more conditional statements so that it always evaluates to true	YES	The where clause in an execution plan is computed/ optimized.
Illegal/ Incorrect	Injected statements cause syntax, type conversion and logical errors in the database.	YES	An execution plan contains all the objects involved in the query

Table 3.1 continued

Union based		Injected query is joined to the original query by using UNION keyword	YES	An execution plan contains all the objects involved in the query
Piggy-Backed		Inject new queries that "piggy-backs" on the original query	YES	An execution plan contains all the objects involved in the query
Stored Procedure		Inject queries in stored procedures	YES	An Execution Plan of a stored procedure is similar to that of a simple query statement
Inference	Blind SQLIA	Inject true/false questions	NOT ALWAYS	An execution plan contains all the objects involved in the query
	Timing attacks	Inject if/ then statements	YES	
Alternate Encodings		Inject to overcome defensive coding practices	YES	The encoded characters are decoded before Query optimizer tries to generate Execution Plans.
Second Order Injections		Inject to store harmful queries as data.	YES	An execution plan contains all the objects involved in the query. The following are the significant objects that can be found in an Execution Plan. An execution plan helps us in identifying the type of DML/ DDL query that is executed.

CHAPTER 4. IMPLEMENTATION AND PERFORMANCE RESULTS

To gain further understanding about how Execution Plans are used to detect SQL Injection Attacks, a web application architecture was implemented using .NET framework. ASP.NET web pages were developed using C# and web services were used to represent Intrusion Detection Systems. Details of these implementations are given below, and the results of the experiments using them are presented in section 4.2.

Web Application Architecture

Figure 4.1 shows the web application architecture that is implemented to detect SQLIA using the Execution Plan approach. The technology used in implementing each component is also indicated in Figure 4.1. The communication between each component is mentioned in the body of the arrow.

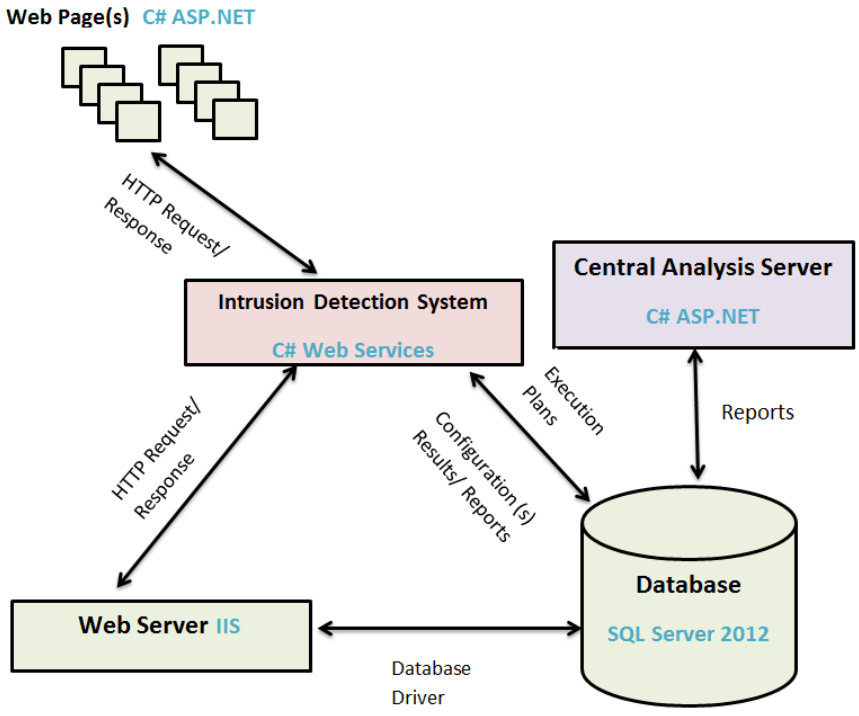

Figure 4.1: Web Application Architecture implemented to test Execution Plan Approach

Web Page(s)

The web page(s) represents the browsers which users use to access the web servers. ASP.NET web pages are developed using C# in the implementation. Web application developers define their queries to database in the web pages. Attackers use this component to break into the network.

Web Server

The web server helps to deliver the web content over a network. IIS server is used as web servers in the implementation. Web server receives requests from the web pages, processes the request and sends response to the web pages.

Database

A database is an organized collection of data. A database is used to store data. SQL Server 2012 is used as database in the implementation. The data displayed or used by the web site is stored in the database. Web application developers write queries to access data from the database. The aim of the attacker is to retrieve unprivileged data from the database or to compromise the integrity of the data stored in the database. Intrusion detection system store their configurations and reports in the database. These reports are displayed to security analysts using Central Analysis Server. IDS systems retrieve their execution plans from the database.

HTTP Request/ Response

Represents the request and respond communication between web pages and webserver.

Database Driver

Web server uses the database driver to access the data stored in the database.

Intrusion Detection System

This component is responsible for monitoring the HTTP request/ response traffic. C# web services are used to implement the detection algorithm. Execution Plan based approach is used in detecting SQLIA. The execution plans used by the IDS systems are retrieved from the database. IDS systems store their alerts and configurations in the database.

Central Analysis Server

This component is used by the security analysts to view the reports/ alerts generated by the IDS system

Performance of Execution Plan Approach

This section evaluates the performance of Execution Plan approach in detecting SQL Injections Attacks. This section also compares the effectiveness of the Execution Plan approach with other detection based approaches.

As discussed in section 4.1, web application architecture was implemented to test the Execution Plan approach for detecting SQLIA. A wide set of SQLIA data sets were used to break into the network. These data sets included all the classification of SQLIA discussed in section 2. The Execution plan based approach was able to identify all the queries that represented SQLIA with no false positive or false negatives. The Table 4.1 compares the Execution plan based approach with the other detection algorithms.

Table 4.1: Comparison of detection based techniques and execution plan approach with respect to attack types

Technique	Taut	Illegal/ Incorrect	Piggy -Back	Union	Stored Proc	Inference	Alternate Encodings	Second Order
AMNESIA	•	•	•	•	×	•	•	○
CSSE	•	•	•	•	×	•	×	○
IDS	○	○	○	○	○	○	○	○
Java Dynamic Tainting	-	-	-	-	-	-	-	×
SQL Check	•	•	•	•	×	•	•	×
SQL Guard	•	•	•	•	×	•	•	×
SQLrand	•	×	•	•	×	•	×	×
Tautology Checker	•	×	×	×	×	×	×	×
Web App Hardening	•	•	•	•	×	•	×	×
Execution plan approach	•	•	•	•	•	•	•	•

• Technique can successfully detect the types of attack.

◦ does not provide guarantee of completeness.

- Technique that addresses the attack type considered only partially because of intrinsic limitations of the underlying approach.

× Technique cannot detect the type of attack.

Advantages of Execution Plan Approach

This section discusses the advantage of execution plan approach over other detection approaches in detecting SQLIA.

Well-Developed Execution Plans

The Execution Plans used in SQL Server are well-developed. They have been used extensively by the database administrators in optimizing SQL queries. The Execution Plans can be retrieved in three formats

- XML
- Text
- Graphical

Lowest Level Detection

The Execution Plan approach detects SQLIA after the Execution Plan of a query is generated. Execution Plans are generated just before a query is executed. This can be considered as the lowest level of a database driven web application. Thus Execution Plan approach detects SQLIA at the lowest possible level.

Secure model

Execution Plans approach can detect SQLIA just by using the schema of the database. The organization can just share their schema (not data) to the security team while implementing the Execution Plan approach.

Stored Procedures

Execution Plan approach detects SQLIA effectively in the stored procedures. Detecting SQLIA in stored procedures has been a challenge to researchers. The detection approaches mentioned in section 2 have either failed or partially detected SQLIA in stored procedures.

Second Order SQLIA

Execution Plan approach detects second order SQLIA, where the attacker stores SQL statements in the database which when executed causes harmful transactions.

CHAPTER 5. CONCLUSION AND FUTURE WORK

Conclusion

In this thesis we have presented the Execution Plan based approach for detecting SQL Injection Attacks in SQL Server databases. This approach uses the Execution Plans generated in SQL Server to detect SQLIA.

We have shown, through the implementation and results, that Execution Plans can be used to detect the changes in the structure of the query, which in turn can detect SQLIA. The Execution Plan approach is capable of detecting all the classes of SQLIA including the class of second order attacks. The Execution Plan based approach detects SQLIA in the stored procedures effectively. The well-developed model of Execution Plans not only aids in the consistency of the detection algorithm but also reduces the complexity of the detection algorithm.

The Execution Plan approach detects SQLIA at the lowest possible level i.e. after Execution Plan is generated for a query. The Execution Plan algorithm just uses the schema of the database to detect SQLIA. Hence organizations need not share their data while implementing the Execution Plan based approach. The Execution Plan approach does not depend on the language in which the web site is developed as it detects SQLIA from the database.

The false positive and false negative rates can be improved widely using the Execution Plan approach. Even though Execution Plans are pertinent only to SQL Server databases, the concept of detecting SQLIA using the plan developed by the query optimizer can be implemented in any database.

Future Work

As noted above, the Execution Plan approach can be only used in detecting SQLIA in SQL Server databases. This approach can be extended to detect SQLIA in other databases as well.

Finding alternates of Execution Plans

Although the Execution Plans are used only in SQL Server databases, equivalent of Execution Plans can be found in other databases. Every database server prepares a query plan before executing the query. This query plan can be used in detecting SQLIA. For e.g. Oracle creates Explain Plan that represent the query plan developed by the query optimizer.

Unifying the query plans

Even though all the databases generate query plans before executing the query, the properties of each of these plans may differ. Also the ease of access of the effectiveness of each of these plans may differ. If the approach of generating query plans from the databases can be unified, the approach of detecting SQLIA can be unified.

Updating the Execution Plans

The Execution plans can be updated with more parameters that can be used in detecting SQLIA.

References

[1] Web application http://en.wikipedia.org/wiki/Web_application

[2] OWASP https://www.owasp.org/index.php/Top_10_2013-A1-Injection

[3] SQL injection attacks rise sharply in second quarter of 2012. http://www.computerweekly.com/news/2240160266/SQL-injection-attacks-rise-sharply-in-second-quarter-of-2012

[4] O. Maor and A. Shulman. SQL Injection Signatures Evasion.

http://www.imperva.com/application defense center/white papers/sql injection signaturesevasion.html

, April 2004. White paper

[5] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters.

http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php, April 2004. White paper

[6] SecuriTeam. SQL Injection Walkthrough.

http://www.securiteam.com/securityreviews/5DP0N1P76E.html,

May 2002. White paper.

[7] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[8] D. Litchfield. Web Application Disassembly with ODBC Error Messages. Technical document, @Stake, Inc., 2002.

http://www.nextgenss.com/papers/webappdis.doc.

[9] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.
http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php

[10] A Classification of SQL Injection Attacks and Countermeasures, William G.J. Halfond, Jeremy Viegas, and Alessandro Orso College of Computing Georgia Institute of Technology {whalfond|jeremyv|orso}@cc.gatech.edu, 2006.

[11] M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.

[12] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., 2002.
http://www.spidynamics.com/assets/documents/WhitepaperSQLInjection.pdf.

[13] F. Bouma. Stored Procedures are Bad, O'kay? Technical report,Asp.Net Weblogs, November 2003. http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx.

[14] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical report, Argeniss Information Security, Black Hat Briefings, Black Hat USA, 2005.

[15] P. Finnigan. SQL Injection and Oracle - Parts 1 & 2. TechnicaReport, Security Focus, November 2002.

http://securityfocus.com/infocus/1644

http://securityfocus.com/infocus/1646.

[16] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January 2005.

http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp.

[17] C. Anley. (more) Advanced SQL Injection. White paper, Next Generation Security Software Ltd., 2002.

[18] K. Spett. Blind sql injection. White paper, SPI Dynamics, Inc., 2003.

http://www.spidynamics.com/whitepapers/BlindSQLInjection.pdf.

[19] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003

[20] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) – Formal Demos, pages 697–698, 2004.

[21] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pages 645–654, 2004.

[22] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[23] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA

2005), pages 22–28, St. Louis, MO, USA, May 2005.

[24] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[25] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.

[26]Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[27] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.

[28] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.

[29] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.

[30] M. Martin, B. Livshits, and M. S. Lam. Finding Application Erroand Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and application (OOPSLA 2005), pages 365–383, 2005.

[31] R. McClure and I. Kruger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

[32] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed

Objects as Remotely Executable Queries. In Proceedings of the 27tInternational Conference on Software Engineering (ICSE 2005), 2005.

[33] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.

[34] D. Scott and R. Sharp. Abstracting Application-level Web Security. In Proceedings of the 11 International Conference on the World Wide Web (WWW 2002), pages 396–407, 2002.

[35] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302, June 2004.