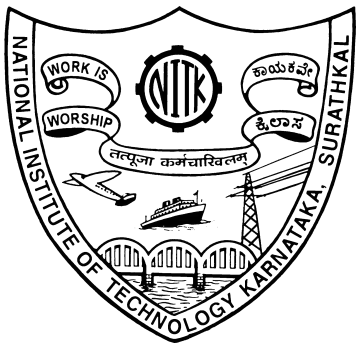
CS850 DATABASE SECURITY

A REPORT ON THE PROJECT ENTITLED

"SECOND-ORDER SQL INJECTION DETECTION & MITIGATION"



SUBMITTED BY

VIVEK JAIN - 202IS024
SHIVAM PANDEY - 202IS029

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE -575025

2020-2021

Contents

Chapter 1

Abstract

A SQL injection is a web application vulnerability that arises when user-controllable data is incorporated into database SQL queries in an unsafe manner. SQL injection may result in the application's back end database server either surrendering confidential data to the attacker or causing the execution of malicious scripting content on the database that could result in the attacker taking control of the database server, which may lead to modifying critical application data, interfering with application logic or escalating privileges within the database.

In this project, we are developing a method to detect the second-order SQL injection attacks based on ISR(Instruction Set Randomization). The method randomizes the trusted SQL keywords contained in web applications to dynamically build new SQL instruction at the beginning. And at the end, it is detected whether the received SQL instruction contains standard SQL keywords to find attack behaviour. Research shows that this system can effectively detect second-order SQL injection attack and has a low processing cost.

Chapter 2

Introduction

A SQL injection can be either first-order SQL injection or second-order SQL injection. A first-order SQL injection occurs when a web application fails to sanitize the user input. In that case, the attacker enters a malicious string and commands it to be executed immediately. The first-order SQL injection can result in copying, modifying or deleting the database by the attacker. An attacker can also modify cookies to poison a web application's database query with the help of first-order SQL injection.

A second-order SQL injection arises when user-supplied data is stored by the application and later incorporated into SQL queries in an unsafe way. Second-order SQL injection is a serious threat to the web application, and it is more difficult to detect than first-order SQL injection. The attack payload of second-order SQL injection is from untrusted user input and stored in a database or file system, the SQL statement submitted by the web application is usually dynamically assembled by a trusted constant string in the program and untrusted user input, and the DBMS is unable to distinguish the trusted and untrusted part of a SQL statement.

We divided our project into five phases. In each phase we performed a sub-task that contributed to our project. In the first phase we took a web application which was vulnerable to first-order SQL injection and then we successfully performed first-order SQL injection on the web application.

In the second phase, we improved the web application by mitigating first-order SQL injection and hence preventing the web application from such kind of SQL injection attacks. For this purpose, we wrote a safe script which uses the parameterized SQL queries. The parameterized queries forces us to define the SQL query and use placeholders for user-provided variables in the query. After the SQL statement is defined, we can pass each parameter to the query. This allows the database to distinguish between the SQL command and data supplied by a user. If an attacker inputs SQL commands, the parameterized query treats them as untrusted input and the database does not execute injected SQL commands. If we properly parametrize SQL queries, all user input that is passed to the database is treated as data and can never be confused as being part of a command.

In the third phase, we performed second-order SQL injection attack on the web application. A second-order SQL injection arises when user-supplied data is stored by

the application and later incorporated into SQL queries in an unsafe way. So, in order to perform second-order SQL injection we stored an unsafe SQL query in the database with the help of one script and then executed the query with the help of another script which resulted in dropping one of the tables from the database. In this way, second-order SQL injection was performed on the web application.

In the fourth phase, we mitigated the second-order SQL injection attack and hence prevented the web application from such kind SQL injection attacks. Here also, we used the parametrized queries for the mitigation. The parameterized queries are the prepared statements with the variable bindings. Parameterized queries forces us to first define all the SQL code, and then pass in each parameter to the query later. This allows the database to distinguish between code and data at the time of SQL query execution, regardless of what user input is supplied.

In the final phase of our project we performed second-order SQL injection detection with the help of Instruction Set Randomization(ISR). This method randomizes the trusted SQL keywords contained in web applications to dynamically build new SQL instruction at the beginning. And at the end, it is detected whether the received SQL instruction contains standard SQL keywords to find attack behaviour.

Chapter 3

Existing System

There has been a lot of work done the field of mitigating or preventing SQL injection attacks. There are many ways to prevent SQL injection attacks like use of prepared statements(parameterized queries), use of stored procedures, input validation, escaping user supplied input etc. But very less work has been done in order to detect attacks like second-order SQL injection. So, in this project we are using an existing technique i.e. parameterized queries to stop first-order and second-order SQL injection as well we are implementing a new technique i.e. Instruction Set Randomization(ISR) to detect second-order SQL injections.

Chapter 4

Proposed System

4.1 Mitigating SQL Injection Attacks

In this project, the mitigation of SQL injection attacks was performed with the help of parameterized queries(also known as prepared statements).The parameterized queries are the prepared statements with the variable bindings. The parameterized queries forces us to define the SQL query and use placeholders for user-provided variables in the query. After the SQL statement is defined, we can pass each parameter to the query. This allows the database to distinguish between the SQL command and data supplied by a user. If an attacker inputs SQL commands, the parameterized query treats them as untrusted input and the database does not execute injected SQL commands. If we properly parametrize SQL queries, all user input that is passed to the database is treated as data and can never be confused as being part of a command.

4.2 Detecting Second-Order SQL Injection

Detecting Second-Order SQL Injection in the project was performed with the help of Instruction Set Randomization(ISR). This method randomizes the trusted SQL keywords contained in web applications to dynamically build new SQL instruction at the beginning. And at the end, it is detected whether the received SQL instruction contains standard SQL keywords to find attack behaviour.

Chapter 5

Implementation Details

5.1 First-Order SQL Injection Attack

A first-order SQL injection occurs when a web application fails to sanitize the user input. In that case, the attacker enters a malicious string and commands it to be executed immediately. The first-order SQL injection can result in copying, modifying or deleting the database by the attacker.

5.2 First-Order SQL Injection Mitigation

In order to mitigate first-order SQL injection, we wrote a safe script which uses the parameterized SQL queries. The parameterized queries forces us to define the SQL query and use placeholders for user-provided variables in the query. After the SQL statement is defined, we can pass each parameter to the query. This allows the database to distinguish between the SQL command and data supplied by a user. If an attacker inputs SQL commands, the parameterized query treats them as untrusted input and the database does not execute injected SQL commands. If we properly parametrize SQL queries, all user input that is passed to the database is treated as data and can never be confused as being part of a command. ‘

5.3 Second-Order SQL Injection Attack

A second-order SQL injection arises when user-supplied data is stored by the application and later incorporated into SQL queries in an unsafe way. Second-order SQL injection is a serious threat to the web application, and it is more difficult to detect than first-order SQL injection. The attack payload of second-order SQL injection is from untrusted user input and stored in a database or file system, the SQL statement submitted by the web application is usually dynamically assembled by a trusted constant string in the program and untrusted user input, and the DBMS is unable to distinguish the trusted and untrusted part of a SQL statement.

Github Repository : `https://github.com/nxzshivam/Second-Order-SQLi`

5.4 Addressing Second-Order SQL Injection Attack

5.4.1 Mitigation

The second-order SQL injection is also mitigated with the help of parameterized queries.

5.4.2 Detection

The second-order SQL injection attack was detected with the help of Instruction Set Randomization(ISR). This method randomizes the trusted SQL keywords contained in web applications to dynamically build new SQL instruction at the beginning. And at the end, it is detected whether the received SQL instruction contains standard SQL keywords to find attack behaviour.

Chapter 6

Advantages

- The major objective of preventing or mitigating any kind of attack is to safeguard our data and prevent it from any kind of modification.
- In this project, we implemented a model which will prevent any attacker from copying, modifying or deleted our data without our consent.
- We also implemented a method to detect second-order SQL injection attacks which will help us if we want to investigate any second-order SQL injection attack.
- Finally, we developed a model which will prevent our data from first-order SQL injection attacks as well as second-order SQL injection attacks and will also help us to detect second order SQL injection attacks at a low cost.
- Our model is less complex with low implementation cost.

Chapter 7

Limitations

7.1 Computational Complexity

The computational complexity of our model is a bit more and the model will require more time if a process requires to execute a large number of queries. It is because during the second-order SQL injection detection after instruction set randomization, string matching is performed in order to check whether the resultant query is safe or unsafe.

7.2 Misclassification

There may be a case in which it required to enter a input value which matches the SQL keyword. In that case, the query will be misclassified as the unsafe one because once we randomize the parameterized query and then we pass the input values into the query and perform string matching then the input value will match with the SQL keyword and will show that the query is unsafe even when it is not so.

Chapter 8

Future Work

The future work regarding this project can be done in order to address the limitations of the project as mentioned in the previous chapter. The first thing that could be done is to decrease the computational complexity of the project by finding a way to optimise string matching process.

Another thing that could be done is to address the case when misclassification happens i.e. when user input contains the SQL keyword and there is no intention of attack. In that case, it is required that the query should be treated as a safe query but it is not so. Hence, it can also be improved in such a way that the system can efficiently handle these cases too.

Chapter 9

Conclusion

In this project, at first we mitigated first-order and second-order SQL injections and also we implemented a method for detecting second- order SQL injection attacks. The implementation process shows that this system has a good effect on preventing SQL injection and low running cost. The system does not rely on any database or server platform and is easy to implement.