Submitted: 10/5/2021

SQL INJECTION ATTCKS ON WEBSITES

OFFENSIVE HACKING: TACTICAL & STRATERGIC

ASSIGNMENT 1

SLIIT

Trinity Melder

MS19817286

ABSTRACT

Nowadays, Web applications occupies a great role in the development of the any sector. The use of web application has reached its maximum, as anything that exists, has definitely got a website. An attack of SQL injection creates menace for the safety of a website applications, as any approach towards a web application might give the attackers unobstructed access to databases that contain sensitive information. Not only does this attack enforce a leakage, but also create a vulnerability of exposing the future systems that contains the database.

This script is to provide an overview to the SQL Injection attacks that along the way would discuss planned replicas to halt SQL Injections, provide a detail on numerous categories of SQL injection outbreaks and some preclusion methods. The discussion is elaborated on covering up all the aspects that one should know about SQL Injections.

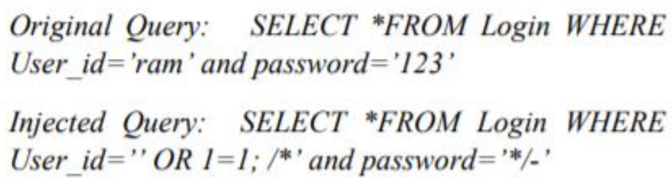
KEYWORDS*:* *SQL Injection, database security, stored procedures, detection, prevention*

1. INTRODUCTION

A "SQL injection" is a form of "security flaw" that affects website applications that bind to databases. The attacker inserts a malicious SQL query in this attack to exploit data or even obtain admission to the "back-end" of the databases via website app. The most common web server flaw is SQL injection [1]. Most of the time, this deficiency is caused by flaws in source code.

Another cause of this vulnerability may be a programming language flaw or a lack of the validation of the input. An effective injection outbreak can edit, change, or even erase the data in the databases, reading of confidential info out of databases, and even execute administrative tasks on the database, such as shutting down the database management system (DBMS).

For example:



*Original Query: SELECT *FROM Login WHERE User_id='ram' and password='123'*

*Injected Query: SELECT *FROM Login WHERE User_id='' OR 1=1; /*' and password='*/-'*

1=1 is checked as valid in this query.

As a statement, a portion of the question is evaluated. The query is run, allowing the attacker in gaining admission to these databases. As a result, these SQL injection attacks compromise database integrity, and because SQL databases hold sensitive data, secrecy is a conventional problem associated with SQL injection vulnerabilities.

If a frail SQL query is taken into consideration to display a "user's" name and password, another "user" with no prior knowledge of the password may access the details. This flaw influences authentication. If a SQL injection loophole is successfully exploited, the permission information can be changed. Changing or deleting classified information compromises the credibility of the system. An effective approach is needed to address these issues.

The parts of this paper are listed below.

The SQL injection attack is described in section 2, the literature analysis of SQL injection identification and mitigation strategies is described in section 3, and the article is concluded in section 4.

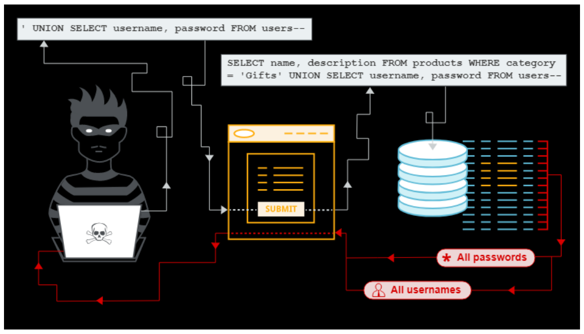

' UNION SELECT username, password FROM users--
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
SUBMIT
All passwords
All usernames

Image 1: [*] What is a SQL Injection attack

2. ATTACK ON SQL INJECTION

SQL injection attacks come in a variety of forms. The attacks described below compromise the security of web applications.

2.1 ATTACK ON TAUTOLOGY

The aim of these types of attacks is inserting a SQL query directly in a 'conditional statement' such that the considered 'all true' values are returned. This is done as simple as making the "WHERE" clause valid for any question. These famous tautology attacks are commonly taken to get through pages with authentication trails and steal vital information from databases.

*Original Query: SELECT *FROM User_info WHERE Username='Mohan' and password='133045'*

*Injected Query: SELECT *FROM User_info WHERE Username=''OR 1=1 and password='1456'*

Now an intruder will output all 'true' values for the username and password of all the kept users in the back end using this injected query.

2.2 QUESTION THAT IS LOGICALLY INCORRECT

The attacker's goal is to collect as much information as possible regarding the constructions and forms of the back end in the application. An intruder accomplishes the by using different 'manipulated' parameters in producing an 'error message' from the database. The 'table name', 'field name', and in some cases, even a 'malfunction state' are all included in the error code.

*SELECT *FROM User_unit1 WHERE Username='xyz' HAVING 1='1'; -- and password='15672'*

The columns 'User_unit1' and 'User_id' is null in the selected list as it is absently included in the 'aggregate' function or the GROUP BY CLAUSE function," according to the 'error message'. The 'table name', 'User_unit1', and the 'User_id' columns are shown in the error.

```
SELECT *FROM User_unit2 WHERE Username=
HAVING 1='1'; -- and password='15672'
```

The columns ‘User_unit2’, ‘User_id' is null in the selected list as it is absently included in ‘aggregate’ function or the GROUP BY CLAUSE function,” according to the ‘error message’. The table with the column name ‘Userinfo.Username' [2] is shown in the ‘error message’.

As a result, the intruder has access to all of the table's field names. Logical and syntactical error messages are the two kinds of error messages that can be returned. Logical errors show the names of the fields, while ‘syntactical errors’ show what paras are susceptible to the SQL injection attacks.

2.3 UNION QUERIES

The key goals of these attacks are code injection and knowledge manipulation. The operator ‘UNION’ is to retrieve intel off several DB tables for this purpose. This attacker now uses the UNION operator to join the inserted query with data from other tables in the program.

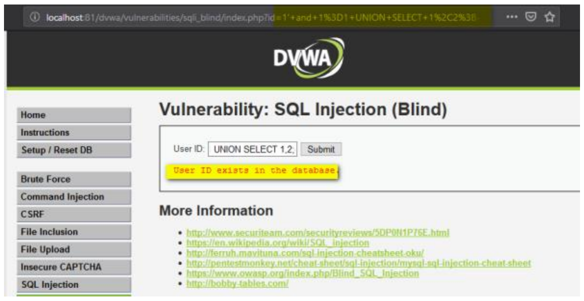
```
SELECT Name, Phone FROM Users WHERE id=$id
```

Injection of the ID values are as follows:

```
$id=UNION ALL SELECT CreditCardNumber,
1FROM CreditCardTable
```

```
SELECT Name, Phone FROM User WHERE
id=1UNION ALL SELECT CreditCardNumber, 1
FROM CreditCardTable
```

The above query will join the output of the original query with all credit card users.



localhost:81/dvwa/vulnerabilities/sqli_blind/index.php?id=1'+and+1%3D1+UNION+SELECT+1%2C2%3

Vulnerability: SQL Injection (Blind)

User ID: UNION SELECT 1,2;

User ID exists in the database.

More Information

- http://www.securiteam.com/securityreviews/5DP0N1P76E.html
- https://en.wikipedia.org/wiki/SQL_injection
- http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet
- https://www.owasp.org/index.php/Blind_SQL_Injection
- http://bobby-tables.com/

Image 2: [*] Union queries

2.4 QUERIES FOR PIGGYBACKING

The additional queries are injected into the initial query in this attack. To apply the extra query in of the initial query. An attacker uses a query delimiter, such as ‘;'. As an example:

```
SELECT accounts FROM Customers WHERE
User_id= 'ram' and password= '123'; drop table
Customers
```

Because of the “;” character, if the database allows two queries to be run in the same thread, the database admits all queries and executes them. The initial query is the one before the ‘;', and the SQL injection attack is the one after the ‘;'.

Customers tables will be lowered as the database executes the second query, resulting in the loss of useful data. They manipulate data by using the ‘INSERT, UPDATE, and DELETE’ clauses.

2.5 INFERENCE

To alter the behavior of a database or program, the two forms of attacks "Blind injection" and "Timing attack" are used in this attack.

2.5.1 BLIND INJECTION

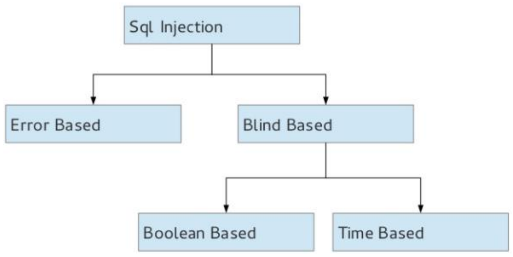


Image 3: [*] Types of SQL Injections

Developers also mask error specifics, allowing an intruder to obtain database information.

Instead of an error message, the attacker is presented with a generic website created by the developer. By running queries with a Boolean answer, an attacker can always steal data. For example:

```
SELECT accounts FROM Customers WHERE
User_id='karan' and 1=0-- AND passwd= AND
pin=0
```

```
SELECT accounts FROM Customers WHERE
User_id='karan' and 1=1-- AND passwd= AND
pin=0
```

Because of input validation, all queries would fail if the application was safe. However, if the input validation is bad, the attacker will send its initial query to obtain an 'error message' as "1=0" is not accurate. The second question is then sent. Since "1=1" is always valid, if there is no error code, the attacker looks for a field that is open to injection.

2.5.2 TIMING ATTACK

The intruder collects knowledge after looking carefully at the database suspensions in its responses in the approach. This attack is as same as to the 'blind injection' where the attacker decides if the injected argument is correct by measuring the time it takes for a page to load. To inject questions, an 'if-then' argument is used. 'WAITFORE' is a keyword that tells the back end to wait a certain amount of time before responding.

```
Declare@ varchar (7000) select @ = db_nameO if
(ascii (substring@, 1, 1)) & (power (2, 0)) > 0
waitfore delay '0:0:5'
```

The database would pause for five seconds, if the 1st bit of the 1st byte of the current database's 'name' is 1. When that state is true, the inserted code is able to produce a response time delay.

2.6 STORED PROCEDURES

Since stored procedures can be programmed by programmers, they are as injectable as web application types. Since each database has its own collection of kept measures, it's impossible for the use of them without first learning which database was used. The intruder will have the power to execute instructions on the server's OS.

2.7 ALTERNATE ENCODING

To bypass all the 'signature and filter-based checks', an intruder modifies the injection 'strings' by shifting the expression of inserted SQL queries, you can get around different detection mechanisms. As encoding methods, that can be used for this are 'ASCII', 'BASE 64', 'HEX', or 'Unicode'.

3. LITERATURE REVIEW

The SQL injection attack takes advantage of security flaws in web apps. Several methods for detecting and preventing this flaw have been suggested. Many studies have been conducted on SQL injection detection and prevention strategies, and the main findings out of the studies are summarized in the next pages.

3.1 DETECTION OF THE SQL INJECTION

The query parser approach uses Query Tokenization [3], which is introduced by the query parser method. Tokenization is done independently for the initial query and the injected query. A token is formed by the strings in front of a void, ‘single quotation’, or ‘double dashes’. Each token is an element of the array, which is generated out of the tokens. The initial and injected queries result in two distinct arrays. Both array lengths are compared for the detection of the attack of SQL injection. If the lengths of the arrays are the same, there is no SQL injection; if not, there is.

An automated solution [4] is proposed, to represent workloads that are used to validate the web server, and a vast number of ‘SQL/Xpath’ injection attacks are used to expose vulnerabilities. To detect vulnerabilities, the architecture of the SQL/Xpath commands that are given at the instance of the attack is contrasted to earlier studied commands. The ‘CIVS-WS tool’ identifies vulnerable parameters as well as all the lines that has vulnerabilities in the code.

(5) ‘Mutation’ based SQL injection vulnerability checking (MUSIC) is a Mutation based research technique that is used to evaluate SQL injection vulnerabilities. It inputs a ‘syntax error’ to see if any errors are there. After comparing the output, it will decide if the argument includes the misshapen. Only the test cases that has the injection attacks will destroy the produced mutant.

Following minimization, a ‘grammar-based algorithm’ models string values as ‘CFGs’ (Context Free Grammar) and string ‘operations’ as ‘language transducers. [6]. This technique highlights the input string and assigns labels to it, after which it operates on the input string as required. It gives the ‘direct mark’ to strings from the user directly, like ‘GET requests. It uses ‘indirect labeling’ to mark strings that come from the back end. The contexts of the named strings are found, and then the protection of each string in terms of syntax is verified using standard and context-free languages.

[7] To detect a SQL injection attack, both static and dynamic analysis are used. After eliminating the attribute value, the static SQL queries are equated to ‘dynamically’ generated queries. The feasibility of this method has been evaluated and confirmed in web apps by experiments.

[8] Runtime validation and static application code inspection are merged. The program analysis methodology is used to represent the data in the static analysis process.

SQL queries are represented as Finite State Automata in a SQL graph. The dynamically developed SQL queries are tested for compatibility with the static data structure during the runtime validation process, and they are labeled safe or unsafe. To simplify the runtime analysis, this approach does not require code change, and ‘SQL graph’ and ‘SQL database validation’ are used in parallel.

3.2 SQL INJECTION PREVENTION

Two modules are included in the negative tainting approach: the protection module and the attack database [9]. The Prevention module functions as a separate layer. This layer filters the query before sending it to the database server. This segment analyzes all queries received from the ‘application layer’. If a SQL injection is detected, the query is blocked & a warning message is sent to the application program. The question is redirected to the database server whether there is no SQL injection. The related list layout in the attack database module stores the symptoms of all documented SQL injection attacks. Symptoms are converted to tokens, which are then converted to integer numbers, which are then used to create a primary set. Similarly, the incoming question is used to build a secondary list. The secondary list and the main list are compared. It's a SQL injection attack if any matches are detected. Except for stored procedures and character encoding, this technique can stop all known attacks. Multithreading will be used in the future to will the time taken for pattern matching.

Randomized SQL queries [10] are used to determine whether a malicious statement exists before terminating them. A ‘proxy server’ is used between the ‘site server’ and the ‘database server’ to decipher the ‘random’ SQL query and then forward these decoded queries to the database for computation using the standard set of keywords. The SQL injection attack is completed by the keyword without randomization. The two primary components between the site server and the database server are the rerandomization feature and the communication protocol. Without understanding the random key, the attacker cannot perform a SQL injection attack. The best part about this method is that it has little impact on results.

[nine] To suggest an avoidance approach, static analysis and runtime control are merged. To detect the malicious query prior to it being executed, a ‘model-based’ technique is used. In the static part, a database of valid queries is built, and in the dynamic part, runtime monitoring is used to inspect dynamically generated queries, which are then compared to the statically built model. There exists no SQL injection attack if the statements satisfy the criteria, and not if the SQL statement does not fulfill the necessities.

The key tasks of this approach are identifying the hotspot, creating SQL query templates, instrumenting the program, and tracking it in real time. The outcome of this strategy demonstrates that it can stop any failed attacks. The disadvantage of this method is that it does not accept segmented queries.

For preventing authentication against SQL injection, the ‘Preventing SQL Injection Attack in Web Application’ (PSIAW) technique is proposed [12]. A login table typically has two columns: i.e., the username and password.

The hash value of the username & password is stored in two additional tables. The ‘hash value’ of the username and password is computed as one logs into the database. If they are equivalent, the user is given access to the information. The key feature of PSIAW is the SQL Query component, which calculates the hash value of the username and password. The disadvantage of the mentioned strategy is: it does not protect against any SQL injection attacks other than authentication.

A multi-level prevention strategy is developed by combining query tokenization and an adaptive approach [13]. Thinking of a methodical order of tokens of malevolent questions, static analysis of program code is used. This DB is now authenticated compared to the variations in incoming SQL query architectures during runtime prior to submitting the query to the database server for execution, to intercept all malicious SQL queries.

This approach can be used on a variety of systems that support the ASP.NET programming language.

Centered on input validation, the built methodology aids in the protection of web applications from SQL injection. The web application extracts the user's input from the created query, and then in SQL proxy-based blocker technique [14], the data is then authenticated off the generated query's syntactic perspective. This is accomplished by the application of a genetic algorithm. This method would not necessitate learning the application's source code or the authentication mechanism.

(15) One resolution for detecting and preventing SQL injection attacks is developed by combining static and runtime analysis approaches. By using a runtime analysis technique, tracking approaches using to map & record the running of all received queries. The creator prepares a list of planned modifications, which is then applied to the results of the affected items. This comparison determines whether a SQL injection attack exists. The static method compares the obtained and predicted SQL queries using a string reference. It can detect and block all kinds of SQL injection attacks, according to the results. This strategy would continue to be improved in the future by reducing the time delay.

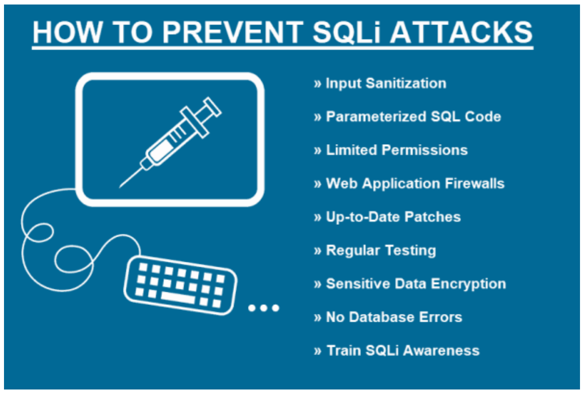


Image 4: [*] Preventing SQL Injection attack

4. CONCLUSIONS

An SQL injection assault on a web application is a very serious issue. It is critical to find an effective solution to this crisis. Many methods have been developed by researchers to identify and avoid this vulnerability. There is no one-size-fits-all solution to eradicate the SQL injection attacks. This paper examines numbers of various methods of SQL injection attacks on the back-end databases. The SQL injection identification and prevention techniques are discussed, as well as their strengths and weaknesses.

5. REFERENCES

[1] OWASP, "Top 10 2013-Top 10," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013 - Top_10.

[2] J. Fonseca, M. Vieira and H. Madeira, "Vulnerability & attack injection for web applications," in International Conference on Dependable Systems & Networks, Lisbon, 2009.

[3] L. Ntagwabira and S. L. Kang, "Use of Query Tokenization to detect and prevent SQL Injection Attacks," in International Conference on Computer Science and Information Technology , Chengdu, 2010.

[4] N. Antunes, N. Laranjeiro, M. Vieira and H. Madeira, "Effective Detection of SQL/XPath Injection

Vulnerabilities in Web Services," in International Conference on Services Computing, Bangalore, 2009.

[5] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," in The Eighth International Conference on Quality Software, Oxford, 2008.

[6] G. Wassermann and Z. Su, "Sound and Precise Analysis ofWeb Applications," in ACM SIGPLAN Conference on Programming Language Design and Implementation, California, 2007.

[7] J. G. Kim, "Injection Attack Detection Using the Removal of SQL Query Attribute Values," in International Conference on Information Science and Applications , Jeju Island, 2011.

[8] M. Muthuprasanna, K. Wei and S. Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism," in International Symposium on Web Site Evolution, Philadelphia, 2006.

[9] A. S. Gadgikar, "Preventing SQL Injection Attacks Using Negative Tainting Approach," in International Conference on Computational Intelligence and Computing Research, Enathi, 2013.

[10] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL Injection," in Applied Cryptography and Network Security, Berlin Heidelberg, Springer, 2004, pp. 292-302.

[11] W. G. J. Halfond and A. Orso, "AMNESIA: analysis and monitoring for NEutralizing SQLinjection attacks," in international Conference on Automated software engineering, New York, 2005.

[12] M. Gandhi and J. Baria, "SQL INJECTION Attacks in Web Application," International Journal of Soft Computing and Engineering (IJSCE), 2013. [13] N. A. A. Othman, F. H. M. Ali and M. B. M. Noh, "Secured web application using combination of Query Tokenization and Adaptive Method in preventing SQL Injection Attacks," in International Conference on Computer, Communications and Control Technology , Langkawi, 2014.

[14] A. Liu, Y. Yuan, D. Wijesekera and A. Stavrou, "SQLProb: a proxy-based architecture towards preventing SQL injection attacks," in ACM symposium on Applied Computing, New York, 2009.

[15] J. O. Atoum and A. J. Qaralleh, "A Hybrid Technique for SQL Injection Attacks Detection and Prevention," International Journal of Database Management Systems (IJDMS), 2014.

[*]https://www.google.co.uk/search?q=data+baselining&sxsrf=ALeKk01VOEeOl_lq72J9yKExsbh03kFfxg:1620976015209&source=lnms&tbm=isch&sa=X&ved=2ahUKEwjdmM2czsjwAhU1xDgGHV6uBosQ_AUoAXoECAEQAw&biw=1536&bih=754