Auction,
Matteo Bordignon 15/03/2024

Background

The system is vulnerable to SQL injection[1], more precisely to Inferential (or Blind) Time Based SQL injection. SQL injection attacks are a type of injection attacks, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands. It occurs when:

1. An unintended data enters a program from an untrusted source.
2. The data is used to dynamically construct a SQL query

The main consequences are:

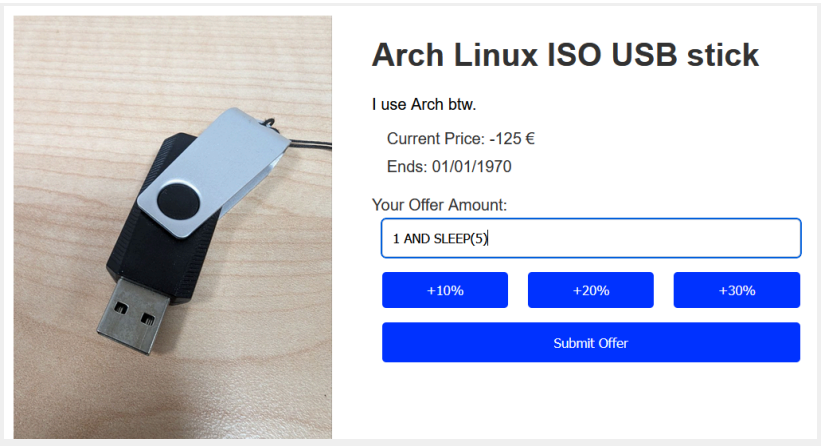
- Confidentiality: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- Authentication: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Authorization: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.
- Integrity: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

Blind SQL injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the application's response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection[2].
Time-based blind SQL injection relies on the database pausing for a specified amount of time, then returning the results, indicating successful SQL query executing.
SQL injection mitigation strategies[3]:

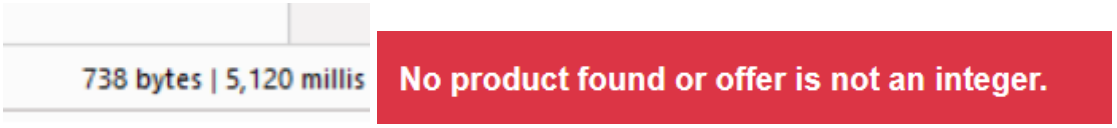
- Option 1: Use of Prepared Statements (with Parameterized Queries)
- Option 2: Use of Properly Constructed Stored Procedures
- Option 3: Allow-list Input Validation
- Option 4: STRONGLY DISCOURAGED: Escaping All User Supplied Input

Vulnerability

The vulnerability is present in the "offer" input field because there is no input validation.
Using a simple payload such as: `1 AND SLEEP(5)`

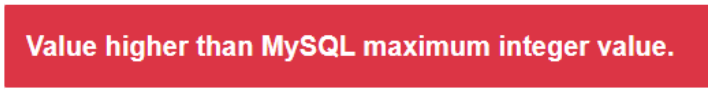


The application will execute the sleep and then return a generic '`No product found or offer is not an integer`' error.



Solution

After finding the vulnerability, my initial thought was to determine the database type and version used by the application. To do that I started to put some strange payload in the vulnerable input field and I find out that providing an input such as "100000000000000000" the application responds with the following error:



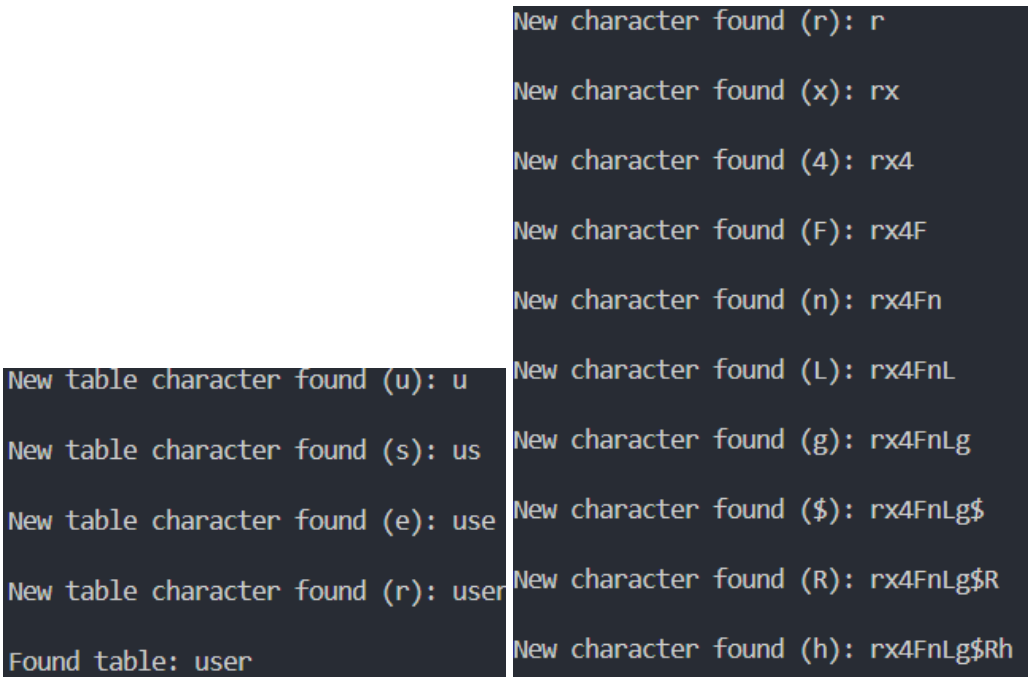
Thus I was able to discover the type of DBMS used, with this information I started searching for a query that could make me find out the name of the table where all the user credentials are stored. So I decided to use a modified version of the "blind.py" script provided during the last lesson. Using this query I was able to retrieve the table:

```
injection = f"19399 AND IF(SUBSTRING((SELECT table_name FROM
information_schema.tables WHERE table_schema=DATABASE() LIMIT
1,1),{i},1)='{character}', SLEEP(2), null)"
```

The code snippet checks if a character at a specific index in the second table name (the first was the table "products") obtained from the current database schema matches a given character. If there is a match, the code introduces a delay of 2 seconds using the SLEEP function. Then I passed the table name that I had just retrieved in a second while loop with the following query:

```
injection = f"19399 AND IF(HEX(SUBSTRING((SELECT password FROM {table}
WHERE username='admin'),{i},1))='{string_to_hex(character)}', SLEEP(2),
null)"
```

This code snippet consists of an IF statement that checks if a character in the password of the 'admin' user matches a given hexadecimal value. It uses the HEX function to convert the character at position `i` in the password to its hexadecimal representation. If the hexadecimal representation matches the specified **`string_to_hex(character)`**, the code will introduce a delay of 2 seconds using the SLEEP function; otherwise, it will do nothing.
For more information this is the link to the fully commented python script.



results of the script execution.

References

[1] OWASP SQL injection:
https://owasp.org/www-community/attacks/SQL_Injection
[2] OWASP Blind SQL injection:
https://owasp.org/www-community/attacks/Blind_SQL_Injection
[3] OWASP SQL Injection Prevention Cheat Sheet:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
[4] What is Blind SQL injection?
https://portswigger.net/web-security/sql-injection/blind
[5] SQL injection:
https://book.hacktricks.xyz/pentesting-web/sql-injection
[6] Free online query builder:
https://zzzcode.ai/sql/query-explain?id=0d892d20-6360-47ac-ac79-b8a7902e7bc5
[7] “SQLmap”: useful tool to discover SLQi vulnerabilities, I didn’t use it to do this assignment but I know that it exists:
https://sqlmap.org/