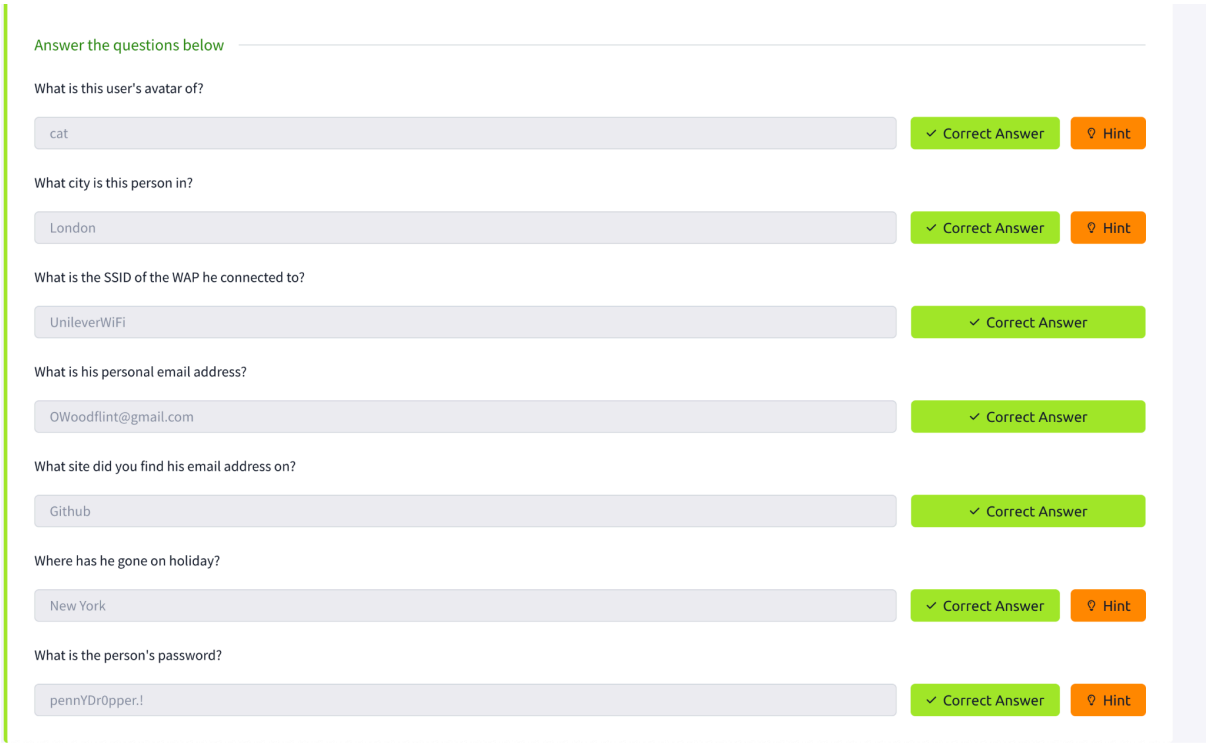
1) TryHackMe | Ohsint

Completion:



Objective:
To complete this room by using the OSINT techniques and tools taught to us.
Used google dorking to find usernames and digital footprints like his github profile.

Key Learnings:
I learned practical OSINT tricks using focused Google searches, `whois`/`dig`, and the Wayback Machine to find people and domain details. I always double-checked sources to be sure.
Learnt about Wiggle.net used for driving and scouting wifi names

Thought-Process:

1. Understand the Question.
2. Google dork for online presence (most of them from social media)
3. Checked page sources for every webpage.

2) Portswigger Labs

Lab 1-
https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data?utm_source=chatgpt.com

Lab: SQL injection vulnerability in WHERE clause allowing retrieval of hidden data

APPRENTICE

LAB ✓ Solved

This lab contains a SQL injection vulnerability in the product category filter. When the user selects a category, the application carries out a SQL query like the following:

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

To solve the lab, perform a SQL injection attack that causes the application to display one or more unreleased products.

ACCESS THE LAB

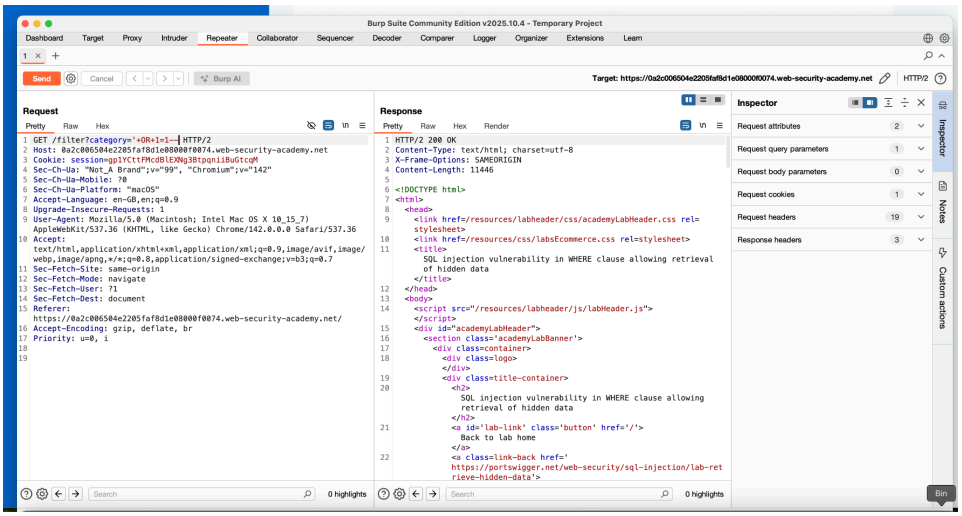
Solution

Community solutions

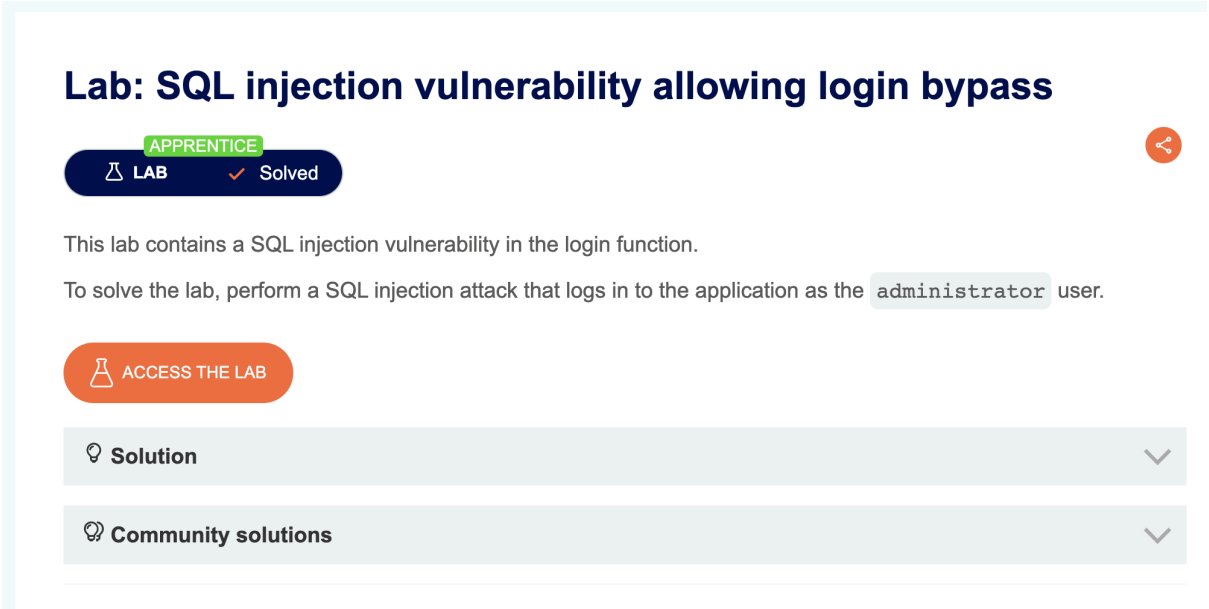
Steps performed: Accessed the lab using Burp's embedded browser and clicked a product category to capture the request.

Sent the GET request to Repeater from Burp Proxy history.

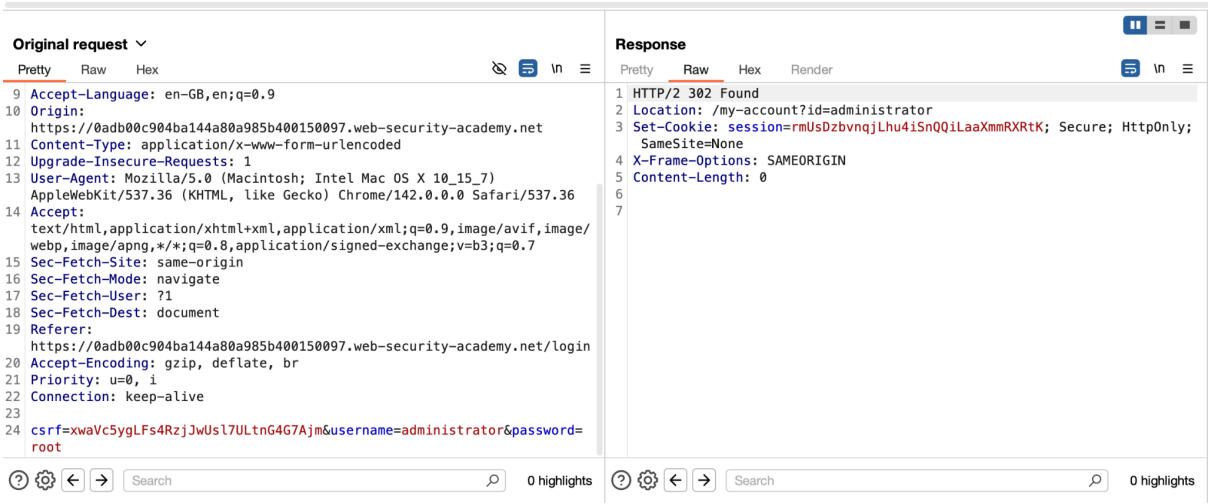
Modified the `category` parameter to:



Lab2 - https://portswigger.net/web-security/sql-injection/lab-login-bypass



1. Intercepted the `POST /login` request using Burp and sent it to Repeater.
2. Modified the username parameter to `' OR 1=1--` to bypass the password check.
3. Sent the request, received a `302 /my-account` redirect, and accessed the admin account without credentials.



Lab3 - https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text

Lab: SQL injection UNION attack, finding a column containing text

PRACTITIONER

LAB Solved

This lab contains a SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response, so you can use a UNION attack to retrieve data from other tables. To construct such an attack, you first need to determine the number of columns returned by the query. You can do this using a technique you learned in a previous lab. The next step is to identify a column that is compatible with string data.

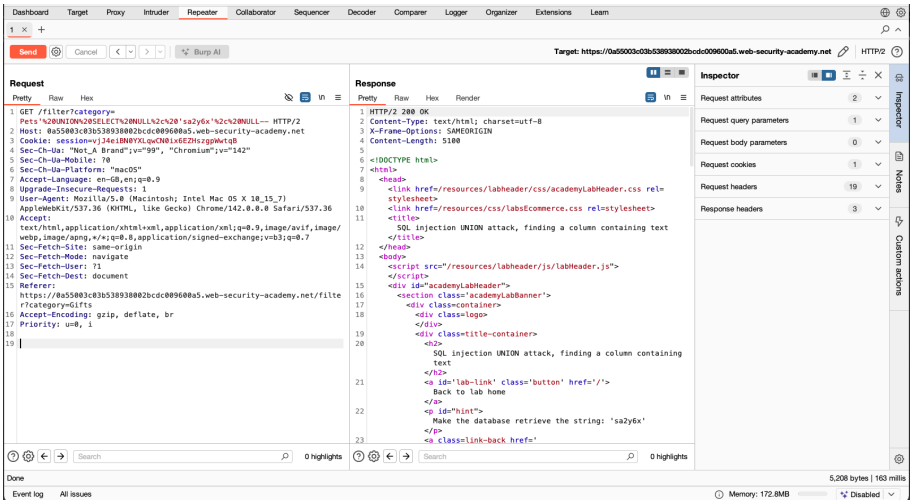
The lab will provide a random value that you need to make appear within the query results. To solve the lab, perform a SQL injection UNION attack that returns an additional row containing the value provided. This technique helps you determine which columns are compatible with string data.

ACCESS THE LAB

Solution

Community solutions

Captured the `/filter?category=` request cuz we know its vulnerable.
Created `UNION SELECT` payloads with `NULL` placeholders and replaced them with `'a'` one at a time to detect which column supports text.
Observed the string appearing on the page, confirming the text-compatible column and solving the lab.



Lab4 - https://0af400960356864f80f28596000400e2.web-security-academy.net/

Web Security Academy

SQL injection UNION attack, retrieving data from other tables

Back to lab description

Congratulations, you solved the lab! Share your skills!

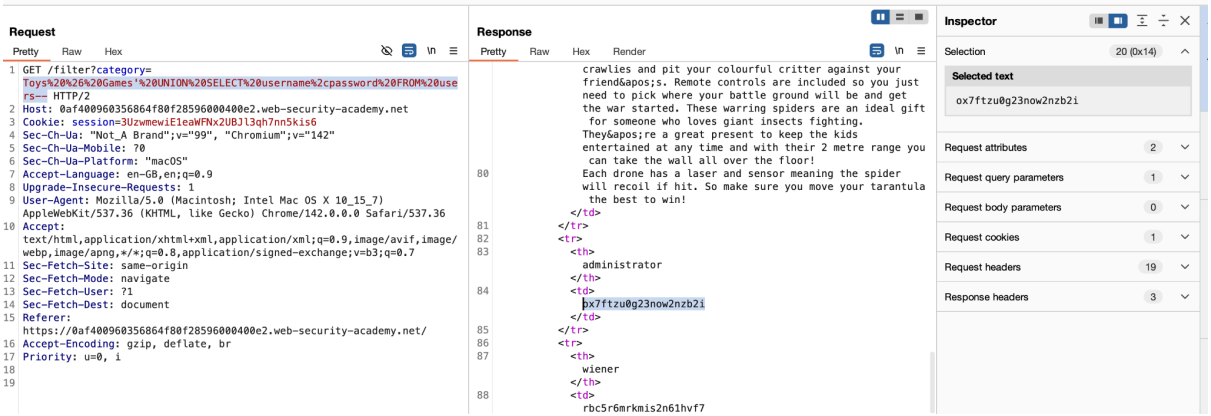
My Account

Your username is: administrator

Email

Update email

Identified two columns and the text-compatible column using `UNION SELECT` testing.
Injected `UNION SELECT username,password FROM users--` into the category parameter.
Retrieved admin credentials from the scrolling the html page and logged in successfully to complete the lab.



4)Tiny SQL CHECKER

The python code:

```
import requests
import json
import hashlib
import time
import argparse

# ---- Utility: fingerprint response ---- #
def fingerprint(resp):
    return {
        "status": resp.status_code,
        "length": len(resp.text),
        "hash": hashlib.md5(resp.text.encode()).hexdigest(),
        "elapsed_ms": int(resp.elapsed.total_seconds() * 1000)
    }

# ---- Send request with one payload ---- #
def send_request(url, param, payload):
    data = {param: payload}
    try:
        resp = requests.get(url, params=data, timeout=5)
        return fingerprint(resp)
    except Exception as e:
        return {"error": str(e)}

# ---- Main checker ---- #
def run_checker(url, param, delay):
    print(f"[+] Testing target: {url} on parameter: {param}")

    baseline_payload = "normaltest"
    true_payload = "' OR 1=1--"
    false_payload = "' AND 1=2--"

    print("[+] Sending baseline request...")
    baseline = send_request(url, param, baseline_payload)
    time.sleep(delay)

    print("[+] Sending TRUE-like payload...")
    true_test = send_request(url, param, true_payload)
    time.sleep(delay)

    print("[+] Sending FALSE-like payload...")
    false_test = send_request(url, param, false_payload)

    # ---- Comparison logic ---- #
    def is_suspicious(test, label):
```

```
        """Flag differences from baseline."""
        if "error" in test:
            return f"{label}: ERROR {test['error']}"
        if (test["status"] != baseline["status"] or
            test["length"] != baseline["length"] or
            test["hash"] != baseline["hash"]):
            return f"{label}: DIFFERENT from baseline → possible SQLi"
        return f"{label}: Looks normal"

    result_summary = {
        "baseline": baseline,
        "true_test": true_test,
        "false_test": false_test,
        "analysis": {
            "true_check": is_suspicious(true_test, "TRUE payload"),
            "false_check": is_suspicious(false_test, "FALSE payload")
        }
    }

    # Save JSON report
    with open("sqli_report.json", "w") as f:
        json.dump(result_summary, f, indent=4)

    print("\n[+] Report saved to sqli_report.json\n")
    return result_summary

# ---- CLI ---- #
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Tiny SQLi Checker")
    parser.add_argument("--url", required=True, help="Target URL like
http://test.com/page")
    parser.add_argument("--param", required=True, help="Parameter name to test")
    parser.add_argument("--delay", type=float, default=1.0, help="Delay between
requests")
    args = parser.parse_args()

    run_checker(args.url, args.param, args.delay)
```

Technologies : python, json,hashlib
Target : http://testphp.vulnweb.com/artists.php
Parameter tested : artist

Methods:

Set the Target

I first confirmed I was allowed to test the target, then provided the tool with the URL and the parameter I wanted to check. This ensures that the tool only tests what I explicitly allow.

Send a Normal Request (Baseline)

The tool sends a completely harmless, normal request to the server.
I recorded the response status, size, content hash, and time taken.

Send a TRUE-like SQLi Payload

Next, the tool sends a classic SQL injection test payload (`' OR 1=1--`).
If the website behaves differently here, it may be a sign of SQL injection.

Send a FALSE-like SQLi Payload

Then the tool sends another harmless payload (`' AND 1=2--`) which will always be FALSE.
If the website reacts differently to this one, it helps confirm suspicious behavior noticed in the TRUE payload.

Compare All Responses

The tool compares all three responses, the normal one, the TRUE-like one, and the FALSE-like one.
If the TRUE payload response is noticeably different from the baseline while the FALSE one is not, that often indicates SQL injection patterns.

Record All Results

Every fingerprint and comparison is saved into a structured JSON file, making it easy to review later or share as part of a report.

Manual Verification

The final step is simply checking any suspicious result manually (in a browser or with curl) to make sure it’s not just a random fluctuation or false alarm.

```
┌──(shaun㉿kali)-[~]
└─$ cat sqli_report.json

{
    "baseline": {
        "status": 200,
        "length": 2072,
        "hash": "44f72424b0e7819258ca598ad90038aa",
        "elapsed_ms": 510
    },
    "true_test": {
        "status": 200,
        "length": 2183,
        "hash": "0628c91eca77cc0be65cc22c11ecd3dc",
        "elapsed_ms": 523
    },
    "false_test": {
        "status": 200,
        "length": 2184,
        "hash": "e392f10f619009a748b92ee17804ebd7",
        "elapsed_ms": 525
    },
    "analysis": {
        "true_check": "TRUE payload: DIFFERENT from baseline \u2192 possible
SQLi",
        "false_check": "FALSE payload: DIFFERENT from baseline \u2192 possibl
e SQLi"
    }
}
```