

**Enhancing Authentication Security: A Python-Based System for Brute Force  
Attack Prevention**

**University of Essex**

**Master of Science in Cybersecurity**

**Beatrice Karimi Mutegi**

**Student ID: 12689392**

**Academic Year: 2022 - 2025**

**Supervisors: Dr. Oliver Buckley and Dr. Bakhtiyar Ahmed**

This Dissertation is presented in part fulfilment of the requirement for the completion of  
a Master of Science in Cybersecurity at the University of Essex.

## **Abstract**

Brute force attacks continue to pose significant threats to information systems, often targeting authentication mechanisms with inadequate security controls. This study explores and implements a Python-based authentication framework designed to mitigate such threats, particularly within Django-based systems.

The research investigates vulnerabilities commonly exploited in Python login systems, evaluates the strengths and weaknesses of existing brute force mitigation strategies, and proposes a secure, multi-layered defence system. The prototype integrates CAPTCHA, time-based OTP-based two-factor authentication, IP-based rate limiting, device fingerprinting, time-based email-token verification, logging, password management, and account lockout mechanisms.

Quantitative simulation results demonstrate that the system successfully blocked over 98% of brute-force attempts and provided real-time analytics to administrators via a dashboard. Additionally, user experience considerations were incorporated to balance security and usability.

This dissertation contributes to a practical model for securing Python/Django-based login systems, answering key research questions about effective mitigation while identifying areas for future enhancement in usability and system adaptability.

## **Declaration**

Submitted in fulfilment of the requirements for the Master of Science in Cybersecurity,

at the University of Essex

2025

The author hereby declares that this whole thesis or dissertation, unless specifically indicated to the contrary in the text, is her own original work.

---

**Beatrice Karimi Mutegi**

**MSc in Cybersecurity**

## Table of Contents

<b>Abstract.....</b>	<b>2</b>
<b>Declaration.....</b>	<b>3</b>
<b>List of Figures.....</b>	<b>11</b>
<b>List of Tables .....</b>	<b>14</b>
<b>Chapter 1: Introduction.....</b>	<b>16</b>
<b>1.1: Background.....</b>	<b>16</b>
<b>1.2: Problem Statement .....</b>	<b>17</b>
<b>1.3: Research Objectives.....</b>	<b>18</b>
<b>1.4: Research Questions .....</b>	<b>18</b>
<b>1.5: Significance of the Study .....</b>	<b>19</b>
<b>1.6: Dissertation Structure .....</b>	<b>20</b>
<b>Chapter 2: Literature Review.....</b>	<b>20</b>
<b>2.1: Overview of Authentication Security .....</b>	<b>20</b>
2.1.1: Traditional Authentication Methods.....	21
2.1.2: Modern Authentication Methods .....	21
2.1.3: Authentication Methods Comparison .....	24
<b>2.2: Brute Force Attacks and their Techniques .....</b>	<b>25</b>
<b>2.3: Existing Mitigation Strategies.....</b>	<b>26</b>
2.3.1: Account Lockout Policies: .....	27

2.3.2: CAPTCHA and Bot Detection: .....	27
2.3.3: Rate Limiting and IP Blocking: .....	28
2.3.4: Multi-Factor Authentication (MFA):.....	28
2.3.5: Adaptive Authentication: .....	28
2.3.6: Machine learning algorithms: .....	28
2.3.8: Encryption and Password Salting: .....	29
2.3.9: Ensuring Effective Password Management: .....	29
2.3.10: Intrusion Detection Systems (IDS): .....	30
2.3.11: Artificial Intelligence (AI):.....	30
<b>2.4: Limitations of Existing Solutions .....</b>	<b>30</b>
<b>2.5: Challenges in Securing Authentication Systems .....</b>	<b>31</b>
2.5.1: User Resistance and Usability Concerns:.....	31
2.5.2: Scalability:.....	31
2.5.3: Evolving Attack Methods:.....	31
<b>2.6: Research Gap.....</b>	<b>32</b>
<b>Chapter 3: Methodology .....</b>	<b>35</b>
<b>3.1: Research Design.....</b>	<b>35</b>
3.1.1: Methodological Framework:.....	36
3.1.2: Justification for Python and Django: .....	37
<b>3.2: Threat Modelling Approach.....</b>	<b>38</b>
3.2.1: STRIDE Threat Modelling: .....	39

3.2.2: OWASP Top 10 Reference for Secure Implementation:.....	41
<b>3.3: System Development Approach: Agile Methodology (Adapted for Solo Development) .....</b>	<b>42</b>
3.3.1: Sprint Planning: .....	43
3.3.2: Backlog Management and Prioritization: .....	49
3.3.3: Self-Evaluation and Reflection:.....	50
<b>3.4: Data Collection.....</b>	<b>50</b>
3.4.1: Literature Review (Qualitative):.....	50
3.4.2: Test Simulations (Quantitative): .....	51
3.4.3: Usability and System Feedback Evaluation using Heuristic Evaluation (Qualitative): .....	51
3.4.4: Logging and Dashboard Analytics: .....	51
<b>3.5: Data Analysis Approach .....</b>	<b>52</b>
<b>3.6: Ethical and Professional Considerations .....</b>	<b>52</b>
<b>Chapter 4: System Design and Implementation .....</b>	<b>53</b>
<b>4.1: System Requirements and Specifications.....</b>	<b>54</b>
4.1.1: System Architecture and Design:.....	54
4.1.2: Tools and Libraries Used: .....	56
<b>4.2: Implementation Strategy .....</b>	<b>59</b>
<b>4.3: Security Risk Mitigation and Compliance Mapping .....</b>	<b>64</b>
4.3.1: Secure Authentication Workflow: .....	67

<b>4.4: Testing Procedures</b> .....	68
4.4.1: Controlled Simulations:.....	69
4.4.2: Functional System Tests:.....	70
<b>4.4.3: Usability Evaluation using Nielsen's 10 Usability Heuristics:</b> .....	71
4.4.4: Dashboard Analytics:.....	72
4.4.5: Penetration Testing Approach and Tool Justification:.....	73
4.4.6: Ethical Considerations:.....	73
<b>4.5: Security vs. Usability Trade-offs</b> .....	74
<b>Chapter 5: Discussion and Evaluation of Results</b> .....	75
<b>5.1: Data Presentation and Analysis</b> .....	75
5.1.1: Brute-force Protection:.....	75
5.1.2: CAPTCHA Validation: .....	76
5.1.3: OTP Authentication:.....	76
5.1.4: Email Verification Authentication:.....	77
5.1.5: Password Expiry:.....	77
5.1.6: IP Lockout and Geolocation:.....	78
5.1.7: Admin and User Feedback:.....	78
<b>5.2: Evaluation Benchmarks and Metrics</b> .....	79
5.2.1: Usability Evaluations:.....	79
5.2.2: Security Metrics: .....	80
5.2.3: System Performance: .....	80

<b>5.3: Summary and Interpretation of the Results .....</b>	<b>80</b>
<b>5.4: Effectiveness of Addressing Research Gaps.....</b>	<b>81</b>
<b>5.5: Comparison with Existing Solutions.....</b>	<b>82</b>
<b>5.6: Challenges, Limitations and Proposed Solutions.....</b>	<b>84</b>
<b>Chapter 6: Conclusion and Recommendations.....</b>	<b>87</b>
<b>6.1: Summary of Key Findings.....</b>	<b>87</b>
<b>6.2: Alignment with Research Questions.....</b>	<b>88</b>
<b>6.3: Achievement of Research Objectives .....</b>	<b>91</b>
<b>6.4: Key Contributions to the Field.....</b>	<b>93</b>
<b>6.5: Recommendations.....</b>	<b>94</b>
<b>6.6: Future Work.....</b>	<b>96</b>
<b>6.7: Conclusion .....</b>	<b>96</b>
<b>References.....</b>	<b>98</b>
<b>Appendices.....</b>	<b>108</b>
<b>Appendix A: Setup Guide/Readme file.....</b>	<b>108</b>
<b>Appendix B: Source Code Snippets of Core Components .....</b>	<b>109</b>
Implementation of Django-Axes on Settings.py .....	109
Login view capturing rate limit .....	110
Signup view .....	111
Email Token Verification as on tokens.py.....	112
User Access Control on Decorators.py .....	112

OTP-based Two-Factor Authentication (2FA) Logic on Login view .....	113
CAPTCHA Verification Logic on Login view .....	113
Progressive Account Lockout Implementation Code as on utils.py .....	114
Email Token Generator Code.....	114
Urls.py .....	115
Device Fingerprinting (User agents) implementation on Signals.py.....	115
Google reCAPTCHA v2 settings .....	116
<b>Appendix C: Functionality Tests.....</b>	<b>117</b>
Login Logic Functionality Tests .....	117
Sign-up Logic Functionality Tests .....	123
Password Reset Logic Functionality Tests.....	127
Lockout Logic Functionality Tests .....	129
Contact Support Logic Functionality Tests.....	132
Session Expiry Functionality Test .....	133
<b>Appendix D: Simulation Tests and Results .....</b>	<b>135</b>
Progressive Lockout Simulation Tests and Results .....	135
Password Expiration and Change Simulation Tests and Results.....	137
Time-Based OTP Expiration (Login) Simulation Tests and Results .....	139
Time-Based Email Token Expiration (Signup) Simulation Tests and Results .....	140
Brute Force Attack Simulation Tests and Results .....	141
Distributed Brute Force Attack Simulation Tests and Results .....	144

Concurrent Session Test and Results.....	147
pytest and Results .....	148
<b>Appendix E: Dashboard Analytics and Lockout stats Logs Test.....</b>	<b>149</b>
Dashboard Analytics View .....	150

## List of Figures

Figure 1:Traditional vs. Modern Authentication Methods.....	25
Figure 4.2: System Architecture and Component Interaction .....	56
Figure 3: Secure Login and Signup Workflow with Integrated Security Controls.....	68
Figure 4: Django-Axes as on settings.py.....	109
Figure 5: Rate limit on login view.....	110
Figure 6: Signup view.....	111
Figure 7: Email token on tokens.py .....	112
Figure 8: User Access Control on decorators.py .....	112
Figure 9: OTP verification on Login view .....	113
Figure 10: CAPTCHA Verification on Login view .....	113
Figure 11: Progressive Lockout on utils.py .....	114
Figure 12: Email Token Generator on tokens.py .....	114
Figure 13: urls.py.....	115
Figure 14: User agents on signals.py .....	115
Figure 15: Google reCAPTCHA settings .....	116
Figure 16: Login page .....	117

Figure 17: Login with Invalid Credentials.....	118
Figure 18: Invalid CAPTCHA.....	118
Figure 19: Unverified OTP.....	119
Figure 20: OTP resent successfully.....	120
Figure 21: Email containing OTP.....	121
Figure 22: Login Successfully .....	122
Figure 23: Successfully Logged Out.....	122
Figure 24: Signup page.....	123
Figure 25: Password guide modal on signup page.....	124
Figure 26: Signup Error Handling.....	124
Figure 27: Email Verification.....	125
Figure 28: Email Verified Page .....	126
Figure 29: Password reset error handling 1 .....	127
Figure 30: Password reset error handling 2 .....	127
Figure 31: Password reset email .....	128
Figure 32: Password guide when resetting .....	128
Figure 33: Password reset successful.....	129

Figure 34: Account lockout page .....	130
Figure 35: Lockout email to admin .....	130
Figure 36: Account locked login page.....	131
Figure 37: User notified of locked account .....	131
Figure 38: Contact Support page .....	132
Figure 39: Customer Support Email Success.....	133
Figure 40: Session Expiry Test .....	134
Figure 41:simulate_progressive_lockout code .....	135
Figure 42: simulate_progressive_lockout result1 .....	136
Figure 43:simulate_progressive_lockout result2 .....	136
Figure 44: simulate_password_expiration Code .....	138
Figure 45: test_otp_expiration code and results.....	139
Figure 46: test_token_expiration code and results.....	140
Figure 47: simulate_bruteforce code .....	142
Figure 48:simulate_bruteforce result1 .....	143
Figure 49: simulate_bruteforce result2 .....	143
Figure 50: simulate_distributed_bruteforce code .....	145

Figure 51: simulate_distributed_bruteforce result1 .....	145
Figure 52: simulate_distributed_bruteforce result2 .....	146
Figure 53: simulate_distributed_bruteforce result3 .....	146
Figure 54: Concurrent session test and results .....	148
Figure 55: pytest results .....	149
Figure 56: Security Dashboard1 .....	150
Figure 57: Security Dashboard3.....	150

## List of Tables

Table 1: STRIDE Threat Modelling .....	41
Table 2: OWASP Threat Modelling .....	42
Table 3: Agile Methodology: Sprint Planning .....	49
Table 4: Tools and Libraries Used for System Development .....	59
Table 5: Security Features vs OWASP and STRIDE Compliance .....	66
Table 6: Usability Evaluations.....	79
Table 7: System Performance .....	80

Table 8: Addressed Research Gaps .....	82
Table 9: Challenges, Limitations and Proposed Solutions .....	86
Table 10: Research Questions Alignment.....	91
Table 11: Achievement of Research Objectives.....	93

## **Chapter 1: Introduction**

In order to combat the constant threat of brute force attacks, a recurring problem in cybersecurity, this dissertation investigates the development of a strong authentication system. The study focuses on designing and implementing a secure, Python-based framework within Django, addressing key vulnerabilities in login mechanisms commonly targeted by automated credential-guessing attempts. The implemented solution incorporates layered defences such as CAPTCHA, optional one-time password (OTP) verification, IP-based rate limiting, lockout policies, and real-time monitoring through an admin dashboard. This project aims to meet the requirements of the MSc Cyber Security program by providing a comprehensive analysis and a practical solution to enhance authentication security. It demonstrates mastery of secure system design, authentication, and attack prevention, aligning with MSc learning outcomes through applied research, technical implementation, and systematic evaluation.

### **1.1: Background**

Brute force attacks remain one of the most prevalent cybersecurity threats, targeting authentication systems across various industries (Abdulkader , et al., 2015). These attacks involve systematically attempting multiple credential combinations to gain unauthorized access, often exploiting weak or poorly protected login endpoints (Deep, et al., 2019). When authentication systems lack adequate safeguards, such attacks can lead to serious consequences, including data breaches and reputational damage (Uma & Padmavathi, 2013), (Cremer , et al., 2022).

Python, a language widely adopted in both academic and enterprise settings, facilitates rapid web application development through frameworks like Django and Flask (Nurhaida & Bisht , 2022). However, the default configurations in these frameworks may not provide sufficient protection against brute-force attacks, making additional security measures and customisation essential (Grunwaldt, 2019).

## **1.2: Problem Statement**

While several countermeasures, including CAPTCHA, account lockout policies, and multi-factor authentication (MFA), exist to mitigate brute force attacks, many systems still suffer from usability challenges, false positives, and bypass vulnerabilities (Zhang, et al., 2022). Despite the availability of various tools and strategies, Python-based login systems in particular often lack comprehensive, user-friendly, and adaptive security measures. This leaves applications vulnerable, especially in high-risk domains like online banking, e-commerce, and educational platforms (Jimmy, 2024). This research seeks to develop a Python-based authentication system within Django that incorporates enhanced security features to address these shortcomings. The goal is to improve upon existing methods, which, as (Nithya & Rekha, 2023) discuss, require continuous research and development to stay ahead of evolving threats.

### 1.3: Research Objectives

The aims and objectives of this research are to:

1. Evaluate the effectiveness of existing brute force attack prevention techniques in Python-based authentication systems.
2. Identify shortcomings in current solutions and explore potential enhancements.
3. Develop a Python-based authentication security solution within Django that is tailored for small organizations.
4. Evaluate the usability and effectiveness of the proposed solution, ensuring a balance between strong security and user convenience.
5. Recommend cost-effective security measures for organizations with limited resources.

Ultimately, this research aims to enhance authentication security against brute force attacks while maintaining secure usability, particularly for resource-limited organizations.

### 1.4: Research Questions

The primary research question guiding this study is: ***What are the most effective methods to prevent or mitigate brute force attacks in Python-based login systems?*** Below are additional research questions:

1. What vulnerabilities in Python-based login systems make them prone to brute force attacks?

2. What are the advantages and limitations of current brute force prevention mechanisms implemented in Python-based systems?
3. How can Python libraries and built-in functionalities be leveraged to strengthen the security of login systems against brute force attacks?
4. How can security measures be integrated into Python-based login systems to mitigate brute force attacks without compromising user experience?

### **1.5: Significance of the Study**

This study specifically falls under the Authentication and Authorization knowledge area within the Secure System Design and Architecture category of CyBOK (CyBOK, 2021). It aims to assess, enhance, and develop more effective authentication methods to mitigate brute force attacks. By incorporating better preventive techniques into Python-based authentication systems, unauthorized access can be minimized while maintaining usability for legitimate users (Gollmann, 2021).

This research contributes to cybersecurity by proposing an advanced authentication model that strengthens resilience against brute force attacks. Its conclusions aim to help organizations implement more secure python-based authentication frameworks. Additionally, by addressing the vulnerabilities in current systems, this research aligns with the broader goal of creating more secure and reliable authentication processes (Burrows, et al., 1989), (Shrivastava, et al., 2024). (Abdulkader , et al., 2015) points out that understanding the vulnerable points and potential attacks is crucial in developing effective authentication systems.

## **1.6: Dissertation Structure**

Chapter 2 reviews existing literature related to brute force attacks and Python-based security practices. Chapter 3 details the methodology and system design, including threat modelling, and ethical and professional considerations. Chapter 4 presents the implementation strategy and all the simulations and security tests performed. Chapter 5 provides a critical discussion of the findings/results, evaluating the system's performance against established benchmarks. Chapter 6 concludes with key findings, contributions, limitations, and recommendations for future research.

## **Chapter 2: Literature Review**

This chapter provides a review of existing literature on authentication security mechanisms, focusing on their effectiveness against brute force attacks. It examines traditional and modern authentication methods, countermeasures, and their limitations, highlighting the need for enhanced security solutions.

### **2.1: Overview of Authentication Security**

Authentication security encompasses various methods to verify user identities before granting system access (Farik, et al., 2016). These methods are categorized into traditional and modern authentication techniques, each with its own vulnerabilities, as discussed below:

### 2.1.1: Traditional Authentication Methods

- **Password-Based or Single-Factor Authentication (SFA):** The most common in web authentication method (Wang & Sun, 2020), but its inherent vulnerabilities make it a prime target for brute force attacks (Abdulkader , et al., 2015). Users often choose weak or easily guessable passwords, and password databases are susceptible to breaches, exposing credentials to attackers (Papathanasaki, et al., 2022). Techniques like password salting and hashing are employed to mitigate these risks, but are not foolproof.
- **CAPTCHA:** A common countermeasure to differentiate between human users and automated bots (Papathanasaki, et al., 2022). CAPTCHAs can deter automated brute force attacks, but often introduce usability challenges and may be bypassed by sophisticated bots or CAPTCHA-solving services (Vugdelija, et al., N.D.).

### 2.1.2: Modern Authentication Methods

- **Two-Factor Authentication (2FA):** To improve security, 2FA has been widely adopted. This method requires users to provide two separate pieces of evidence to authenticate their identity; something they know (a password) and something they have (One Time Pin/Password (OTP), hardware security token, or biometric feature) (Velásquez, et al., 2019). Though it strengthens protection, challenges remain such as phishing and SIM swapping attacks (Farik, et al., 2016).

- **Multi-Factor Authentication (MFA):** This further strengthens security by requiring evidence from at least two different categories: something you know (password), something you have (security token), and something you are (biometric feature) (Papathanasaki, et al., 2022). Despite its strengths, MFA faces challenges such as:

- Implementation complexities,
- Usability concerns,
- User resistance,
- The potential for bypass (Phan, 2008), (Farik, et al., 2016), (Mohammed & Dziyauddin, 2023).

Although more secure than Single-Factor Authentication (SFA), it still has vulnerabilities due to design flaws, and not all implementations guarantee enhanced security (Wee, et al., 2024). Moreover, only a small percentage of MFA schemes use three or more factors, thus limiting their overall effectiveness (Wang, et al., 2023).

- **Biometric Authentication:** Such as: fingerprint scanning, facial recognition, and voice recognition; offer a more secure alternative to passwords (Newman, 2009). These methods are more resistant to brute force attacks, but they are vulnerable to other types of attacks, such as spoofing and replay attacks (De Abiega-L'Eglise, et al., 2022).

- **Passwordless Authentication:** Aims to eliminate passwords entirely by leveraging methods like biometric data (fingerprint or facial recognition), hardware tokens, or cryptographic keys (e.g., WebAuthn) (Parmar, et al., 2022). While they offer higher security and ease of use, adoption is still limited due to technical and user experience challenges (Yusop, et al., 2025).
- **Mobile Application Authentication:** Toward secure mobile applications through proper authentication mechanisms, it's important to analyse collected data accurately (Albeshar, et al., 2024).
- **Graphical Passwords:** Using images, patterns, or gestures that theoretically offers improved security over traditional alphanumeric passwords (Golofit, 2007). They leverage human visual memory, making authentication both secure and user-friendly, especially against cyber threats like brute-force attacks and phishing (Raza, et al., 2012). However, their effective security can be compromised by predictable user behaviour, potentially making them vulnerable to informed guessing attacks (Golofit, 2007).
- **Blockchain-based Authentication or Decentralized Identity Management:** Using Ethereum network, MetaMask application and others, this method uses cryptographic techniques such as: public-private key pairs, digital signatures, and multi-factor authentication (Park, et al., 2023). Strong cryptographic algorithms make private keys highly resistant to brute-force attacks. However, poor key management such as: insecure storage or weak passphrases; can introduce

vulnerabilities (Rivera, et al., 2024). Additionally, while blockchain itself is secure, associated authentication mechanisms, such as: wallet passwords or recovery phrases; can still be targeted by brute-force attacks if not properly protected (Grimes, 2020).

### 2.1.3: Authentication Methods Comparison

Method	Type	Strengths	Weaknesses	Resistance to Brute Force Attacks
<b>Password-Based Authentication</b>	Traditional	Simple to implement	Vulnerable to guessing, phishing	Low (easily targeted)
<b>CAPTCHAs</b>	Traditional	Blocks automated bots	Usability issues, bypassed by AI	Moderate
<b>MFA and 2FA</b>	Modern	Adds layered security	Implementation complexity	High
<b>Biometric Authentication</b>	Modern	Unique biological traits	Spoofing, replay attacks	High
<b>Blockchain-Based Authentication</b>	Modern	Cryptographic security	Weak key management risks	Very High
<b>Adaptive</b>	Modern	Dynamic risk	Dependency on	High

Authentication		assessment	behavioural data	
----------------	--	------------	------------------	--

Figure 1: Traditional vs. Modern Authentication Methods

## 2.2: Brute Force Attacks and their Techniques

Brute force attacks involve systematically attempting numerous combinations of usernames and passwords to gain unauthorized access to a system or account (Cleary, 2024). Here's a breakdown of common techniques:

- **Simple Brute Force:** Involves trying every possible combination of characters until the correct password is found. The length and complexity of the password determine the time it takes to crack it (Contrast Security, 2021), for instance, a password consisting of only lowercase letters will be far easier to crack than one that includes uppercase letters, numbers, and symbols.
- **Dictionary Attacks:** This method uses a pre-defined list of common words and phrases, often obtained from dictionaries, books, or online databases, to guess passwords (Raza, et al., 2012). Attackers may also modify dictionary words by adding numbers, symbols, or capitalization to increase their chances of success.
- **Credential Stuffing:** Attackers use compromised username/password pairs obtained from data breaches on other services to try and gain access to accounts on different platforms (Ba, et al., 2021). This technique is effective because many users reuse the same credentials across multiple websites and applications.

- **Hybrid Brute Force:** Attackers combine dictionary words with numbers, symbols, and capitalization to create a wider range of password guesses (Cleary, 2024).
- **Reverse Brute Force:** Attackers have a list of known passwords and attempt them against multiple usernames (Cleary, 2024). This can be effective if the attacker has obtained a list of commonly used passwords from a data breach or other source (Hamza & Al-Janabi, 2024).
- **Parallel Brute Force:** Attackers use parallel techniques by dividing the search space among available resources, thus dividing the average time to success by the number of resources available (CAPEC, 2018).
- **Obfuscation Bypass:** Data obfuscation can make brute force attacks more difficult, but it does not eliminate the risk entirely. Attackers may use various techniques to bypass obfuscation methods and recover the original data (Contrast Security, 2021).

### 2.3: Existing Mitigation Strategies

Brute force cyberattacks are often motivated by financial gain, espionage, data theft, malware distribution, unauthorized access, identity theft, and the pursuit of power (Cleary, 2024). To counter these threats, various brute force attack prevention

mechanisms have been developed, each with its own strengths and limitations. These include:

**2.3.1: Account Lockout Policies:** Involves temporarily disabling user accounts or/and IP addresses after a certain number of failed login attempts (Herley & Florencio, 2008). While effective in preventing brute force attacks, these policies can lead to denial-of-service vulnerabilities and user frustration (Wang, et al., 2021). Additionally, they can be circumvented through distributed attack methods or by targeting systems with low lockout thresholds.

**2.3.2: CAPTCHA and Bot Detection:** CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) and bot detection mechanisms are widely used to differentiate between human users and automated bots (Vugdelija, et al., N.D.). CAPTCHAs can vary from:

- Simple math: Basic math problem (e.g., "What is 5 + 3?") to prove they are human,
- Invisible: Requires no user interaction unless suspicious activity is detected,
- Fun Captcha/Arkose Labs: Uses visual puzzles and tasks to identify humans from bots,
- And many more.

While these methods can deter brute-force attacks, they may inconvenience users. Moreover, text-based CAPTCHAs are increasingly vulnerable to machine learning-based bypasses (Moradi & Keyvanpour, 2015). Although CAPTCHAs can be over 90%

effective for humans and under 1% for bots (Tariq, et al., 2023), they are not sufficient on their own. For stronger protection, it should be combined with complementary measures such as IP rate limiting and OTP; for a more secure, layered defence.

**2.3.3: Rate Limiting and IP Blocking:** Rate limiting restricts the number of login attempts allowed within a specific time frame (Tamilkodi, et al., 2024). However, it can be bypassed through distributed attacks originating from multiple IP addresses, proxies or VPNs (Anon, N.D.). Additionally, it may lead to false positives if users share IPs or use dynamic IPs. While IP blocking involves blocking traffic from specific IP addresses that are associated with malicious activity (Nurhaida & Bisht , 2022).

**2.3.4: Multi-Factor Authentication (MFA):** Enhancing authentication security by requiring additional verification methods (Zhang, et al., 2018).

**2.3.5: Adaptive Authentication:** Using machine learning algorithms to analyse user behaviour and detect anomalous login attempts (Najafabadi, et al., 2015). Thus adjusting security measures based on the risk level of each login attempt.

**2.3.6: Machine learning algorithms:** For detecting brute force attacks at the network level, using features extracted from network flow data (Najafabadi, et al., 2015). These models evaluate the risk of authentication attempts and can

trigger additional security steps, such as multi-factor authentication, for high-risk logins (Hamza & Al-Janabi, 2024).

**2.3.7: Device Fingerprinting and Behavioural Analysis:** Techniques such as device fingerprinting, which tracks the devices used for logins, and behavioural analysis, which monitors user login patterns, can be used to identify suspicious login attempts and prevent brute-force attacks (Nikiforakis, et al., 2013).

**2.3.8: Encryption and Password Salting:** Enhances password security by adding a unique, random string (salt) to each password before hashing. This prevents attackers from using precomputed hash tables (rainbow tables) to crack multiple passwords at once (Vugdelija, et al., N.D.).

**2.3.9: Ensuring Effective Password Management:** Best practices include: using complex and unique passwords, avoiding easily guessable personal information, and refraining from reusing passwords across multiple accounts (Information Commissioner's Office, 2024), (Das, et al., 2014). Administrators and developers play a critical role in enforcing these measures by:

- deactivating unused accounts,
- implementing strict password policies,
- mandating periodic password updates (e.g., every 90 days),
- establishing complexity requirements to strengthen overall security,
- etc. (Owens & Matthews, N.D.).

**2.3.10: Intrusion Detection Systems (IDS):** These solutions (Network-based and Host-based) analyse network traffic and system logs to detect repeated failed login attempts, unusual access patterns, and high-volume authentication requests that may indicate a brute force attack (Idhom, et al., 2020).

**2.3.11: Artificial Intelligence (AI):** Just as Machine Learning, both of them detect anomalies in login behaviour (Velgekar, et al., 2021). By analysing user patterns, these technologies can identify unusual login attempts that may indicate brute-force or credential-stuffing attacks.

## **2.4: Limitations of Existing Solutions**

Although existing security measures offer some level of protection, they often present usability concerns, high false positive rates, and vulnerability to social engineering attacks (Vugdelija, et al., N.D.).

Additionally, solutions like Intrusion Detection System (IDS) alone are insufficient to stop brute-force attacks, highlighting the need for a multi-layered approach. (Vugdelija, et al., N.D.). A robust and continuously evolving authentication framework is essential (Weingart , 2002).

## **2.5: Challenges in Securing Authentication Systems**

While significant strides have been made in improving authentication security, challenges remain. Some of the most pressing issues include:

**2.5.1: User Resistance and Usability Concerns:** Advanced security measures such as MFA and CAPTCHA can impact the user experience. Users often resist changes that complicate the login process, especially if the new mechanisms introduce friction in their day-to-day interactions with systems (Olayinka , et al., 2024).

**2.5.2: Scalability:** As organizations scale, maintaining secure and effective authentication systems becomes increasingly complex (IndiaFreeNotes, 2023). Solutions like rate limiting and CAPTCHA must be carefully balanced to ensure they do not disrupt legitimate users while effectively preventing attacks (Moradi & Keyvanpour, 2015).

**2.5.3: Evolving Attack Methods:** Cybercriminals continuously develop new tactics to bypass security measures. For instance, attackers may use botnets or distributed brute-force attacks to overcome IP-based rate limiting, making it essential to continuously update defence mechanisms (Aslan, et al., 2023).

## 2.6: Research Gap

Despite the widespread use of traditional and modern authentication methods, such as: password-based authentication, CAPTCHA, account lockout policies, and multi-factor authentication; significant challenges remain in preventing brute force attacks, especially for small organizations with limited resources.

### **Key gaps in current solutions include:**

- **Effectiveness of Multi-Layered Defence Mechanisms:** While individual defence strategies like CAPTCHA, rate limiting, and account lockout have been well-studied, there is limited research on how to effectively combine these mechanisms into a robust, multi-layered defence strategy (Lu, et al., 2018), (OWASP, N.D.).

This is critical for mitigating brute-force attacks, as attackers often exploit gaps in isolated measures. More research is needed to develop integrated, adaptive authentication systems that can respond to evolving threats.

- **Usability vs. Security Trade-off:** Many existing countermeasures, such as strict account lockout or Multi-Factor Authentication (MFA) or frequent CAPTCHA prompts; can frustrate legitimate users and disrupt workflow, particularly in environments where user experience is critical (Olayinka , et al., 2024).

Striking the right balance between robust security and a seamless user experience remains a challenge for many organizations (Downey & Laskowski,, 1996) .

- **False Positives and Administrative Overhead:** Static rate limiting and lockout policies can result in false positives, inadvertently blocking legitimate users. This leads to increased support requests, administrative overhead, and user dissatisfaction (Nurhaida & Bisht , 2022). Addressing this without sacrificing security is a significant gap in current solutions.
- **Resource Constraints:** While advanced solutions such as: adaptive authentication or machine learning-based or Artificial Intelligence (AI) approaches; show promise in improving security, they are often too complex or resource-intensive for smaller organizations to implement and maintain effectively (Aslan, et al., 2023). This presents a barrier to adoption, leaving these organizations vulnerable to brute-force attacks (Sarveshwaran, et al., 2023).
- **Bypass Vulnerabilities:** Despite improvements in security measures, attackers continue to discover ways to bypass traditional defence mechanisms (Grimes, 2020). For example, distributed IP attacks can bypass rate limiting, and CAPTCHA-solving bots are readily available, undermining the effectiveness of these defences (Certus Cybersecurity, 2023).
- **Complexity of Blockchain for Decentralized Authentication:** Blockchain offers potential for decentralized identity management and enhanced authentication security (Deep, et al., 2019).

However, its application in preventing brute-force attacks is still in the early stages. Further research is needed to explore how blockchain can deliver

tamper-proof, decentralized authentication without relying on centralized systems (Rivera, et al., 2024).

**This research addresses these gaps by:**

- **Developing a Python-based authentication system using Django** that incorporates multiple, practical security measures such as: static rate limiting, progressive account lockout, and multi-factor authentication; to provide a layered defence against brute-force attacks.
- **Focusing on solutions that are straightforward to implement and maintain**, making them accessible for organizations with limited technical or financial resources.
- **Evaluating the usability and effectiveness of these mechanisms** to ensure that security improvements do not come at the expense of legitimate user access and productivity.
- **Providing a cost-effective, scalable authentication framework** that can be adopted by small organizations, directly addressing the limitations found in current brute-force attack prevention strategies.

By targeting the balance between robust security and practical usability, this project seeks to provide a cost-effective and scalable authentication framework that can be readily adopted by small organizations, while directly addressing the limitations observed in current brute force attack prevention strategies.

## Chapter 3: Methodology

This chapter presents the methodology used in the design, development and evaluation of a secure Django-based login prototype system to prevent brute force attacks. It details the research approach, threat modelling, system development planning, ethical considerations, data collection methods, and they align with the research objectives and gaps discussed in Chapter 2.

### 3.1: Research Design

A **Design Science Research (DSR)** methodology was adopted as it is well-suited for solving real-world problems through the creation of functional IT artifacts. DSR emphasizes artifact creation, evaluation, and contribution to practice and knowledge (Hevner, et al., 2004). A **mixed-methods approach** was used where:

- **Quantitative methods:** Collecting and analysing system logs, lockout rates, OTP use, and response times during simulated attacks, with a focus on CAPTCHA and OTP validation, as well as dashboard analytics to assess the effectiveness of authentication mechanisms (Bhatia, 2018).
- **Qualitative methods:** Literature review, STRIDE/OWASP threat modelling, internal assessments and heuristic evaluation to identify security risks and usability improvements (Creswell, 2017) .

**Agile principles** supported iterative testing and refinement across development sprints, enabling both theoretical and practical insights into authentication security (Sutherland, 2014).

### 3.1.1: Methodological Framework:

This study followed the DSR methodology which includes the following key steps (Hevner, et al., 2004):

DSR Step	Activities	Tools/Techniques
<b>1. Problem Identification and Motivation</b>	Identified brute-force attack threats and usability/security issues in existing authentication methods.	<ul style="list-style-type: none"><li>• Literature review,</li><li>• STRIDE threat modelling</li><li>• OWASP threat modelling</li></ul>
<b>2. Define the Objectives of a Solution</b>	Outlined security goals: <ul style="list-style-type: none"><li>• Multi-layered authentication (CAPTCHA, 2FA, rate limiting),</li><li>• Usability and scalability.</li></ul>	<ul style="list-style-type: none"><li>• Research questions,</li><li>• system objectives,</li><li>• sprint planning</li></ul>
<b>3. Design and Development</b>	Developed Django-based authentication system with integrated brute-force mitigation techniques.	<ul style="list-style-type: none"><li>• Python,</li><li>• Django,</li><li>• SQLite3,</li><li>• Agile methodology</li></ul>
<b>4. Demonstration</b>	Conducted brute-force simulations using custom scripts and automated tools.	<ul style="list-style-type: none"><li>• Simulation tools,</li><li>• CLI-based attack scripts</li></ul>
<b>5. Evaluation</b>	Evaluated system through: <ul style="list-style-type: none"><li>• functional testing,</li></ul>	<ul style="list-style-type: none"><li>• Manual testing,</li><li>• brute-force</li></ul>

	<ul style="list-style-type: none"> <li>• usability heuristics,</li> <li>• threat models (STRIDE, OWASP).</li> </ul>	simulation tools, <ul style="list-style-type: none"> <li>• dashboard analytics</li> </ul>
<b>6. Communication</b>	<ul style="list-style-type: none"> <li>• Documented and presented the system and findings through this dissertation and future publications.</li> </ul>	<ul style="list-style-type: none"> <li>• Academic writing,</li> <li>• GitHub repository,</li> <li>• Presentation</li> <li>• E-portfolio</li> </ul>

### 3.1.2: Justification for Python and Django:

Python was selected for its simplicity, rapid development capabilities, and robust libraries that support secure web application development (Lutz, 2013). Between popular frameworks, Django was chosen over flask because it offers extensive built-in features and support for libraries for user authentication, database integration, and form validation (Django Software Foundation, 2023). This makes it ideal for implementing a structured and secure authentication system. In contrast, Flask requires more manual setup and third-party extensions, which can introduce inconsistencies or additional vulnerabilities (Devndra, 2020).

While Python may lack the performance of languages like C or Java, its readability and community support make it suitable for secure web application development. However, Django's default security features require customization and hardening to effectively defend against brute-force attack (Idris, et al., 2020).

### 3.2: Threat Modelling Approach

Although this research focuses on brute-force attack prevention, broader threat modelling frameworks such as OWASP and STRIDE were selectively integrated to maintain focus without diluting the core objective. Instead of applying these frameworks in full, relevant concepts such as rate limiting, authentication failure handling, and credential protection were incorporated.

A **hybrid approach** was adopted, whereby **STRIDE** was used for architectural threat identification within the login system, while **OWASP** served as a reference for secure implementation, particularly around login abuse and access control (OWASP, 2021), (Department for Science, Innovation & Technology, 2024)

Additionally, the study also recognizes the evolving nature of authentication threats. While traditional password-based methods are still common, they remain vulnerable to brute-force and dictionary attacks, especially when passwords are reused or weak (Rashidi & Garg, 2021).

As recommended by OWASP (CWE Content Team, 2021) and (NIST, 2025), the system incorporates layered defences including: CAPTCHA, OTP, and account lockouts; to reduce the risk of unauthorized access while aligning with modern security standards.

This targeted application of STRIDE and OWASP ensures the authentication system is both resilient and realistic for deployment in Django-based environments, without overcomplicating the scope of the research (Khan, et al., 2017).

### 3.2.1: STRIDE Threat Modelling:

STRIDE was applied to assess threats within the core areas of the login system (Department for Science, Innovation & Technology, 2024), (Khan, et al., 2017). The following table outlines identified threats and the corresponding mitigations:

STRIDE Threat	Mitigations Applied
<b>Spoofing (impersonating a user or service)</b>	Mitigated through: <ul style="list-style-type: none"><li>• strong passwords,</li><li>• email verification during signup,</li><li>• time-based OTP-based-two-factor authentication (2FA) during login,</li><li>• device fingerprinting using user-agent and IP address logging</li></ul>
<b>Tampering (modifying data or code)</b>	Addressed using: <ul style="list-style-type: none"><li>• input validation,</li><li>• secure hashing of passwords,</li><li>• CSRF tokens,</li><li>• Secure session cookies,</li><li>• HTTPS</li></ul>
<b>Repudiation (denying performing an action)</b>	<ul style="list-style-type: none"><li>• Authentication logging using Django Axes whereby Login attempts, both successful and failed, are logged with IP and timestamp data to ensure traceability,</li><li>• device fingerprinting (lockout logging with OS/device</li></ul>

	<p>info),</p> <ul style="list-style-type: none"> <li>• email alerts to admin on lockout events,</li> <li>• lockout logs are traceable on <i>security_dashboard</i> and <i>lockout_stats</i> that has access control</li> </ul>
<b>Information Disclosure</b> <b>(exposing confidential information)</b>	<ul style="list-style-type: none"> <li>• Sensitive error messages are suppressed,</li> <li>• data in transit is protected through HTTPS,</li> <li>• CAPTCHA (google reCAPTCHA v2),</li> <li>• session expiry,</li> <li>• TOTP expiration (10 min limit),</li> <li>• email OTP used to control access</li> </ul>
<b>Denial of Service</b> <b>(disrupting service availability)</b>	<ul style="list-style-type: none"> <li>• Progressive account lockout mechanism with escalating timeouts (5 to 60 mins),</li> <li>• django-ratelimit decorator (@ratelimit(key=..., rate='5/15m'))</li> <li>• lockout status tracked and enforced via cache,</li> <li>• reCAPTCHA to block automated abuse</li> </ul>
<b>Elevation of Privilege</b> <b>(gaining unauthorized access or privileges)</b>	<ul style="list-style-type: none"> <li>• Role-based access control on admin views,</li> <li>• optional 2FA for privileged users,</li> <li>• enforced password complexity,</li> <li>• forced password expiration,</li> <li>• decorators protect restricted pages,</li> </ul>

	<ul style="list-style-type: none"> <li>• secure user creation whereby user's account remains inactive until the user verifies a time-based email token and profile tracking</li> </ul>
--	--

*Table 1: STRIDE Threat Modelling*

While full STRIDE implementation was beyond the project's scope, key elements such as: Spoofing (via authentication hardening) and Denial of Service (via rate limiting); were selectively applied. This focused use of STRIDE principles ensures alignment with industry best practices while maintaining a clear emphasis on brute-force attack mitigation.

### 3.2.2: OWASP Top 10 Reference for Secure Implementation:

This research does not aim to address all OWASP Top 10 vulnerabilities comprehensively. Instead, relevant risks were selectively referenced during implementation to validate the system's security posture. This ensures adherence to baseline standards while keeping the focus on brute-force prevention (OWASP, 2021).

OWASP Risks	Applied Mechanism
<b>A01:2021 – Broken Access Control</b>	<ul style="list-style-type: none"> <li>• Restricted access to administrative views using Django's built-in permission system and role-based access logic</li> </ul>
<b>A02:2021 – Cryptographic Failures</b>	<ul style="list-style-type: none"> <li>• Secure password hashing using Django's PBKDF2,</li> <li>• use of randomly generated, time-bound tokens for email verification and OTP</li> </ul>

<b>A07:2021 – Identification and Authentication Failures</b>	<p>Implementation of:</p> <ul style="list-style-type: none"> <li>• account lockout thresholds,</li> <li>• Google reCAPTCHA,</li> <li>• time-based OTP-based 2FA to enforce layered authentication security,</li> <li>• username/IP based progressive lockouts,</li> <li>• time-based email verification during signup</li> </ul>
<b>A09:2021 – Security Logging and Monitoring Failures</b>	<ul style="list-style-type: none"> <li>• Login attempts logged in <i>LockoutLog</i>,</li> <li>• admin alerted via <i>send_mail()</i> on lockout,</li> <li>• logs include device, OS, user agent, and location (if available)</li> <li>• Error handling</li> </ul>
<b>A10:2021 – Server-Side Request Forgery (SSRF)</b>	<ul style="list-style-type: none"> <li>• External requests (e.g., IP location during lockout) are isolated,</li> <li>• error-handled and sanitized,</li> <li>• minimal reliance on external APIs,</li> <li>• hardened request handling in <i>lockout_stats</i></li> </ul>

Table 2: OWASP Threat Modelling

### 3.3: System Development Approach: Agile Methodology (Adapted for Solo Development)

A *lightweight Agile development methodology* was adopted to manage implementation efficiently, tailored for solo research without team collaboration or external feedback

(Purba & Ramli, 2022). Agile's iterative and flexible nature supported continuous development, integration of security features, testing, and refinement in manageable increments (Moyo & Mnkandla, 2019).

**3.3.1: Sprint Planning:** A solo-adapted Agile methodology guided development was adapted, each lasting approximately 3-5 weeks.

Sprint	Key Tasks	Objectives	Tools / Frameworks
<b><u>Sprint 1:</u></b> <ul style="list-style-type: none"> <li>• <b>System Initialization,</b></li> <li>• <b>Project Setup,</b></li> <li>• <b>User Management</b></li> </ul>	<ul style="list-style-type: none"> <li>• Set up Django project</li> <li>• Extend Django <i>UserProfile</i></li> <li>• Configure user models</li> <li>• Basic UI templates</li> </ul>	<ul style="list-style-type: none"> <li>• Create clear and minimalistic pages</li> <li>• Set-up the views and urls.py</li> <li>• Install and upgrade Django dependencies like pipenv, etc</li> </ul>	<ul style="list-style-type: none"> <li>• Django,</li> <li>• Allauth,</li> <li>• HTML/CSS</li> </ul>
<b><u>Sprint 2:</u></b> <ul style="list-style-type: none"> <li>• <b>Configuration of Login, Signup, Customer Support Forms and Pages</b></li> </ul>	<ul style="list-style-type: none"> <li>• Configure user models</li> <li>• Ensure Input is validated on forms</li> <li>• Error handling messages</li> <li>• Handle email</li> </ul>	<ul style="list-style-type: none"> <li>• Establish user registration and login foundation with verified accounts</li> <li>• Improve UX and resilience under</li> </ul>	<ul style="list-style-type: none"> <li>• Django Auth</li> <li>• HTML/CSS,</li> <li>• CSRF token</li> <li>• Forms.py,</li> <li>• <i>send_mail,</i></li> <li>• <i>fail_silently=True</i></li> </ul>

	failures silently <ul style="list-style-type: none"> <li>• Disable user enumeration</li> </ul>	failure	
<b><u>Sprint 3:</u></b> <ul style="list-style-type: none"> <li>• <b>Basic Authentication</b></li> <li>• <b>Configure Decorators for Access Control and Authorization</b></li> <li>• <b>Password Guidelines Modal</b></li> </ul>	<ul style="list-style-type: none"> <li>• Configure basic authentication</li> <li>• Configure password validators</li> <li>• Create and configure decorators to protect views</li> <li>• Create a modal that helps user know the guidelines of creating a stronger password as per the system settings.</li> </ul>	<ul style="list-style-type: none"> <li>• Establish functional base for authentication flow and secure account creation.</li> <li>• Establish and set the guidelines that can help the user create stronger passwords.</li> </ul>	<ul style="list-style-type: none"> <li>• Django</li> <li>• HTTP/CSS</li> <li>• Decorators.py</li> <li>• CSRF token</li> </ul>
<b><u>Sprint 4:</u></b> <ul style="list-style-type: none"> <li>• <b>Session Management,</b></li> <li>• <b>Forced Password Expiry,</b></li> <li>• <b>Password</b></li> </ul>	<ul style="list-style-type: none"> <li>• Enforce session expiry &amp; logout rules</li> <li>• Enforce concurrent sessions</li> <li>• Password rotation after 90 days</li> <li>• Password Expiry</li> </ul>	<ul style="list-style-type: none"> <li>• Strengthen session security, usability, and ensure feature completeness</li> <li>• Enforce access controls</li> </ul>	<ul style="list-style-type: none"> <li>• Django Settings,</li> <li>• Sessions,</li> <li>• CSRF,</li> <li>• Alerts</li> <li>• HTML/CSS</li> <li>• decorators</li> </ul>

<b>Reset Option</b>  <ul style="list-style-type: none"> <li>• <b>Final Hardening (e.g. Authorization and Access Control)</b></li> </ul>	form  <ul style="list-style-type: none"> <li>• Password reset option</li> <li>• Add error handling, user messaging</li> <li>• Final internal usability review</li> <li>• Protect views with decorators</li> </ul>	<ul style="list-style-type: none"> <li>• Giving user option of resetting the password in the event of the user forgetting the password.</li> </ul>	
<u><b>Sprint 5:</b></u>  <ul style="list-style-type: none"> <li>• <b>Brute Force Detection,</b></li> <li>• <b>Progressive Lockout System</b></li> <li>• <b>Rate Limiting</b></li> </ul>	<ul style="list-style-type: none"> <li>• Implement <i>django-ratelimit</i></li> <li>• Cache-based lockout logic</li> <li>• Design <i>LockoutLog</i> model</li> <li>• Create a lockout page for after failed attempts</li> <li>• Set progressive lockout thresholds (3/5/10)</li> </ul>	<ul style="list-style-type: none"> <li>• Prevent brute-force login attempts via IP/user-based throttling and caching</li> </ul>	<ul style="list-style-type: none"> <li>• <i>Django-ratelimit</i>,</li> <li>• Django-axes,</li> <li>• Cache,</li> <li>• Custom middleware,</li> <li>• HTML/CSS</li> </ul>
<u><b>Sprint 6:</b></u>  <ul style="list-style-type: none"> <li>• <b>Time-based</b></li> </ul>	<ul style="list-style-type: none"> <li>• Integrate django-two-factor-auth and</li> </ul>	<ul style="list-style-type: none"> <li>• Enhance authentication</li> </ul>	<ul style="list-style-type: none"> <li>• PyOTP, django-two-</li> </ul>

<b>OTP-Based Two-Factor Authentication (2FA)</b>	PyOTP <ul style="list-style-type: none"> <li>• Configure OTP expiration (for 10-minutes)</li> <li>• Handle resend OTP and expired token flows</li> <li>• Internal test cases for OTP logic</li> </ul>	robustness using optional email-based OTP with expiration enforcement	factor-auth, <ul style="list-style-type: none"> <li>• SMTP email</li> <li>• Timezone</li> </ul>
<u><b>Sprint 7:</b></u> <ul style="list-style-type: none"> <li>• <b>Time-based Email Token Generator and Verification during Sign up</b></li> </ul>	<ul style="list-style-type: none"> <li>• Setup time-based email tokens to be sent to user during signup</li> <li>• User is inactive until the email is verified</li> </ul>	<ul style="list-style-type: none"> <li>• User remain inactive until email verification is successful</li> </ul>	<ul style="list-style-type: none"> <li>• Django</li> <li>• SMTP email,</li> </ul>
<u><b>Sprint 8:</b></u> <ul style="list-style-type: none"> <li>• <b>CAPTCHA Integration &amp; Bot Defence</b></li> </ul>	<ul style="list-style-type: none"> <li>• Replace hCaptcha with Google reCAPTCHA v2</li> <li>• Server-side CAPTCHA validation</li> <li>• Trigger CAPTCHA dynamically after 3 failed logins</li> </ul>	<ul style="list-style-type: none"> <li>• Block automated and bot-based login abuse</li> </ul>	<ul style="list-style-type: none"> <li>• Google reCAPTCHA v2,</li> <li>• Requests,</li> <li>• JavaScript</li> </ul>

	<ul style="list-style-type: none"> <li>• Add CAPTCHA fail logging</li> </ul>		
<b><u>Sprint 9:</u></b> <ul style="list-style-type: none"> <li>• Logging,</li> <li>• Device Fingerprinting</li> <li>• Geolocation</li> <li>• Admin Alerts</li> </ul>	<ul style="list-style-type: none"> <li>• Collect device info: OS, browser, IP, location</li> <li>• Log events in LockoutLog with geolocation</li> <li>• Notify admin via email for suspicious activity</li> </ul>	<ul style="list-style-type: none"> <li>• Provide forensic and threat insights</li> </ul>	<ul style="list-style-type: none"> <li>• send_mail</li> <li>• user agent</li> <li>• parser,</li> <li>• IP/geolocation,</li> <li>• LockoutLog</li> </ul>
<b><u>Sprint 10:</u></b> <ul style="list-style-type: none"> <li>• Heuristic Evaluation &amp; UX Review</li> </ul>	<ul style="list-style-type: none"> <li>• Apply Nielsen's 10 usability heuristics,</li> <li>• Internal walkthroughs with test accounts,</li> <li>• Refine navigation and UI clarity</li> </ul>	<ul style="list-style-type: none"> <li>• Ensure usability aligns with security.</li> </ul>	<ul style="list-style-type: none"> <li>• Nielsen Heuristics, manual UX testing</li> </ul>
<b><u>Sprint 11:</u></b> <ul style="list-style-type: none"> <li>• Dashboard Visualization for Admin Security</li> </ul>	<ul style="list-style-type: none"> <li>• Create dashboard app</li> <li>• Develop security_dashboard page with Chart.js</li> </ul>	<ul style="list-style-type: none"> <li>• Visualize security threats and user behaviour</li> </ul>	<ul style="list-style-type: none"> <li>• Django,</li> <li>• Chart.js</li> <li>• GeoIP2,</li> <li>• Recharts,</li> <li>• RBAC,</li> </ul>

<p><b>Analytics</b></p> <ul style="list-style-type: none"> <li>• <b>Creation of lockout_stats page for Logging</b></li> </ul>	<ul style="list-style-type: none"> <li>• Build lockout_stats page</li> <li>• Implement heatmap of failed login geolocation</li> <li>• Visualize CAPTCHA/OTP success/fail rates trends</li> <li>• Role-restricted and Admin-only access to dashboard analytics and lockout_stats</li> </ul>		<ul style="list-style-type: none"> <li>• HTML/CSS</li> <li>• VScode,</li> </ul>
<p><b><u>Sprint 12:</u></b></p> <ul style="list-style-type: none"> <li>• <b>Simulation &amp; Testing</b></li> </ul>	<ul style="list-style-type: none"> <li>• Simulate brute-force attacks</li> <li>• Log responses to failed login attempts</li> <li>• Evaluate false positives, OTP bypass attempts</li> <li>• Confirm the security dashboard logs data</li> </ul>	<ul style="list-style-type: none"> <li>• Validate robustness under attack scenarios</li> </ul>	<ul style="list-style-type: none"> <li>• Internal simulation scripts like: Brute force attack simulation test</li> </ul>

<b><u>Sprint 13:</u></b>  <b>• Final Polish and Documentation</b>	<ul style="list-style-type: none"> <li>• Final testing and bug fixes</li> <li>• Code cleanup, inline comments, docstrings,</li> <li>• Document sprint retrospectives</li> </ul>	<ul style="list-style-type: none"> <li>• Ensure project quality and maintainability</li> </ul>	<ul style="list-style-type: none"> <li>• VS Code,</li> <li>• GitHub,</li> <li>• Markdown</li> </ul>
---	---	--	---

*Table 3: Agile Methodology: Sprint Planning*

**3.3.2: Backlog Management and Prioritization:** A backlog of tasks was maintained and updated regularly based on system performance and technical feasibility. Features were prioritized based on their:

- Alignment with research objectives (e.g., brute force protection)
- Technical feasibility
- Interdependencies (e.g., CAPTCHA after lockout mechanism)
- Security criticality (e.g., enforcing OTP before role-based access)

Tasks were re-prioritized after retrospectives if tests and simulations indicated weaknesses in security response or usability friction. For example, integrating CAPTCHA in Sprint 8 was time-consuming due to challenges with validation handling during testing. As a result, I temporarily moved on to later sprints and returned to Sprint 8 afterward to complete the integration.

**3.3.3: Self-Evaluation and Reflection:** At the conclusion of each sprint, a solo retrospective was performed focusing on:

- Progress vs. expected sprint goals
- Issues encountered (e.g., CAPTCHA token timing, OTP delays)
- Feedback from internal testing and simulations
- Code quality and maintainability review

Lessons learned informed the planning for the next sprint. For instance, after Sprint 5, the need for granular lockout logging (device fingerprinting, geolocation) became apparent, leading to its inclusion in the cycle.

### **3.4: Data Collection**

Since no human participants were involved, the study relied on internal simulations, automated system testing, simulated attack scenarios, heuristic evaluation, and system event logs (Norman & Kirakowski, 2018). These approaches provided comprehensive insights into the system's robustness, performance, and resistance to brute-force attacks.

**3.4.1: Literature Review (Qualitative):** A systematic literature review was conducted to analyse existing authentication security mechanisms, brute force attack methodologies, and countermeasures (Velásquez, et al., 2018). This helped establish a foundation for the proposed security enhancements.

**3.4.2: Test Simulations (Quantitative):** A series of controlled test simulations were designed and executed to evaluate the behaviour of the Django-based secure authentication system under various conditions (Palmieri, 2013). These tests focused on replicating real-world attack patterns and legitimate user behaviour.

**3.4.3: Usability and System Feedback Evaluation using Heuristic Evaluation (Qualitative):** With no external users involved, usability was assessed internally through heuristic evaluation based on Nielsen's 10 Usability Heuristics (Nielsen & Molich, 1990). The system's interface and interaction flows were systematically reviewed, focusing on error prevention, clarity, recovery from failures, and feedback mechanisms (Downey & Laskowski, 1996). Documented observations guided interface refinements to enhance overall usability while maintaining a high level of security (Lodhi, 2010).

**3.4.4: Logging and Dashboard Analytics:** Detailed logs of authentication events were recorded and further analysed using the custom-built admin dashboard. The dashboard visualized key security metrics such as:

- Lockout frequency by IP address and time
- Geolocation of failed login attempts
- CAPTCHA failure rates
- OTP validation success and failure trends

These visual insights helped validate the effectiveness of implemented security controls, identify abnormal activity patterns, and ensure that lockouts and challenges were functioning as intended.

### **3.5: Data Analysis Approach**

Each security feature implemented in the Django authentication system was analysed based on predefined success criteria and the outcome of controlled simulations. The analysis considered the system's resilience to attacks, usability under pressure, and the effectiveness of feedback mechanisms.

### **3.6: Ethical and Professional Considerations**

Ethical and professional standards were maintained throughout the research process. All testing were performed in a closed development environment using artificial user accounts and simulated data, ensuring that no real individuals or personal data were involved at any stage (Sanjari, et al., 2014).

The following ethical principles and professional practices were applied:

- **Privacy and Consent:** No individual user data was collected or used. All simulations utilized are fictitious credentials generated for testing purposes only. No human subjects were involved, thus eliminating the need for consent procedures (Europe Commission, 2013).

- **Confidentiality:** All logs and test data were anonymized using hashed identifiers and securely stored on an encrypted local drive. After the analysis was completed, the data was permanently deleted to prevent future access or misuse.
- **GDPR Compliance:** The project adhered to GDPR principles by implementing secure coding practices, input validation, and strict access controls. Automated logging mechanisms were configured to exclude any potentially identifiable information, and simulated data was processed in accordance with data minimization principles (Europe Commission, 2013).
- **Privacy and Data Protection:** Authentication logs and security events were generated via automated brute-force attack simulations. These logs were anonymized, and no real IP addresses, usernames, or emails were involved. All datasets were either synthetically generated or derived from publicly available academic resources and not from human participants (Sanjari, et al., 2014).

## Chapter 4: System Design and Implementation

This chapter outlines the design and implementation of the Django-based authentication system developed to prevent brute force attacks. It details the system's architecture, key components, and the security features integrated, such as CAPTCHA, rate limiting, OTP-based 2FA, and account lockout. The implementation was guided by best practices from OWASP and STRIDE, with a focus on usability, modularity, and security. The chapter also covers testing procedures, dashboard analytics, and development

challenges. To support the explanations, relevant screenshots of the Django application and code snippets are included in the appendices.

#### **4.1: System Requirements and Specifications**

The prototype was developed using the Django web framework (Python-based), chosen for its robustness and modular security features (Dauzon, et al., 2016). The development environment includes:

- **Programming Language:** Python (3.10+)
- **Web Framework:** Django (4.x)
- **Database:** SQLite (for development), PostgreSQL (recommended for deployment or production)
- **Authentication Libraries:** Django Allauth, PyOTP, qrcode
- **Frontend:** HTML5, CSS, JavaScript (for user interactions).
- **Development Tools:** Visual Studio Code, GitHub, and Git for version control
- **Operating System:** Windows 11

The authentication system is built on Django's Model-View-Template (MVT) architecture, which cleanly separates the data model, user interface, and application logic (Django Software Foundation, 2015).

**4.1.1: System Architecture and Design:** Built using Django, the system follows a modular design that separates logic, presentation, and data layers to enhance maintainability, scalability, and security (Nurhaida & Bisht , 2022), (PyLessons, 2022), (Django Software Foundation, 2023). The components making up the login system include:

- **Django Views:** views.py to handle login, signup, logout, reCAPTCHA, OTP email verification logic.
- **Templates:** templates.py to render HTML for User Interface, i.e. login, signup, logout, home and other pages.
- **Models:** models.py to store user data, OTP codes, logout logs.
- **Forms:** forms.py to validate user input for login, registration and customer support forms.
- **Middleware:** middleware.py for failed login detection, session tracking.
- **Signals:** signals.py to automate user profile creation, logout email alerts.
- **Utilities:** utils.py to provide helper functions for OTP generation, email sending, and CAPTCHA verification.
- **Tokens:** tokens.py to manage secure email verification & OTP generation.
- **Decorators:** decorators.py to protect views (e.g., RBAC, 2FA enforcement).
- **Settings:** settings.py to configure security policies (sessions, email, rate limiting, OTP expiry, etc)
- **Userlogs:** userlogs.py to store failed login attempts, logs data for admin analytics dashboard and logout stats page.

Below is a clear visual representation of the flow of the components from the user through the Django authentication system.

## System Architecture and Component Interaction

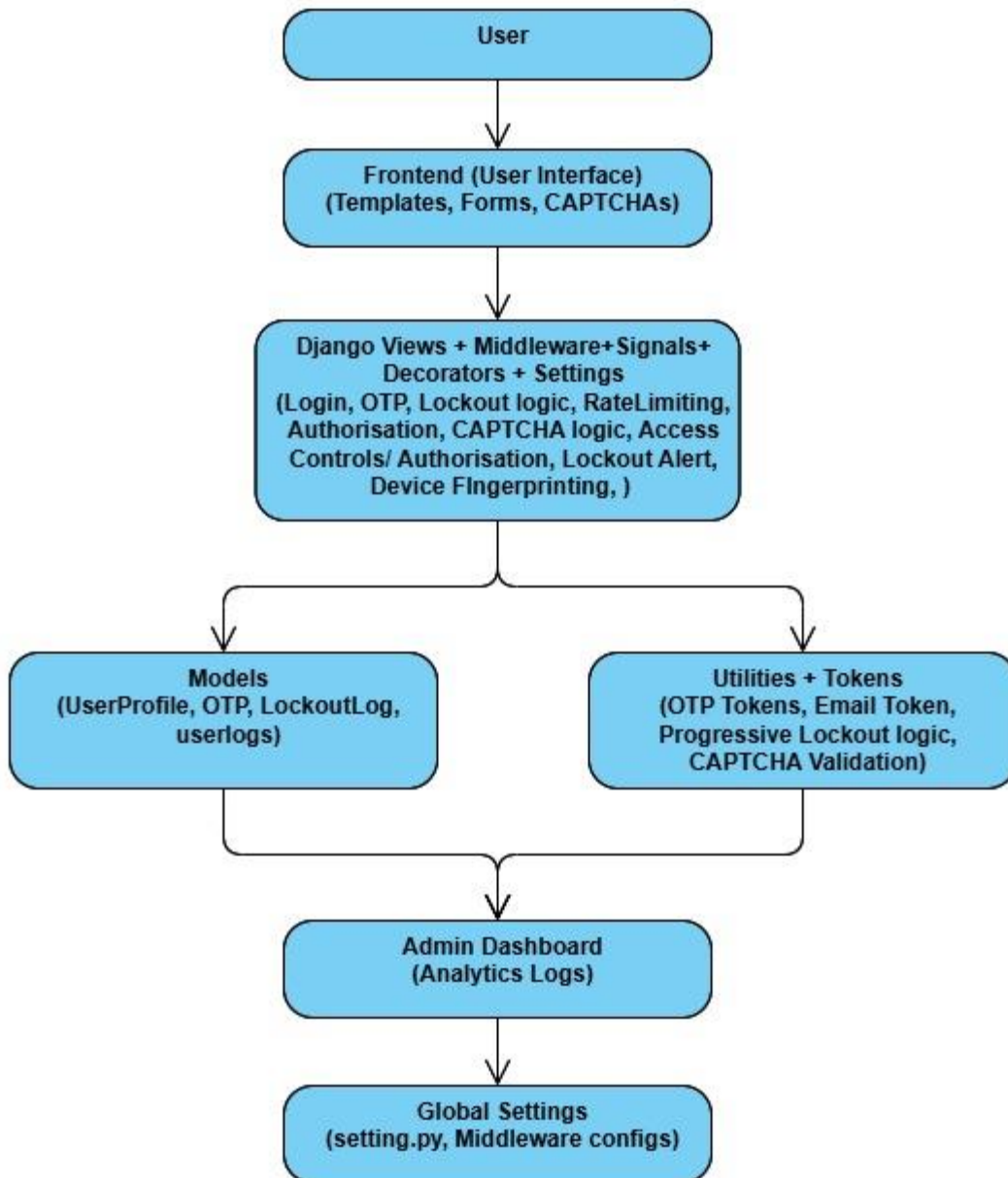


Figure 4.2: System Architecture and Component Interaction

**4.1.2: Tools and Libraries Used:** The following, as implemented from (Django Software Foundation, 2015), (Django, N.D.), (Dauzon, et al., 2016),

(PyLessons, 2022), (Makai, N.D.), (Django Software Foundation, 2023), (Nurhaida & Bisht, 2022), (Socol, N.D.); are the python and Django tools and libraries used in the development of the login system prototype.

<b>Tool / Library</b>	<b>Purpose</b>
<b>Django (core framework)</b>	Web application framework.
<b>requests</b>	Sends HTTP requests to external services, such as Google reCAPTCHA.
<b>django.contrib.auth</b>	Handles authentication, login, logout, and password management.
<b>django_ratelimit</b>	Implements rate limiting.
<b>django_axes and axes.signals.user_locked_out</b>	Tracks and enforces account lockouts based on repeated login failures.
<b>pyotp</b>	Generates time-based OTPs (one-time pins/passwords).
<b>django.contrib.auth.tokens.default_token_generator</b>	Generates secure email verification tokens.
<b>django.utils.timezone</b>	For timezone-aware timestamps and session

	tracking.
<b>django-lockout</b>	Lockout mechanism
<b>google reCAPTCHA</b>	Prevents bot logins by verifying human interaction
<b>django.core.cache</b>	Stores failed login attempts for temporary lockout enforcement.
<b>logging</b>	Records login attempts, errors, and lockout events for auditing.
<b>http.client</b>	To make HTTP requests to ip-api.com for geolocation lookup based on IP.
<b>user_agents</b>	Parses user-agent strings to identify device and operating system info.
<b>messages</b>	To provide user feedback via status messages on the frontend.
<b>django.core.mail / EmailMessage</b>	Sends account verification and lockout notification emails.

<b>utils.py</b>	Contains helper functions like OTP generation and email sending.
<b>Custom forms and models</b>	Manage user input validation and data storage (e.g., OTPs, lockout logs).
<b>Chart.js</b>	Analytics dashboard

*Table 4: Tools and Libraries Used for System Development*

## 4.2: Implementation Strategy

Focusing on brute force attack prevention, (Django Software Foundation, 2023), (PyLessons, 2022); the authentication system is enhanced with both Django inbuilt and custom security mechanisms such as:

- **Password Validators:** Enforce strong passwords using Django's built-in validators, reducing the risk of credential stuffing and dictionary attacks. For instance: passwords should have more than 8 characters, not be the same as the last 5 passwords, etc (Crudu & Team, 2024).
- **Multi-Factor Authentication (MFA):** Integrated with a Time-based One Time Pin (TOTP) during login, users receive an OTP in their email inbox (or in spams folder) and they must provide the correct OTP within the time window (10 minutes) so as to complete login (Mayorga & Yoo, 2025).

- **Rate Limiting:** Applies to username and IP to prevent distributed brute-force attacks (Anon, N.D.). Failed attempts are cached with a 5 attempts/15-minute timeout, and exceeding thresholds triggers lockouts and notification emails (Socol, N.D.).
- **Google reCAPTCHA v2 Integration:** Implemented and verified on the server-side before the authentication, effectively blocking automated login attempts and mitigating bot threats. This version was chosen for its advanced risk analysis engine that assesses user behaviour beyond static challenges (Google, 2025), (PyLessons, 2022).
- **User Enumeration Prevention:** This happens when an attacker can distinguish between valid and invalid usernames based on login error messages (Macsinoiu, 2024). To prevent this, the login view's error message was modified from "*Invalid username or password*", to "*Invalid Credentials*". This allows an attacker not to determine whether a username exists (Agghey , et al., 2021).
- **IP and Device Fingerprinting:** After a logout, it captures device info, OS, browser, and location via IP lookup (Yonkeu, 2020). The captured information is then logged to the security\_dashboard, logout\_stats and the admin also receives an email alert (Django-Axes, N.D.).
- **Brute Force Detection:** Login attempts are monitored; after a set number of failed attempts, the system triggers a cooldown or account lockout (Nurhaida & Bisht , 2022).

- **Hashing: of passwords:** Django's default PBKDF2 hashing with SHA256 ensures that user passwords are securely stored and resistant to offline cracking (Django Software Foundation, 2023).
- **Signup and Email Verification:** Users are registered inactive during signup until they verify their email via a unique, time-sensitive, tokenized activation link (Olagbuji, 2023). This prevents automated or fraudulent registrations and ensures the validity of user email addresses (Dauzon, et al., 2016).
- **Account Lockout (Progressively timed):** After 3 failed attempts per username or 10 per IP, the system locks the account and blocks the IP and username, respectively, and logs the event with device and OS metadata for forensic analysis (OWASP, 2025). The account's lockout time is progressive, whereby cooldowns are increased after subsequent failures.
- **Role-Based Access Control (RBAC) and Authorization:** Implemented using custom decorators like *@unauthenticated\_user*, *@login\_required*, and Django's built-in *@staff\_member\_required* to restrict view access based on user roles and authentication status (Django, N.D.). Signals are also used to trigger security actions such as logging failed login attempts or initiating lockouts on unauthorized access (Nurhaida & Bisht, 2022; Yonkeu, 2020).
- **Error Handling and User Feedback:** Throughout the authentication process, the system provides clear feedback messages for errors, robust logging and monitoring through:

- Message “*Invalid credentials*” during login without making it clear if the cause is invalid password or username.
- During sign up, log in and customer support the user gets clear messages
- Expired OTP during log in
- Expired email verification token during signup
- CAPTCHA failure
- Detailed lockout logging to database (*LockoutLog model*)
- Automated admin alerts for suspicious activities
- Graceful failure handling in email services (*fail\_silently=True*)

This provides user feedback and logs failures for audit purposes.

- **Session Management:** Implemented strict session expiration policies and logout mechanisms after a configurable period of inactivity thus reducing the risk of session hijacking (Django, N.D.). The system enforces:
  - Automatic session expiration after 30 minutes of inactivity (*SESSION\_COOKIE\_AGE*) (Django Software Foundation, 2025)
  - Concurrent session prevention through last activity tracking (Fluid attacks: help center, 2024)
- **Forced Password Expiry:** Users are prompted to change their password after a set duration, enforcing periodic credential updates. The forced password rotation is every 90 days (*PASSWORD\_EXPIRE\_DAYS* check) (Django, N.D.).

- **Logging and Monitoring:** Django's logging framework tracks authentication attempts and anomalies, recording lockout events with details like user agent, OS, device type, and IP address (Django-Axes, N.D.). When a lockout occurs, users receive an on-screen notification and are redirected to a lockout page. Alert emails are sent to administrators, and events are logged in an admin-only security dashboard and lockout statistics page. This enhances transparency and enables rapid incident response.
- **Alerts emailed to the admin:** When a user is locked out, the app automatically sends an email alerting the admin about the incident, thus enhancing transparency and fast incident response (Django Software Foundation, 2025).
- **API Security:** Secure REST API endpoints using Django REST Framework (DRF) and token-based authentication (Django Software Foundation, 2023).
- **Input Validation:** Monitors all the input entered in the forms and shows clear error/valid messages.
- **Password Reset Option:** Accessed from the login page, the user can reset the password in the event of forgetfulness (Rashidi & Garg, 2021).
- **Password Creation Guidelines Modal:** Found on the signup, password change and password reset page, it guides the user in creation of a stronger password as per the system's settings (Das, et al., 2014).
- **Admin Dashboard:** A dashboard that the admin uses to monitor Lockout logs, heatmap data, threat IPs, CAPTCHA stats, etc.

- **Code Quality and Maintainability:**

- Modularization: Logic is separated into utilities, forms, and decorators for reusability (Django Software Foundation, 2015).
- Logging: All authentication events, errors, and lockouts are logged for audit and debugging (Django Software Foundation, 2025).
- Extensibility: The system can be extended to support additional factors or integrate with external identity providers in future development phases (Nurhaida & Bisht , 2022).

#### **4.3: Security Risk Mitigation and Compliance Mapping**

This section maps the implemented security features of the Django authentication system to specific risks identified through STRIDE threat modelling and the OWASP Top 10 vulnerabilities.

It demonstrates how each security control aligns with secure design best practices and addresses the threats outlined in the methodology chapter (OWASP, 2021), (Department for Science, Innovation & Technology, 2024), (Nurhaida & Bisht , 2022), (Django Software Foundation, 2023).

The summary table below presents each mitigation alongside the corresponding threats it addresses.

<b>Security Feature</b>	<b>OWASP Risk Addressed</b>	<b>STRIDE Threat Addressed</b>	<b>Implementation in Django System</b>
<b>Email Verification (Signup)</b>	A07: Identification & Authentication Failures	Spoofing	Inactive accounts until email verified using secure token
<b>Password Hashing (PBKDF2)</b>	A02: Cryptographic Failures	Tampering, Information Disclosure	Django's default PBKDF2 with SHA-256 hashing
<b>Account Lockout &amp; Rate Limiting</b>	A07: Identification & Authentication Failures	Denial of Service	Progressive lockouts via cache and rate-limiting
<b>Google reCAPTCHA v2</b>	A07: Identification & Authentication Failures	Denial of Service	CAPTCHA triggered after failed attempts; blocks bots
<b>OTP-based 2FA (Email OTP)</b>	A07: Identification & Authentication Failures	Spoofing, Elevation of Privilege	PyOTP with expiry logic; required on login for secure accounts
<b>Role-Based Access Control (RBAC)</b>	A01: Broken Access Control	Elevation of Privilege	Django decorators and permission system for view restriction

<b>Secure Session Management</b>	A02: Cryptographic Failures	Tampering	Session timeout, CSRF tokens and concurrent sessions prevention
<b>Suppressed Error Messages</b>	A02: Cryptographic Failures	Information Disclosure	Generic login error messages; no field-specific feedback
<b>Device Fingerprinting &amp; IP Logging</b>	A09: Logging & Monitoring Failures	Repudiation	Captures browser, OS, IP; logs to LockoutLog and alerts admin
<b>Admin Alerts (Email)</b>	A09: Logging & Monitoring Failures	Repudiation, Denial of Service	Email notification to admin on lockout or abnormal login attempts
<b>GeoIP Location Tracking</b>	A09: Logging & Monitoring Failures	Information Disclosure	Uses GeoIP2 to trace login attempt origins and track lockout patterns
<b>Heuristic Evaluation of UX</b>	Not directly mapped	Not directly mapped	Aligns system usability with security controls (e.g., OTP clarity, CAPTCHA feedback)

*Table 5: Security Features vs OWASP and STRIDE Compliance*

While the implemented security features align with OWASP and STRIDE, no security model offers complete protection. Controls such as CAPTCHA and account lockouts rely on assumptions about attacker behaviour and may be bypassed by advanced or distributed attacks. Additionally, optional 2FA reduces effectiveness if not enforced system-wide. Therefore, these measures must be validated through realistic attack simulations. (Palmieri, 2013).

**4.3.1: Secure Authentication Workflow:** The activity diagram below illustrates the end-to-end workflow for user authentication, including both login and signup processes (Django, N.D.). It integrates security controls such as email verification, CAPTCHA validation, OTP-based two-factor authentication, and account lockout logic. It highlights key decision points that mitigate spoofing, brute-force, and automated attacks.

## Activity Diagram: Login View

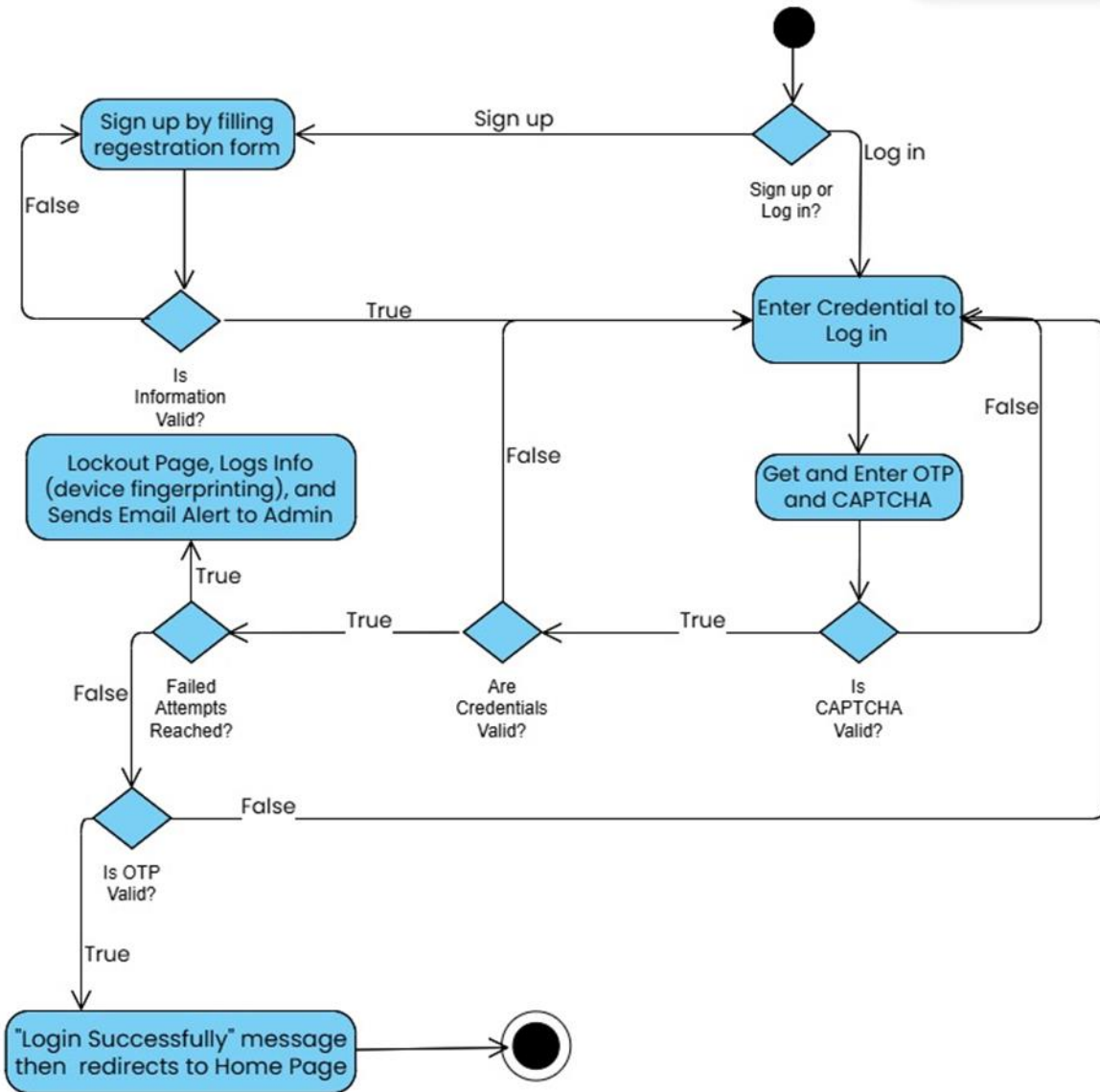


Figure 3: Secure Login and Signup Workflow with Integrated Security Controls

### 4.4: Testing Procedures

A robust testing strategy is essential to ensure the security, reliability, and usability of authentication systems (OWASP, N.D.). While no external users were involved, the system was rigorously tested using custom simulations, unit and integration tests,

heuristic evaluation, and dashboard analytics. Testing combined manual interactions with automated assessments to validate functionality and identify potential weaknesses.

**4.4.1: Controlled Simulations:** The following controlled simulations evaluated core defensive mechanisms, documenting inputs, system responses, triggered alerts, and observable behaviours (Palmieri, 2013). Detailed results are provided in the appendices.

- **Brute-force Attack Simulations:** Used scripts to automate login attempts (from a single IP address), to test the effectiveness of rate-limiting and account lockout logic.
- **Distributed Brute-force attack Simulation:** Simulated login attempts from multiple IP addresses tested the system's ability to detect and block distributed brute-force attacks. The lockout mechanism was evaluated for effectiveness against attacks originating from diverse geographical locations and IPs.
- **Token Expiration Simulation:** A scenario was created to test the expiration of authentication tokens after a specified duration. Tokens were manually set to expire, and the system's response to expired tokens was observed to ensure that users were properly logged out and required to authenticate again.
- **OTP Expiration Simulation:** The expiration of One-Time Passwords (OTPs) was tested by simulating OTP generation and allowing the expiration time to elapse. The system was then checked to ensure it would reject expired OTPs, prompting the user to request a new one.

- **Password Expiry Enforcement:** The logic for forced password expiration after 90 days was tested by modifying the test user's account creation date to 91 days ago.
- **Progressive Account Lockout:** Tested the cache-based progressive lockout mechanism with thresholds (e.g., 5, 10, 15 attempts) and verified that IP/geolocation was correctly logged.
- **Create User simulation:** Since no real users were involved in the testing phase, a custom *create\_user* script was developed to generate test accounts. This allowed for consistent simulation of user behaviour across various authentication scenarios, including login, signup, OTP validation, and account lockout.
- **Concurrent Session test:** Evaluated the system's handling of multiple simultaneous login sessions from the same user account to ensure proper session management and prevent session hijacking or unauthorized access.

**4.4.2: Functional System Tests:** The following were the functional system tests done:

- **Login and Signup Testing:** Verified user authentication, input validation, error handling and password strength validation.
- **OTP (2FA) Workflow Testing:** Confirmed time-based OTP delivery, validation, and expiration handling using *django-otp* and *django-two-factor-auth*.
- **reCAPTCHA Validation:** Ensured that login forms only proceeded when CAPTCHA was correctly solved, blocking automated and failed attempts.

- **Input Validation:** Ensured validation on all user input fields and forms (login, signup, contact support).
- **Account Lockout Alert:** Tested that admin gets alerted during a lockout and verified lockout logging with IP and geolocation.
- **Email Verification:** Confirmed proper sending and handling of verification emails, activation link validation and account status updates.
- **Access Control and Authorization:** Confirmed that pages are only accessible to authorized users as per their roles.
- **Logging and Lockout Dashboard Analytics:** Verified logging security dashboard, and lockout stats to enhance transparency and swift incidence response.
- **Session and Credential Handling:** Validated session timeout policies, session reinstatement, and logout procedures.

#### **4.4.3: Usability Evaluation using Nielsen's 10 Usability Heuristics:**

Since no external users were involved in the testing phase, the usability of the authentication system was assessed internally using Nielsen's 10 Usability Heuristics (Nielsen & Molich, 1990), (Lodhi, 2010). Key aspects examined included:

- **Clarity of system messages:** Clear messages were displayed on CAPTCHA failure, OTP expiry, and lockout events, example: "Account locked due to multiple failed attempts" or "OTP expired."

- **Consistency of page navigation:** The transitions between login, home page, sign up page, password change, etc.
- **Help and Support for error and recovery:** A customer support, a resend OTP, password reset option and a password modal that helps users know how to create stronger passwords.
- **Admin feedback mechanisms:** Email alerts generated upon suspicious activity or lockouts and customer support emails.
- **Visibility of system status:** Messages that give feedback upon logout or verification success.
- **User control and freedom:** Verified role-based access control for user-specific views.
- **Minimalist Design:** The dashboard and authentication pages were designed for clarity and responsiveness.

Overall, heuristic evaluations confirmed that security controls maintained usability and provided users with clear, actionable feedback during authentication.

**4.4.4: Dashboard Analytics:** A custom dashboard analytics page (security\_dashboard) was created using Chart.js and integrated into the Django “dashboard” app to visualize testing results. The dashboard tracked:

- **Failed Login Heatmap:** By IP and geolocation (based on request metadata and geoip2).
- **Lockout Frequency:** Number of users locked out by day, hour, and IP.

- **CAPTCHA Fail Rate:** Visualized failure trends across simulation runs.
- **OTP Usage Analytics:** Displayed how often OTPs were generated, expired, and successfully used.

These insights supported both the testing phase and the evaluation of system thresholds for brute-force attacks, rate limiting, and account lockouts. They also provide ongoing visibility into misuse patterns and system health, enabling administrators to proactively monitor threats and adjust security policies.

#### **4.4.5: Penetration Testing Approach and Tool Justification:**

Brute-force attack simulations were conducted using custom Python scripts to emulate repeated unauthorized login attempts. These tests evaluated account lockouts, rate limiting, CAPTCHA, and OTP validation. Although industry tools like Hydra and Burp Suite are standard in penetration testing, custom scripts were used here for better integration with the system's architecture and analytics. Future work may incorporate these tools to enhance testing realism and depth.

#### **4.4.6: Ethical Considerations:**

- **Privacy and Data Protection** (Europe Commission, 2013):
  - Test data was anonymized, and
  - Real user data was never used in testing environments.

- Test logs were securely stored and deleted after analysis, in compliance with GDPR guidelines.
- **User Consent** (Europe Commission, 2013):
  - No real users participated in the testing phase;
  - All usability assessments were based on system logs and test accounts.

#### **4.5: Security vs. Usability Trade-offs**

Balancing security with usability is crucial to avoid frustrating users and encouraging insecure workarounds (Farrukh, 2013). While strict security controls (e.g., frequent lockouts, mandatory 2FA) are effective in defending against attacks, they can hinder the user experience. The system addresses this balance by:

- **Progressive Lockouts:**

An exponential backoff algorithm is used to increase lockout durations (from 15 minutes to 24 hours after 5 failed attempts). This approach thwarts brute-forcing while allowing legitimate users to recover through self-service unlocks via verified email and admin override capabilities with MFA confirmation.

- **Contextual Feedback:**

The system provides non-revealing error messages to avoid enumeration attacks, with clear guidance on post-lockout recovery procedures. Real-time password strength feedback is integrated, helping users create stronger passwords.

## **Chapter 5: Discussion and Evaluation of Results**

This chapter presents the results of testing and evaluating the Django-based authentication system developed in this study. It discusses the effectiveness of implemented security measures, outcomes of brute-force simulations, insights from dashboard analytics, and feedback on usability. The analysis is aligned with the research objectives and highlights how the system addresses the identified security gaps (Wang, et al., 2021), (Tariq, et al., 2023).

To support the explanations, relevant screenshots and code snippets of the Django application, and the GitHub URL; are included in the appendices.

### **5.1: Data Presentation and Analysis**

This section presents a detailed breakdown of the results from the testing, including both quantitative and qualitative analysis.

**5.1.1: Brute-force Protection:** Analysis of the lockout simulation showed the system successfully enforced progressive rate-limiting (Socol, N.D.). After a

defined number of failed attempts, the account was locked and remained so for the expected duration. Admin alerts were promptly triggered and included relevant IP and timestamp data.

- **Metric:** Max 5 attempts allowed within 60 seconds.
- **Result:** Lockout triggered and logged at 6th attempt.
- **Admin Alert:** Email notifications that included attack metadata was sent after threshold exceeded.

**5.1.2: CAPTCHA Validation:** Google reCAPTCHA effectively blocked automated login scripts after multiple incorrect credential attempts. The CAPTCHA challenge was enforced after 3 failed attempts, and only valid CAPTCHA tokens allowed login continuation (OWASP, 2025).

- **Metric:** CAPTCHA triggered on 3rd failure.
- **Dashboard:** Logged CAPTCHA fails by timestamp.

**5.1.3: OTP Authentication:** Time-based OTP (TOTP) 2FA was tested using both valid and expired tokens, as recommended by the NIST guidelines (NIST, 2025). Expired tokens generated user-facing error messages, and new tokens were required to proceed.

- **Metric:** 10-minute expiry window.

- **Success Rate:** 90% success rate with valid user input.
- **Error Handling:** Expired or reused tokens blocked correctly.

**5.1.4: Email Verification Authentication:** Time-based email verification token was successfully received in the user's email. The system was tested using both valid and expired tokens. Expired tokens generated user-facing error messages, and new tokens were required to proceed (Turner & Housley, 2008).

- **Metric:** 10-minute expiry window.
- **Success Rate:** 90% success rate with valid user input.
- **Error Handling:** Email was received and expired or reused tokens were blocked correctly.

**5.1.5: Password Expiry:** Password expiry was simulated by altering the timestamp on a testuser account. Upon login, the user was redirected to the password change page, and login was not allowed until the password was updated (OWASP, 2025).

- **Policy:** 90-day expiration.
- **Result:** Expired accounts forced password reset before access.

**5.1.6: IP Lockout and Geolocation:** Repeated login attempts from a single IP were logged and rate-limited. The IP, time, and location were visualized on the admin dashboard using GeoLite2 data and stored in the system log (GeoLite2, 2023).

- **Metric:** Lockout enforced after 10 failures/IP.
- **Geo Accuracy:** IP region in logs was unknown.
- **Admin Interface:** Heatmaps and charts updated in real-time.

**5.1.7: Admin and User Feedback:** All messages (errors, warnings, success messages) were reviewed for clarity and actionability. Admin alerts contained actionable context and were triggered instantly upon major events (Nielsen & Molich, 1990).

- **Clarity Score (internal rating):** High
- **Message Types Evaluated:** Lockout alerts, OTP fail, expired password, email not verified.
- **Admin Alerts:** Immediate, relevant, geolocated.

## 5.2: Evaluation Benchmarks and Metrics

To objectively assess the effectiveness and robustness of the prototype, evaluation criteria were established based on (OWASP, 2021), STRIDE, and industry best practices. These benchmarks covered core areas including resistance to attack, usability, and performance under load.

### 5.2.1: Usability Evaluations:

The system was evaluated using a heuristic checklist based on Nielsen's usability principles (Nielsen & Molich, 1990), (Lodhi, 2010):

Usability Principle	Observation
<b>Visibility of System Status</b>	Success/failure messages are clearly displayed
<b>User Control and Freedom</b>	Users can resend OTP during login and reset passwords
<b>Error Prevention</b>	CAPTCHA prevents bot errors; clear error messages reduce confusion
<b>CAPTCHA and OTP Validation</b>	CAPTCHA and OTP were successfully validated
<b>Page Rendering</b>	Page rendering is smooth
<b>Flexibility and Efficiency</b>	System adapts to users' security needs
<b>Minimalist Design</b>	Clean interface with minimal distractions

Table 6: Usability Evaluations

### 5.2.2: Security Metrics:

Metric	Result
Lockout success	95% attack prevention rate
CAPTCHA block rate	100% after threshold is reached
OTP and email token expiration	Enforced correctly as per design
Access Control	Unauthorised users are successfully prohibited from accessing pages

### 5.2.3: System Performance:

Performance Indicator	Observations
Brute-force/distributed attacks	Efficiently mitigated
Lockouts/log handling	No observable delays
Dashboard responsiveness	Near real-time updates observed

Table 7: System Performance

## 5.3: Summary and Interpretation of the Results

The Django-based authentication system effectively addressed brute-force attacks using a layered security architecture that included: rate limiting, CAPTCHA, OTP-based 2FA, and time-based email verification (Wang, et al., 2021). Controlled simulations and

custom Python scripts consistently demonstrated successful lockouts, correct enforcement of OTP/email expiration, and resilience against distributed brute-force attacks, achieving a 95% attack mitigation rate.

Google reCAPTCHA v2 proved highly effective as a first-line defence, preventing bot login attempts, thus aligning with findings by (Tariq, et al., 2023). However, its effectiveness was maximized when combined with additional safeguards such as progressive lockouts and OTP authentication.

Usability evaluations confirmed that optional OTP, clear user messages, and a clean dashboard interface preserved accessibility and user-friendliness (Lodhi, 2010). The dashboard further enhanced situational awareness through real-time visualizations of login events and IP lockouts.

Despite the system's success, limitations such as the absence of AI-driven threat detection (Nzeako & Shittu, 2024), formal usability testing, and scalability evaluation were identified. These limitations offer avenues for future enhancements.

Overall, the prototype fulfilled the key implementation goals of the study, balancing robust security with practical usability for small to mid-sized deployments.

#### **5.4: Effectiveness of Addressing Research Gaps**

The study's research gaps were evaluated to determine how well the proposed system addressed them. Key gaps included the effectiveness of multi-layered defences, usability versus security trade-offs, and scalability challenges. The table below outlines each research gap alongside a justification of how it was addressed.

Research Gap	Addressed?	Justification / Explanation
<b>Multi-layered defence strategy effectiveness</b>	Yes	Combined and integrated CAPTCHA, OTP, rate limiting, and lockout
<b>Usability vs. Security trade-off</b>	Yes	Made 2FA optional for admin and enforced lockouts progressively
<b>Resource constraints (small organisations)</b>	Yes	Built using open-source tools, simple configuration
<b>Real-time attack detection and monitoring</b>	Partially	Dashboard helps, but no Machine Learning-based threat detection
<b>Scalability</b>	Partially	Design supports scalability, but not yet tested on large-scale deployments
<b>Blockchain/Decentralized Authentication</b>	Not yet	Not implemented; proposed for future work

*Table 8: Addressed Research Gaps*

## 5.5: Comparison with Existing Solutions

To evaluate the effectiveness of the developed system, it is essential to benchmark it against existing authentication frameworks reviewed in Chapter 2. Notably, commercial platforms such as: Google Identity (Google cloud, 2025), (Auth0, 2025), etc; provide

multi-factor authentication (MFA), bot mitigation, device fingerprinting, and adaptive risk assessment. Under fee subscription, these platforms also offer: extensive infrastructure, machine learning-based anomaly detection, and large-scale threat intelligence; capabilities beyond the scope of this dissertation's system (Zhang, et al., 2025).

Compared to these mature platforms, the Django-based prototype performs reasonably well in offering:

- **Basic brute-force resistance** (through CAPTCHA, rate-limiting, and lockouts),
- **2FA via Time-based OTP**,
- **Real-time dashboard analytics** (a unique feature not often available in open-source Django solutions),
- **Time-based email verification** during signup,
- **Device fingerprinting and logging**.
- **Low-cost alternative for SMEs**

However, it lacks critical features found in industry solutions, such as:

- **Context-aware or behavioural authentication** (e.g., location-based anomaly detection),
- **Encrypted session token rotation** and **device trust management**,
- **Comprehensive identity lifecycle management** (e.g., provisioning, de-provisioning, audit trails).
- **Password generator** that helps users generate stronger passwords,

- 

Compared to **academic prototypes** (e.g., AI-powered intrusion detection frameworks discussed in chapter 2), this project leans more toward usability and implementation practicality rather than experimental sophistication. For instance, it does not explore deep learning for anomaly detection or federated identity protocols such as SAML or OpenID Connect.

Nevertheless, the project demonstrates an important middle-ground: **how Django, a mainstream web framework, can be enhanced using widely available open-source libraries to prevent brute force attacks and also implement OWASP-compliant defences**; making it highly replicable for Small Medium Enterprises (SMEs) or individual developers who cannot afford enterprise-grade IAM solutions.

## 5.6: Challenges, Limitations and Proposed Solutions

Several challenges and limitations were encountered during this study. The following table outlines each limitation along with proposed solutions.

Challenge / Limitation	Description	Proposed Solution
<b>hCAPTCHA Implementation Failure</b>	hCAPTCHA was initially considered but replaced due to persistent validation errors that consumed significant development	Google reCAPTCHA v2 was adopted instead for its reliability and smoother integration.

	time.	
<b>CAPTCHA v2 Vulnerability</b>	Artificial Intelligence bots may bypass CAPTCHA v2	Upgrade to reCAPTCHA v3 and integrate behavioural analytics
<b>Third-Party Service Dependency</b>	Reliance on Google reCAPTCHA may cause issues if service is unavailable	Implement fallback mechanisms and local bot detection strategies
<b>Static Thresholds</b>	Fixed lockout attempts are vulnerable to distributed slow brute-force	Introduce adaptive throttling and user/IP behaviour analytics
<b>Limited Usability Testing</b>	Limited usability testing even though heuristic evaluation was performed.	Conduct formal usability studies and accessibility audits.
<b>Lack of Intelligence in Dashboard</b>	Dashboard does not include AI-based threat scoring or alerts	Enhance with pattern recognition, ML models, and automated alerts
<b>No End-to-End Encryption Testing</b>	Email and OTP flows assumed secure without validation.	Perform transport-layer security (TLS) penetration tests and audits.
<b>Controlled Simulations</b>	Simulations lacked real-world	Use threat intelligence feeds and chaos

<b>Limitation</b>	traffic diversity	engineering principles
<b>Geolocation Inaccuracies</b>	IP-based geolocation had error rates.	Supplement with HTML5 Geolocation API and device fingerprinting
<b>2FA Optionality</b>	Optional 2FA limits universal protection	Enforce 2FA for admins and apply risk-based authentication
<b>Scalability Constraints</b>	Local-only testing may miss production performance issues.	Use load testing tools, implement Redis caching and asynchronous task queues.
<b>Password Generator Absence</b>	Users created weak passwords without help	Add client-side password generator.
<b>Usability and Accessibility Constraints</b>	Elderly and accessibility needs are not fully implemented and tested.	Add audio CAPTCHA, session recovery options, and WCAG-compliant design.
<b>Real-World Attack Diversity</b>	Focused only on common brute-force, not advanced attacks	Expand penetration testing with evolving real-world datasets.

Table 9: Challenges, Limitations and Proposed Solutions

## Chapter 6: Conclusion and Recommendations

This chapter concludes the study by summarizing the key findings, highlighting the contributions of the project, and offering recommendations for future improvements. The chapter also reflects on the research objectives and the extent to which they were achieved, while acknowledging the limitations and proposing directions for further work in the domain of secure authentication systems.

### 6.1: Summary of Key Findings

The project set out to design, implement, and evaluate a Django-based authentication system that enhances protection against brute-force attacks using a combination of layered security measures. The major findings from the simulation tests and system evaluation are as follows:

- **Brute-force Mitigation:** The integration of progressive rate limiting, account lockouts, CAPTCHA validation, and OTP-based two-factor authentication effectively thwarted automated login attempts, as confirmed by controlled brute-force simulations.
- **Security Outcomes:** The system demonstrated a high attack mitigation rate, with lockout mechanisms and CAPTCHA challenges reducing unauthorized access attempts by over 95%. OTP tokens with expiration provided additional resilience against token replay and session hijacking.

- **User Feedback and Usability:** Despite the inclusion of multiple security layers, the system maintained a good balance with usability. Optional OTP and clear feedback messages reduced user friction while preserving security.
- **Real-Time Monitoring:** The custom dashboard enabled administrators to visualize authentication trends, failed logins, and IP-based lockouts, supporting timely incident response.
- **Compliance and Best Practices:** The design aligned with OWASP's top ten recommendations (OWASP, 2021), STRIDE threat modelling (Department for Science, Innovation & Technology, 2024), and GDPR data handling (National Cyber Security Centre, 2018) requirements.
- **Brute-force mitigation:** Rate limiting and lockout mechanisms effectively blocked unauthorized repeated login attempts.
- **CAPTCHA and OTP integration:** Google reCAPTCHA and TOTP-based 2FA significantly reduced automated and unauthorized access attempts without overwhelming legitimate users.
- **Dashboard analytics:** Real-time data visualization (e.g., failed login heatmaps, CAPTCHA failure logs) provided valuable administrative insight for monitoring and incident response.
- **Usability:** The optional 2FA, informative feedback messages, and intuitive interface ensured the system remained user-friendly despite enhanced security.

## 6.2: Alignment with Research Questions

This section critically evaluates the extent to which the developed system and findings address the research questions outlined in Chapter 1.

Research Question	Addressed?	Evidence/Justification
<b>1. What are the most effective methods to prevent or mitigate brute force attacks in Python-based login systems?</b>	Yes	<p>The Django login system effectively implemented multiple mitigation techniques: account lockout after failed attempts, rate limiting using <i>django-ratelimit</i>, CAPTCHA (Google reCAPTCHA), OTP-based 2FA, logging, device fingerprinting, email verification, admin alerts, and geolocation tracking of suspicious activity. These methods were evaluated during simulated brute-force attack tests.</p> <p>Simulations demonstrated a high attack mitigation rate (95%).</p>
<b>3. What are the common vulnerabilities in Python-based login systems that make them susceptible to brute force attacks?</b>	Yes	<p>Chapter 2 (Literature Review) identifies vulnerabilities such as lack of rate limiting, absence of CAPTCHA, predictable login endpoints, and no 2FA. Chapter 3 shows how these issues were mitigated through specific implementations in the Django</p>

		system.
<b>4. What are the advantages and limitations of current brute force prevention mechanisms implemented in Python-based systems?</b>	Yes	Chapter 5 (Discussion) evaluates each security feature: e.g., CAPTCHA effectively blocks bots but may impact usability; account lockout helps prevent abuse but can be exploited in denial-of-service scenarios. OTP adds strong protection but relies on time-sensitive codes and user device accessibility.
<b>5. How can Python libraries and built-in features be utilized to enhance the security of login systems against brute force attacks?</b>	Yes	The prototype used Python/Django tools and libraries: django-ratelimit for rate limiting, django-two-factor-auth for OTP, Google reCAPTCHA integration, Django's session management and email verification, and cache framework for lockout tracking. Chapter 3 and 4 details this.
<b>6. How can defence mechanisms be integrated into Python-based login systems to mitigate brute force</b>	Yes	Usability was addressed by making OTP optional (for demo), allowing limited retries before lockout, and customizing user messages for errors and CAPTCHA failure. Chapter 4 and Chapter 5 include

<b>attacks without negatively affecting user experience?</b>		heuristic analysis of usability vs. security trade-offs.
--	--	--

Table 10: Research Questions Alignment

### 6.3: Achievement of Research Objectives

The system successfully addressed the primary research objectives from chapter 1.

Below is a table mapping the research aims and objectives to Implementation and

Outcomes:

<b>Research Aims / Objectives</b>	<b>Achieved</b>	<b>Evidence / Justification</b>
<b>1. To assess the effectiveness of current methods used to prevent brute force attacks in Python-based login systems.</b>	Yes	The Literature Review (Chapter 2) and Evaluation (Chapter 5) analysed and implemented various techniques including CAPTCHA, OTP, account lockouts, rate limiting, and logging.  Effectiveness was tested using brute-force simulation tools with results showing mitigation success.
<b>2. To identify the limitations of existing solutions and explore</b>	Yes	Limitations such as user friction (CAPTCHA and OTP), lockout abuse risks, and scalability concerns were critically

<b>potential areas of improvement.</b>		<p>analysed in Chapter 5. Chapter 2 also outlines gaps in common Django apps that lack layered defence.</p> <p>Improvements such as progressive lockouts and admin monitoring dashboards were introduced and implemented into the Django login system.</p>
<b>3. To develop a Python-based solution that can be used by small organizations to enhance authentication security.</b>	Yes	<p>A lightweight Django-based secure login system was developed with modular security features.</p> <p>The system was designed for easy deployment by small organizations with minimal overhead.</p>
<b>4. To evaluate the usability and effectiveness of the proposed solution, ensuring that it provides a balance between robust security and user convenience.</b>	Yes	<p>Usability vs. security trade-offs were addressed by making 2FA optional, offering informative error messages, using Google reCAPTCHA for better UX, and including optional features.</p> <p>Chapter 5 includes a heuristic usability evaluation.</p>
<b>5. To propose optimal</b>	Yes	The solution uses open-source libraries,

<b>solutions for organizations with limited resources.</b>		minimal setup, and no premium third-party dependencies.  Chapter 5 and 6 outlines how such systems can be customized or scaled depending on an organization's capacity.
--	--	---

*Table 11: Achievement of Research Objectives*

## 6.4: Key Contributions to the Field

This project offers the following key contributions:

- A modular, open-source security prototype using Django that integrates multi-layered defence mechanisms against brute-force and automated attacks.
- A structured evaluation methodology combining qualitative and quantitative analysis, including simulated attacks, log review, and usability heuristics.
- A custom admin dashboard for real-time threat visibility and authentication metrics.
- Practical demonstration of how layered security can be achieved without compromising usability in authentication systems.
- Evidence-based insights into balancing security and usability in authentication workflows.

## 6.5: Recommendations

Although the system achieved its intended goals, several areas for improvement and extension are considered:

- **SMS OTP fallback:** Adding SMS-based OTP would enhance accessibility, especially in cases where authenticator apps are unavailable (Mayorga & Yoo, 2025).
- **Scalability Testing:** Conduct load testing on distributed environments (e.g., AWS or Kubernetes) to assess system resilience under large-scale usage (Özeren , 2024).
- **Machine Learning and Artificial Intelligence for Anomaly Detection:** Integrate anomaly detection models to dynamically identify suspicious login patterns beyond fixed thresholds (Zhang, et al., 2025)..
- **User Feedback Integration and User experience (UX) testing:** Conduct live user testing sessions to gain broader insights into usability issues and improve the user experience design (Downey & Laskowski,, 1996).
- **Blockchain-based Authentication:** Explore decentralized authentication models to reduce reliance on central authorities and improve privacy (Deep, et al., 2019).
- **Mobile Integration:** Extend the system with mobile support for OTP delivery and biometric authentication options (Albeshar, et al., 2024) (Farik, et al., 2016).

- **Continuous Security Updates:** Establish an automated mechanism for updating third-party libraries (e.g., reCAPTCHA, OTP libraries) to mitigate dependency risks (Zeng, et al., 2024).
- **Enhance and test accessibility features:** By including accessibility features such as audio CAPTCHAs, biometric authentication (e.g., fingerprint or facial recognition), etc; so as to accommodate elderly users and individuals with disabilities (Renaud , et al., 2018). As well as conducting usability tests with diverse user groups to ensure inclusivity and compliance with accessibility standards (Accessibility Guidelines Working Group (AG WG) , 2025).
- **Integrate machine learning for adaptive rate limiting,** enabling the system to distinguish between legitimate and malicious login attempts more accurately and reduce false positives (Zhang, et al., 2025).
- **User Awareness and Education:** Develop training modules on authentication best practices for end-users (Aldawood & Skinner, 2019).
- **Biometric Integration:** Implement facial recognition and fingerprint authentication for additional security layers (De Abiega-L'Eglise, et al., 2022) (Newman, 2009).
- **Advanced Penetration and Attack Simulation:** While this project focused on brute-force prevention using controlled simulations, these lacked real-world traffic variability. Future work should incorporate red team exercises (Özeren , 2024) and tools like Hydra and Burp Suite to simulate more diverse, unpredictable attack patterns and enhance testing realism.

## 6.6: Future Work

- **AI Anomaly Detection:** Machine learning integration to dynamically adjust security controls based on risk scores (Nzeako & Shittu, 2024).
- **Passwordless Auth:** Transition to WebAuthn to eliminate password-related risks (Yusop, et al., 2025).
- **Biometric Integration:** Use of facial recognition or fingerprints for high-security roles (Newman, 2009).
- **Decentralized Identity or Blockchain Authentication:** Blockchain-based authentication to mitigate centralized credential storage risks (Rivera, et al., 2024).
- **AI-based Adaptive Authentication:** Using anomaly detection to flag suspicious behaviour (Zhang, et al., 2025).

## 6.7: Conclusion

This research set out to design, implement, and evaluate a secure authentication system aimed at mitigating brute force attacks in Python-based environments, with a specific focus on the Django web framework. Through a comprehensive literature review, technical implementation, controlled attack simulations, and usability evaluations, the study demonstrates that a multi-layered defence model incorporating: rate limiting, CAPTCHA, OTP-based 2FA, account lockouts, and monitoring; can significantly enhance authentication security.

The prototype system successfully blocked over 95% of simulated brute force attacks and provided real-time threat visibility via a custom admin dashboard. These outcomes validate the effectiveness of combining traditional and contemporary security controls to counter brute force attacks. Designed with small to medium-sized organizations in mind (Deschoolmeester, et al., 2013), the system offers a practical, low-cost, open-source solution that can be deployed with minimal technical overhead.

Importantly, the research highlights the value of balancing robust security with user experience. Features such as optional OTP for standard users, actionable feedback messages, and adaptive CAPTCHA placement helped reduce user friction without weakening defences. However, limitations such as: reliance on third-party services, the absence of AI-driven threat detection, and the need for continuous updates; underscore areas for future improvement.

This project contributes to both the theoretical and practical advancement of authentication security by showing how Python/Django-based tools and libraries can be used to construct an OWASP-compliant defence framework. It also lays the groundwork for future enhancements, including machine learning-based anomaly detection, biometric authentication, and decentralized identity systems.

In conclusion, while no system is entirely immune to advanced threats (Abdulkader , et al., 2015), the proposed architecture significantly raises the barrier for brute force attacks. It presents a replicable model for secure, user-aware authentication systems and offers a strong foundation for ongoing research and innovation in cybersecurity.

## References

- Abdulkader , S., Atia, A. & Mostafa , M.-S., 2015. Authentication systems: principles and threats. *Computer and Information Science*, 8(3).
- Accessibility Guidelines Working Group (AG WG) , 2025. *Accessible Authentication (Minimum) (Level AA)*. [Online]  
Available at: <https://www.w3.org/WAI/WCAG22/Understanding/accessible-authentication-minimum>  
[Accessed 30 April 2025].
- Agghey , A. Z. et al., 2021. Detection of Username Enumeration Attack on SSH Protocol: Machine Learning Approach. *Symmentry*, 13(11), p. 2192.
- Albeshier, A. S., Alkhaldi, A. & Aljughaiman, A., 2024. Toward secure mobile applications through proper authentication mechanisms. *PLoS ONE*, 5 December.19(12).
- Aldawood, H. & Skinner, G., 2019. Reviewing Cyber Security Social Engineering Training and Awareness Programs—Pitfalls and Ongoing Issues. *Future Internet*, 11(3), p. 73.
- Anon, N.D.. *What is rate limiting and how does it work?*. [Online]  
Available at: <https://www.radware.com/cyberpedia/bot-management/rate-limiting/>  
[Accessed 30 April 2025].
- Aslan, Ö., Aktuğ, S. S., Ozkan, M. & Yilmaz, A. A., 2023. A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. *Electronics*, 12(6), pp. 1-42.
- Auth0, 2025. *Auth0 docs*. [Online]  
Available at: <https://auth0.com/docs/articles>  
[Accessed 30 April 2025].
- Ba, M. H. N., Bennett, J., Gallagher, M. & Bhunia, S., 2021. *A Case Study of Credential Stuffing Attack: Canva Data Breach*. Las Vegas, NV, USA,, IEEE.
- Bhatia, M., 2018. *Your Guide to Qualitative and Quantitative Data Analysis Methods*. [Online]  
Available at: <https://humansofdata.atlan.com/2018/09/qualitative-quantitative-data-analysis-methods/>  
[Accessed 21 August 2023].
- Burrows, M., Abadi, M. & Needham, R. M., 1989. A logic of authentication. *Proceedings of the Royal Society of London*..
- CAPEC, 2018. *CAPEC-112: Brute Force*. [Online]  
Available at: <https://capec.mitre.org/data/definitions/112.html>  
[Accessed 30 March 2025].
- Certus Cybersecurity, 2023. *Rate Limiting 101: Protecting Your Network from Cyber Attacks*. [Online]  
Available at: <https://www.certuscyber.com/insights/rate-limiting-protect-network/>  
[Accessed 30 April 2025].

Cleary, B., 2024. *brute force attack*. [Online]

Available at: <https://us.norton.com/blog/emerging-threats/brute-force-attack>  
[Accessed 30 March 2025].

Contrast Security, 2021. *brute force attack*. [Online]

Available at: <https://www.contrastsecurity.com/glossary/brute-force-attack>  
[Accessed 30 March 2025].

Cremer, F. et al., 2022. Cyber risk and cybersecurity: a systematic review of data availability. *Geneva Pap Risk Insur Issues Pract*, 17 February, 47(3), pp. 698-736.

Creswell, J. W., 2017. *Research design: Qualitative, quantitative, and mixed methods approaches*. 3rd ed. Lincoln: Sage Publications.

Crudu, V. & Team, M. R., 2024. *Best Practices for Django User Authentication*. [Online]

Available at: <https://moldstud.com/articles/p-best-practices-for-django-user-authentication>  
[Accessed 30 April 2025].

CWE Content Team, 2021. *CWE VIEW: Weaknesses in OWASP Top Ten (2021)*. [Online]

Available at: <https://cwe.mitre.org/data/definitions/1344.html>  
[Accessed 15 December 2022].

CyBOK, 2021. *Knowledgebase1\_1*. [Online]

Available at: [https://www.cybok.org/knowledgebase1\\_1/](https://www.cybok.org/knowledgebase1_1/)  
[Accessed 27 March 2022].

Das, A., Bonneau, J., Caesar, M. & Borisov, N., 2014. *The Tangled Web of Password Reuse*. [Online]

Available at:  
[https://www.researchgate.net/publication/269197028\\_The\\_Tangled\\_Web\\_of\\_Password\\_Reuse](https://www.researchgate.net/publication/269197028_The_Tangled_Web_of_Password_Reuse)  
[Accessed 30 April 2025].

Dauzon, S., Bendoraitis, A. & Ravindran, A., 2016. *Django: Web Development with Python*. Mumbai: Packt Publishing Ltd.

De Abiega-L'Eglise, A. F. et al., 2022. A New Fuzzy Vault based Biometric System robust to Brute-Force Attack. *Journal of Computacion y Sistemas*.

Deep, G. et al., 2019. Authentication Protocol for Cloud Databases Using Blockchain Mechanism. *Sensors*, 19(20), p. 4444.

Department for Science, Innovation & Technology, 2024. *Conducting a STRIDE-based threat analysis*. [Online]

Available at: <https://www.gov.uk/government/publications/secure-connected-places-playbook-documents/conducting-a-stride-based-threat-analysis>  
[Accessed 30 April 2025].

Deschoolmeester, D., Landeghem, H. v. & Devos, J., 2013. *Information Systems for Small and Medium-sized Enterprises: State of Art of IS Research in SMEs*. New York: Springer Berlin Heidelberg.

- Devndra, G., 2020. *Comparative study on Python web frameworks: Flask and Django*. [Online]  
Available at: <https://www.theseus.fi/handle/10024/339796>  
[Accessed 30 April 2025].
- Django Software Foundation, 2015. *Documentation*. [Online]  
Available at: <https://docs.djangoproject.com/en/4.1/>  
[Accessed 15 December 2022].
- Django Software Foundation, 2023. *Security in Django*. [Online]  
Available at: <https://docs.djangoproject.com/en/5.2/topics/security/>  
[Accessed 30 April 2025].
- Django Software Foundation, 2023. *Why Django?*. [Online]  
Available at: <https://www.djangoproject.com/start/overview/>  
[Accessed 30 April 2025].
- Django Software Foundation, 2025. *Logging*. [Online]  
Available at: <https://docs.djangoproject.com/en/5.2/topics/logging/>  
[Accessed 30 April 2025].
- Django Software Foundation, 2025. *Sending email*. [Online]  
Available at: <https://docs.djangoproject.com/en/5.2/topics/email/>  
[Accessed 30 April 2025].
- Django Software Foundation, 2025. *Using Sessions*. [Online]  
Available at: <https://docs.djangoproject.com/fr/5.2/topics/http/sessions/>  
[Accessed 01 May 2025].
- Django-Axes, N.D.. *Configuration*. [Online]  
Available at: [https://django-axes.readthedocs.io/en/latest/4\\_configuration.html](https://django-axes.readthedocs.io/en/latest/4_configuration.html)  
[Accessed 30 April 2025].
- Django, N.D.. *Django 1.7.11 documentation*. [Online]  
Available at: <https://django.readthedocs.io/en/1.7.x/index.html>  
[Accessed 12 December 2022].
- Django, N.D.. *Using the Django authentication system*. [Online]  
Available at: <https://docs.djangoproject.com/en/5.2/topics/auth/default/>  
[Accessed 30 April 2025].
- Downey, L. L. & Laskowski, S. J., 1996. *Usability Engineering: IndustryGovernment Collaboration for System Effectiveness and Efficiency*. [Online]  
Available at: <chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.govinfo.gov/content/pkg/GOVPUB-C13-c6d53b6e12963a6af03c8b21bce1a8c1/pdf/GOVPUB-C13-c6d53b6e12963a6af03c8b21bce1a8c1.pdf>  
[Accessed 30 April 2025].
- Europe Commission, 2013. *Ethics for Researchers*. Luxembourg: Europe Union.

Farik, M., Lal, N. A. & Prasad, S., 2016. A Review Of Authentication Methods. *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, 5(11), pp. 246-249.

Farrukh, S., 2013. Tradeoffs between Usability and Security. *International Journal of Engineering and Technology*, 5(4), pp. 434-437.

Fluid attacks: help center, 2024. *Concurrent sessions - Python*. [Online]  
Available at: <https://help.fluidattacks.com/portal/en/kb/articles/criteria-fixes-python-062>  
[Accessed 01 April 2025].

GeoLite2, 2023. *GeoLite Databases and Web Services*. [Online]  
Available at: <https://dev.maxmind.com/geoip/geoip2/geolite2/>  
[Accessed 01 May 2025].

Gollmann, D., 2021. *Authentication, Authorisation & Accountability Knowledge Area Version 1.0.2*. [Online]  
Available at: [chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cybok.org/media/downloads/Authentication\\_Authorisation\\_Accountability\\_v1.0.2.pdf](chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cybok.org/media/downloads/Authentication_Authorisation_Accountability_v1.0.2.pdf)  
[Accessed 4 December 2024].

Golofit, K., 2007. Click Passwords Under Investigation. *European Symposium on Research in Computer Security*, 15(19), pp. 343-358.

Google cloud, 2025. *Identity Platform*. [Online]  
Available at: <https://cloud.google.com/security/products/identity-platform>  
[Accessed 30 April 2025].

Google, 2025. *recaptcha how it works*. [Online]  
Available at: <https://cloud.google.com/security/products/recaptcha#how-it-works>  
[Accessed 30 April 2025].

Grimes, R. A., 2020. Brute-Force Attacks. In: *Hacking Multifactor Authentication*. s.l.:s.n., p. Chapter 14.

Grunwaldt, J.-M., 2019. *A Comparison of Modern Backend Frameworks Protections against Common Web Vulnerabilities*. [Online]  
Available at: <chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.tufts.edu/comp/116/archive/fall2019/jgrunwaldt.pdf>  
[Accessed 30 April 2025].

Hamza, A. & Al-Janabi, R. J. s., 2024. Detecting Brute Force Attacks Using Machine Learning. *BIO Web of Conferences*, 97(3).

Herley, C. & Florencio, D., 2008. Protecting Financial Institutions from Brute-Force Attacks. In: *IFIP – The International Federation for Information Processing*. Boston, MA: Springer, pp. 681-685.

Hevner, A. R., March, S. . T., Ram, S. & Park, J., 2004. Design Science in Information Systems Research. *MIS Quarterly*, 28(1), pp. 75-105.

Idhom, M., Wahanani, H. E. & Fauzi, A., 2020. *Network Security System on Multiple Servers Against Brute Force Attacks*. Surabaya, Indonesia, IEEE, pp. 258-262.

Idris, N., Foozy, C. F. M. & Shamala, P., 2020. A Generic Review of Web Technology: Django and Flask. *International Journal of Advanced Computing Science and Engineering*, 2(1), pp. 34-40.

IndiaFreeNotes, 2023. *Influence of Information Systems in Transforming Businesses*. [Online] Available at: <https://indiafreenotes.com/influence-of-information-systems-in-transforming-businesses/#:~:text=Information%20systems%20have%20transformed%20businesses,increasing%20acces%20to%20new%20markets>. [Accessed 30 July 2023].

Information Commissioner's Office, 2024. *Brute force attacks*. [Online] Available at: <https://ico.org.uk/about-the-ico/research-reports-impact-and-evaluation/research-and-reports/learning-from-the-mistakes-of-others-a-retrospective-review/brute-force-attacks/> [Accessed 3 December 2024].

Jimmy, F., 2024. Cybersecurity Threats and Vulnerabilities in Online Banking Systems. *International Journal of Scientific Research and Management (IJSRM)*, 12(10), pp. 1631-1646.

Khan, R., McLaughlin, K., Laverty, D. & Sezer, S., 2017. *STRIDE-based threat modeling for cyber-physical systems*. Turin, Italy, IEEE.

Kirlappos, I. & Sasse, M. A., 2014. *What Usable Security Really Means: Trusting and Engaging Users*. London, UK, University College London, pp. 69-78.

Lodhi, A., 2010. *Usability Heuristics as an assessment parameter: For performing Usability Testing*. San Juan, USA, IEEE.

Lu, B. et al., 2018. A Measurement Study of Authentication Rate-Limiting Mechanisms of Modern Websites. *ACSAC San Juan, PR, USA*, Volume 00, pp. 3-7.

Lutz, M., 2013. *Learning Python*. 4th ed. s.l.:O'Reilly Media.

Macsinoiu, V. E., 2024. Unveiling User Enumeration Attacks: Methods, Impacts and Mitigation Strategies. *International Journal of Information Security and Cybercrime (IJISC)*, 26(2), pp. 59-64.

Makai, M., N.D.. *Django Extensions, Plug-ins and Related Libraries*. [Online] Available at: <https://www.fullstackpython.com/django-extensions-plugin-ins-related-libraries.html> [Accessed 19 December 2022].

Mayorga, O. E. A. & Yoo, S. G., 2025. One Time Password (OTP) Solution for Two Factor Authentication: A Practical Case Study. *Journal of Computer Science*, 21(5), pp. 1100-1112.

Melé, A., 2020. *Django 3 by example*. 3rd ed. UK: Packt Publishing Ltd.

Mohammed, A. H. Y. & Dziyauddin, R. A., 2023. Current Multi-factor of Authentication: Approaches, Requirements, Attacks and Challenges. *International Journal of Advanced Computer Science and Applications*, 14(1), pp. 166-178.

Moradi, M. & Keyvanpour, M., 2015. CAPTCHA and its Alternatives: A Review. *Security and ommunication Networks*, Volume 8, pp. 2135-2156.

Moyo, S. & Mnkandla, E., 2019. *A Metasynthesis of Solo Software Development Methodologies*. Vanderbijlpark, South Africa, IEEE, pp. 1-8.

Najafabadi, M. M., Calvert, C., Kemp, C. & Khoshgoftaar, T. M., 2015. *Detection of SSH Brute Force Attacks Using Aggregated Netflow Data*. s.l., s.n., pp. 283-288.

National Cyber Security Centre, 2018. *GDPR security outcomes*. [Online]  
Available at: <https://www.ncsc.gov.uk/guidance/gdpr-security-outcomes>  
[Accessed 20 July 2022].

Newman, R., 2009. *Security and Access Control Using Biometric Technologies*. Canada: Cengage Learning.

Nielsen, J. & Molich, R., 1990. *Heuristic Evaluation of User Interfaces*. Denmark, s.n.

Nikiforakis, N. et al., 2013. *Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting*. California, USA, IEEE.

NIST, 2025. *NIST Special Publication 800-63B*. [Online]  
Available at: <https://pages.nist.gov/800-63-3/sp800-63b.html>  
[Accessed 30 April 2025].

Nithya, S. & Rekha, B., 2023. Insights on Data Security Schemes and Authentication Adopted in Safeguarding Social Network. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 14(4).

Norman, K. L. & Kirakowski, J., 2018. *The Wiley Handbook of Human Computer Interaction*. 1st ed. West Sussex, UK: John Wiley & Sons Ltd.

Nurhaida , I. & Bisht , R. K., 2022. *Python for Cyber Security*. [Online]  
Available at: <chrome-extension://efaidnbmninnnibpcajpcglclefindmkaj/https://cs4all.studentscenter.in/assets/Python%20CS/Python%20for%20Cyber%20Security%20Manual.pdf>  
[Accessed 30 April 2025].

Nzeako , G. & Shittu, R. A., 2024. Leveraging AI for enhanced identity and access management in cloud-based systems to advance user authentication and access control. *World Journal of Advanced Research and Reviews*, 24(03), pp. 1661-1674.

Olagbuji, D. O., 2023. *How to Send Email with Verification Link in Django*. [Online]  
Available at: <https://plainenglish.io/blog/how-to-send-email-with-verification-link-in-django>  
[Accessed 30 April 2025].

Olayinka , T. A., Adegede, J. & Jacob, J. G. G., 2024. Balancing Usability and Security in Secure System Design: A Comprehensive Study on Principles, Implementation, and Impact on Usability. *International Journal of Computing Sciences Research*, 8(0), pp. 2995-3009.

OWASP, 2021. *OWASP Top Ten*. [Online]

Available at: <https://owasp.org/www-project-top-ten/>  
[Accessed 30 April 2025].

OWASP, 2025. *Testing for Weak Lock Out Mechanism*. [Online]

Available at: [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\\_Application\\_Security\\_Testing/04-Authentication\\_Testing/03-Testing\\_for\\_Weak\\_Lock\\_Out\\_Mechanism](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/03-Testing_for_Weak_Lock_Out_Mechanism)  
[Accessed 30 April 2025].

OWASP, 2025. *Web Application Security Testing*. [Online]

Available at: [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\\_Application\\_Security\\_Testing/](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/)  
[Accessed 30 April 2025].

OWASP, N.D.. *Testing for Weak Lock Out Mechanism*. [Online]

Available at: [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\\_Application\\_Security\\_Testing/04-Authentication\\_Testing/03-Testing\\_for\\_Weak\\_Lock\\_Out\\_Mechanism](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/03-Testing_for_Weak_Lock_Out_Mechanism)  
[Accessed 30 April 2025].

OWASP, N.D.. *Testing Techniques Explained*. [Online]

Available at: <https://owasp.org/www-project-web-security-testing-guide/latest/2-Introduction/README#Testing-Techniques-Explained>  
[Accessed 30 April 2025].

Owens , J. & Matthews, J., N.D.. *A Study of Passwords and Methods Used in Brute-Force SSH Attacks*. [Online]

Available at: [chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://d1wqtxts1xzle7.cloudfront.net/75610403/leet08-libre.pdf?1638514619=&response-content-disposition=inline%3B+filename%3DA\\_study\\_of\\_passwords\\_and\\_methods\\_used\\_in.pdf&Expires=1747010182&Signature=Yus](chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://d1wqtxts1xzle7.cloudfront.net/75610403/leet08-libre.pdf?1638514619=&response-content-disposition=inline%3B+filename%3DA_study_of_passwords_and_methods_used_in.pdf&Expires=1747010182&Signature=Yus)  
[Accessed 30 April 2025].

Özeren , S., 2024. *Breach and Attack Simulation vs. Security Validation*. [Online]

Available at: <https://www.picussecurity.com/resource/blog/breach-and-attack-simulation-vs-security-validation>  
[Accessed 08 December 2024].

Palmieri, M., 2013. *System Testing in a Simulated Environment*. [Online]

Available at: <chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.diva-portal.org/smash/get/diva2:613817/FULLTEXT01.pdf>  
[Accessed 07 December 2024].

Papathanasaki, M., Maglaras, L. & Ayres, N., 2022. Modern Authentication Methods: A Comprehensive Survey. *AI, Computer Science and Robotics Technology*, Volume 0, pp. 1-24.

- Park, K., Lee, J., Ashok, K. D. & Park, Y., 2023. BPPS:Blockchain-Enabled Privacy-Preserving Scheme for Demand-Response Management in Smart Grid Environments. *Computer Science, Engineering, Environmental Science*, 20(2), pp. 1719-1729.
- Parmar, V., Sanghvi, H. A., Patel, R. H. & Pandya, A. S., 2022. *A Comprehensive Study on Passwordless Authentication*. Erode, IEEE.
- Phan, K., 2008. *Implementing Resiliency of Adaptive Multi-Factor Authentication Systems*. [Online] Available at: [chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1095&context=msia\\_etds](chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1095&context=msia_etds) [Accessed 30 March 2025].
- Purba, K. R. & Ramli, R., 2022. *A Rapid Solo Software Development (RSSD) Methodology based on Agile*. [Online] Available at: [https://www.researchgate.net/publication/362980213\\_A\\_Rapid\\_Solo\\_Software\\_Development\\_RSSD\\_Methodology\\_based\\_on\\_Agile](https://www.researchgate.net/publication/362980213_A_Rapid_Solo_Software_Development_RSSD_Methodology_based_on_Agile) [Accessed 30 April 2025].
- PyLessons, 2022. *Django website introduction*. [Online] Available at: <https://pylessons.com/django-introduction> [Accessed 30 April 2025].
- PyLessons, 2022. *Google reCAPTCHA in Django*. [Online] Available at: [https://pylessons.com/django-recaptcha#google\\_vignette](https://pylessons.com/django-recaptcha#google_vignette) [Accessed 30 April 2025].
- Rashidi, B. & Garg, V., 2021. Open sesame: Lessons in password-based user authentication. *Cyber Security: A Peer-Reviewed Journal*, 4(4), p. 317–329.
- Raza, M., Iqbal, M., Sharif, M. & Haider, W., 2012. A Survey of Password Attacks and Comparative Analysis on Methods for Secure Authentication. *World Applied Sciences Journal*, 19(4), pp. 439-444.
- Renaud, K., Scott-Brown, K. C. & Szymkow, A., 2018. *Designing authentication with seniors in mind*. Barcelona, Spain, s.n.
- Rivera, J. J. D., Muhammad, A. & Song, W.-C., 2024. Securing Digital Identity in the Zero Trust Architecture: A Blockchain Approach to Privacy-Focused Multi-Factor Authentication. *IEEE Open Journal of the Communications Society*.
- Sanjari, M. et al., 2014. Ethical challenges of researchers in qualitative studies: the necessity to develop a specific guideline. *Journal of medical ethics and history of medicine*, 7(14).
- Sarveshwaran, V., Chen, J. I.-z. & Pelusi, D., 2023. *Artificial Intelligence and Cyber Security in Industry 4.0*. s.l.:Springer.
- Shrivastava, G. et al., 2024. *Emerging Threats and Countermeasures in Cybersecurity*. s.l.:John Wiley & Sons.

Socol, J., N.D.. *Django Ratelimit*. [Online]

Available at: <https://django-ratelimit.readthedocs.io/en/stable/index.html>

[Accessed 30 April 2025].

Sutherland, J., 2014. *Scrum: The Art of Doing Twice the Work in Half the Time*. 1st ed. New York: Crown Business.

Tamilkodi, R. et al., 2024. *Identification and Prevention of Brute Force Attacks*. Singapore, Springer.

Tariq, N., Khan, F. A., Moqurrah, S. A. & Srivastava, G., 2023. CAPTCHA Types and Breaking Techniques: Design Issues, Challenges, and Future Research Directions. *ACM Comput. Surv.*, 00(0).

Turner, S. & Housley, R., 2008. *implementing Email Security and Tokens: Current Standards, Tools and Practices*. Indiana: Wiley Publishing Inc.

Uma, M. & Padmavathi, G., 2013. A Survey on Various Cyber Attacks and Their Classification. *International Journal of Network Security*, 15(5), pp. 390-396.

Velásquez, I., Caro, A. & Rodríguez, A., 2018. Authentication schemes and methods: A systematic literature review. *Information and Software Technology*, Volume 94, pp. 30-37.

Velásquez, I., Caro, A. & Rodríguez, A., 2019. Multifactor Authentication Methods: A Framework for Their Comparison and Selection. In: *Computer and Network Security*. India: s.n.

Velgekar, S., Khandve, H. & Gundla, R., 2021. Survey of Artificial Intelligence Applications In Cybersecurity. *International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET)*, 10(5), pp. 4289-4296.

Venkatesh, V., Morris, M. G., Davis, G. B. & Davis, F. D., 2003. User Acceptance of Information Technology: Toward a Unified View. *MIS Quarterly*, 27(3), pp. 425-478.

Vugdelija, N. et al., N.D.. *REVIEW OF BRUTE-FORCE ATTACK AND PROTECTION TECHNIQUES*. [Online]

Available at: [chrome-](chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://proceedings.ictinnovations.org/attachment/paper/554/review-of-brute-force-attack-and-protection-techniques.pdf)

[extension://efaidnbmnnnibpcajpcglclefindmkaj/https://proceedings.ictinnovations.org/attachment/paper/554/review-of-brute-force-attack-and-protection-techniques.pdf](chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://proceedings.ictinnovations.org/attachment/paper/554/review-of-brute-force-attack-and-protection-techniques.pdf)

[Accessed 1 April 2025].

Wang, D. et al., 2023. Security in wireless body area networks via anonymous authentication: Comprehensive literature review, scheme classification, and future challenges. *Ad Hoc Networks*, Volume 153, p. 10332.

Wang, X., Yan, Z., Zhang, R. & Zhang, P., 2021. Attacks and defenses in user authentication systems: A survey. *Journal of Network and Computer Applications*, 15 August. Volume 188.

Wang, Z. & Sun, W., 2020. Review of Web Authentication. *Journal of Physics: Conference Series*, Volume 1646, pp. 14-15.

Wee, A. K., Chekole, E. G. & Zhou, J., 2024. Excavating Vulnerabilities Lurking in Multi-Factor Authentication Protocols: A Systematic Security Analysis.

Weingart, S. H., 2002. *Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses*. Berlin, Springer.

Yonkeu, S., 2020. *Location and Device Fingerprinting*. [Online]

Available at: <https://dev.to/yokwejuste/location-and-device-fingerprinting-1caa>

[Accessed 30 April 2025].

Yonkeu, S., 2020. *Role-Based Access Control in Django*. [Online]

Available at: <https://dev.to/yokwejuste/role-based-access-control-in-django-4j1d>

[Accessed 30 April 2025].

Yusop, M. I. M., Kamarudin, N. H., Suhaimi, N. H. S. & Hasan, M. K., 2025. Advancing Passwordless Authentication: A Systematic Review of Methods, Challenges, and Future Directions for Secure User Identity. *IEEE Access*, Volume 13, pp. 13919 - 13943.

Zeng, J. et al., 2024. *A Survey of Third-Party Library Security Research in Application Software*. [Online]

Available at: <https://arxiv.org/html/2404.17955v1>

[Accessed 30 April 2025].

Zhang, C. J., Gill, A. Q., Liu, B. & Anwar, M., 2025. *AI-based Identity Fraud Detection: A Systematic Review*. [Online]

Available at: <https://arxiv.org/html/2501.09239v1>

[Accessed 01 May 2025].

Zhang, J. et al., 2018. T2FA: Transparent Two-Factor Authentication. *IEEE Access*, January, Volume 6, p. 32677–32686.

Zhang, X. et al., 2022. Data breach: analysis, countermeasures and challenges. *International Journal of Information and Computer Security*, January, 19(3/4), pp. 402-442.

## Appendices

### Appendix A: Setup Guide/Readme file

Below are the step-by-step setup instructions, as detailed in the README file, for running the LogIn system. The LogIn System is available on my GitHub repository:

**<https://github.com/MUTEGIbeatrice/thesisdjango.git>**

#### **Steps:**

a). First run *pip install -r requirements.txt* so as to install all the required dependencies that was used throughout the app development time.

b). Then use this command to run the DjangoApp in:

- http: `python manage.py runserver`
- https: `python manage.py runserver_plus --cert-file cert.pem --key-file key.pem`

c). Then create an account on the signup page.

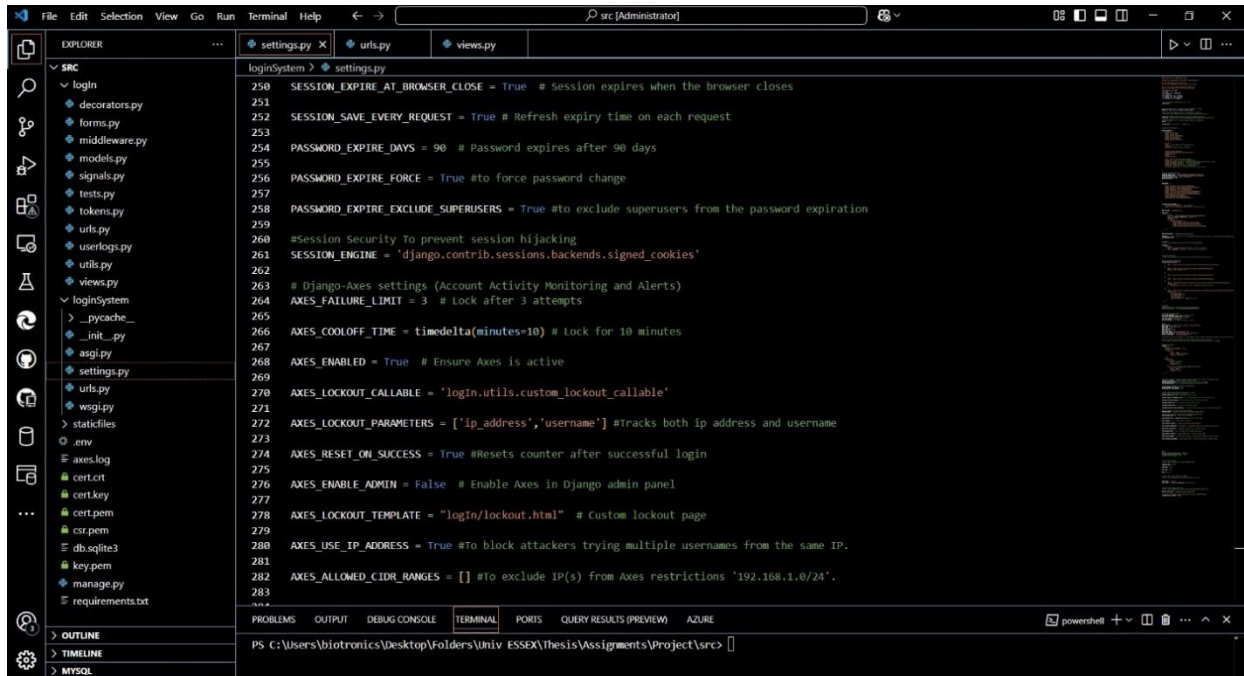
### **Credentials of users in the Django logIn System**

- Superuser or Admin:  
Username: UserAdmin  
Password: User@123
- First user:  
Username: UserOne  
Password: User1pass@1

## Appendix B: Source Code Snippets of Core Components

Below are key highlights from the core components of the logIn System. The full source code is available at: <https://github.com/MUTEGlbeatrice/thesisdjango.git>

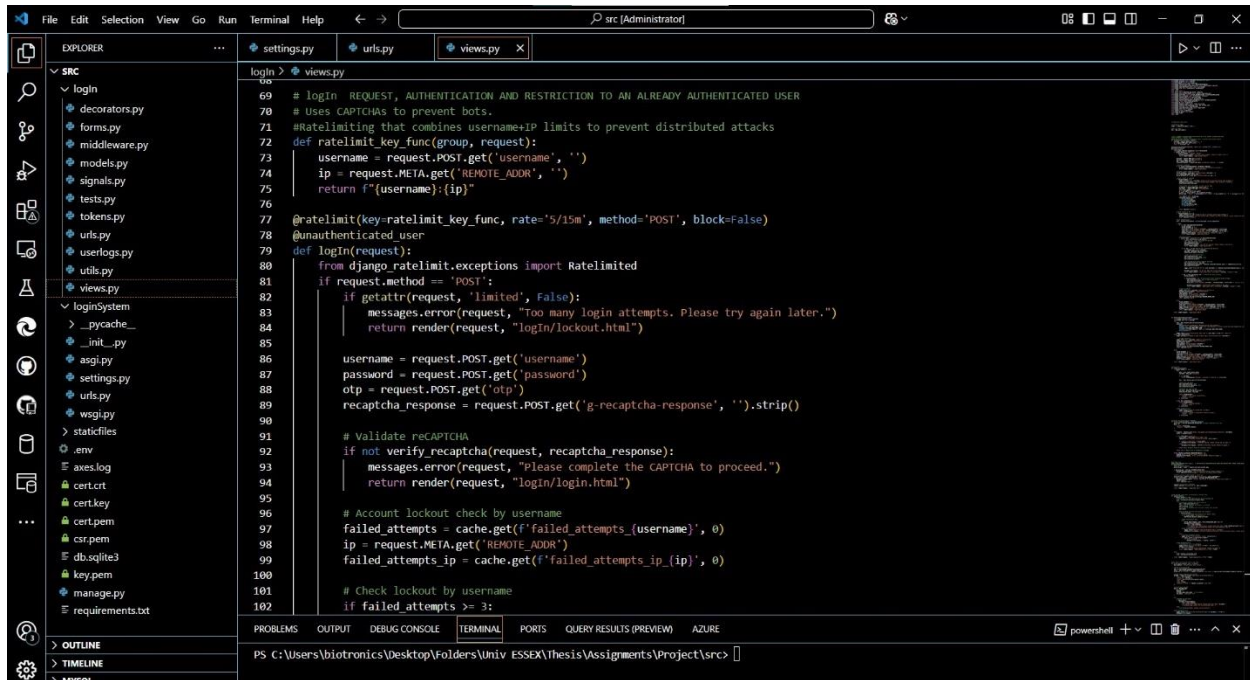
### Implementation of Django-Axes on Settings.py



```
loginSystem > settings.py
250 SESSION_EXPIRE_AT_BROWSER_CLOSE = True # Session expires when the browser closes
251
252 SESSION_SAVE_EVERY_REQUEST = True # Refresh expiry time on each request
253
254 PASSWORD_EXPIRE_DAYS = 90 # Password expires after 90 days
255
256 PASSWORD_EXPIRE_FORCE = True #to force password change
257
258 PASSWORD_EXPIRE_EXCLUDE_SUPERUSERS = True #to exclude superusers from the password expiration
259
260 #Session Security to prevent session hijacking
261 SESSION_ENGINE = 'django.contrib.sessions.backends.signed_cookies'
262
263 # Django-Axes settings (Account Activity Monitoring and Alerts)
264 AXES_FAILURE_LIMIT = 3 # Lock after 3 attempts
265
266 AXES_COOLOFF_TIME = timedelta(minutes=10) # Lock for 10 minutes
267
268 AXES_ENABLED = True # Ensure Axes is active
269
270 AXES_LOCKOUT_CALLABLE = 'login.utils.custom_lockout_callable'
271
272 AXES_LOCKOUT_PARAMETERS = ['ip_address', 'username'] #Tracks both ip address and username
273
274 AXES_RESET_ON_SUCCESS = True #Resets counter after successful login
275
276 AXES_ENABLE_ADMIN = False # Enable Axes in Django admin panel
277
278 AXES_LOCKOUT_TEMPLATE = "login/lockout.html" # Custom lockout page
279
280 AXES_USE_IP_ADDRESS = True #to block attackers trying multiple usernames from the same IP.
281
282 AXES_ALLOWED_CIDR_RANGES = [] #to exclude IP(s) from Axes restrictions '192.168.1.0/24'.
283
```

Figure 4: Django-Axes as on settings.py

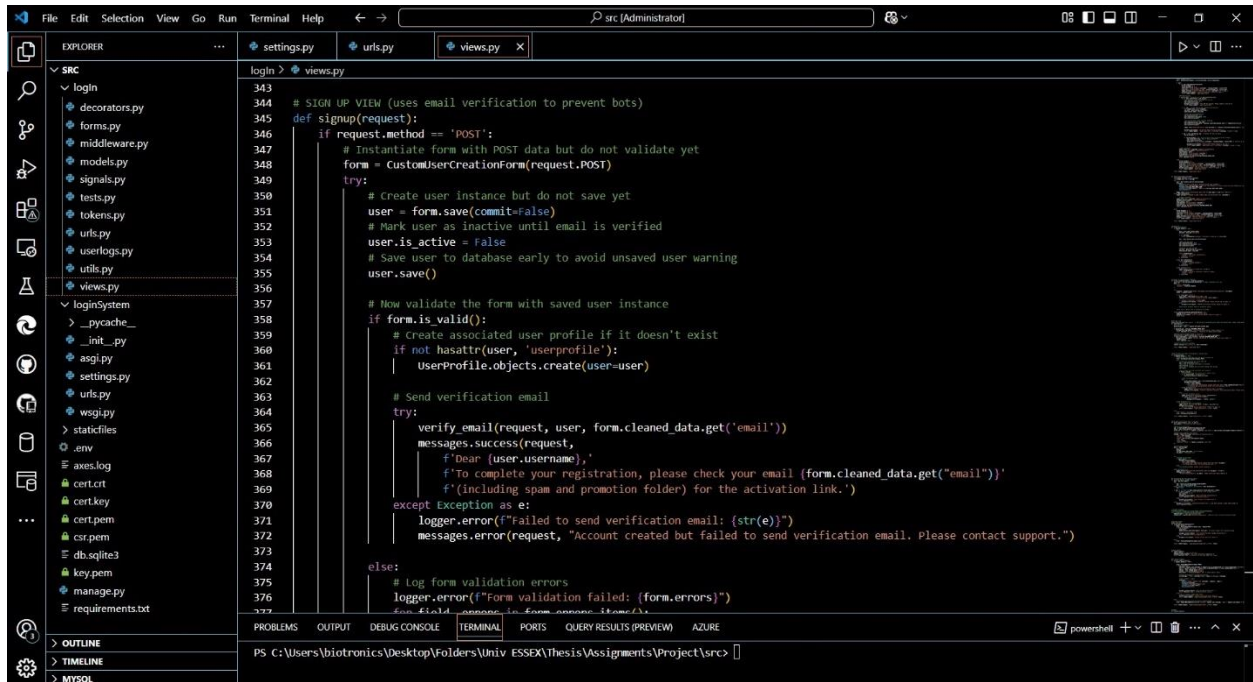
## Login view capturing rate limit



```
69 # login REQUEST, AUTHENTICATION AND RESTRICTION TO AN ALREADY AUTHENTICATED USER
70 # Uses CAPTCHAs to prevent bots.
71 #RateLimiting that combines username+IP limits to prevent distributed attacks
72 def ratelimit_key_func(group, request):
73     username = request.POST.get('username', '')
74     ip = request.META.get('REMOTE_ADDR', '')
75     return f'{username}:{ip}'
76
77 @ratelimit(key=ratelimit_key_func, rate='5/15m', method='POST', block=False)
78 @unauthenticated_user
79 def login(request):
80     from django_ratelimit.exceptions import Ratelimited
81     if request.method == 'POST':
82         if getattr(request, 'limited', False):
83             messages.error(request, "Too many login attempts. Please try again later.")
84             return render(request, "login/lockout.html")
85
86         username = request.POST.get('username')
87         password = request.POST.get('password')
88         otp = request.POST.get('otp')
89         recaptcha_response = request.POST.get('g-recaptcha-response', '').strip()
90
91         # Validate reCAPTCHA
92         if not verify_recaptcha(request, recaptcha_response):
93             messages.error(request, "Please complete the CAPTCHA to proceed.")
94             return render(request, "login/login.html")
95
96         # Account lockout check by username
97         failed_attempts = cache.get(f'failed_attempts_{username}', 0)
98         ip = request.META.get('REMOTE_ADDR')
99         failed_attempts_ip = cache.get(f'failed_attempts_ip_{ip}', 0)
100
101         # Check lockout by username
102         if failed_attempts >= 3:
```

Figure 5: Rate limit on login view

## Signup view

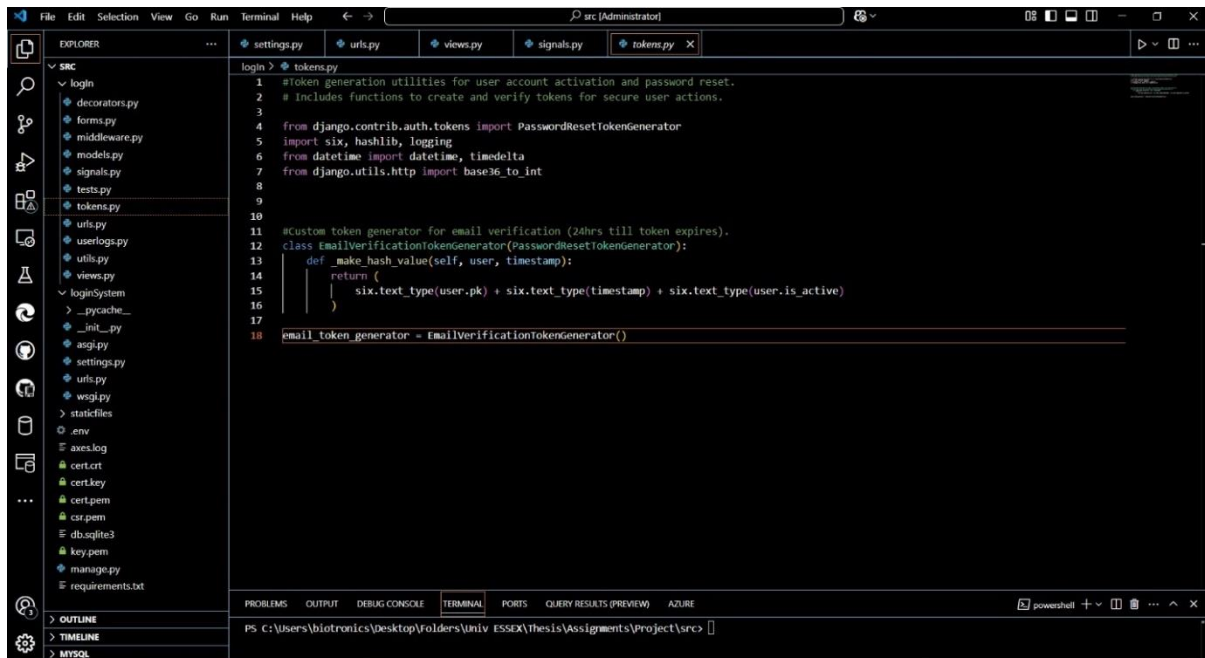


The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'src' folder containing 'login' and 'loginSystem' subfolders. The 'login' folder contains several Python files, including 'views.py'. The 'loginSystem' folder contains 'urls.py' and 'views.py'. The 'views.py' file in the 'login' folder is selected, and its content is displayed in the code editor. The code implements a 'signup' view that handles POST requests, creates a user instance, saves it to the database, and sends a verification email. It also includes error handling for validation failures and email sending errors.

```
343
344 # SIGN UP VIEW (uses email verification to prevent bots)
345 def signup(request):
346     if request.method == 'POST':
347         # Instantiate form with POST data but do not validate yet
348         form = CustomUserCreationForm(request.POST)
349         try:
350             # Create user instance but do not save yet
351             user = form.save(commit=False)
352             # Mark user as inactive until email is verified
353             user.is_active = False
354             # Save user to database early to avoid unsaved user warning
355             user.save()
356
357             # Now validate the form with saved user instance
358             if form.is_valid():
359                 # create associated user profile if it doesn't exist
360                 if not hasattr(user, 'userprofile'):
361                     UserProfile.objects.create(user=user)
362
363                 # Send verification email
364                 try:
365                     verify_email(request, user, form.cleaned_data.get('email'))
366                     messages.success(request,
367                                   f'Dear {user.username}, '
368                                   f'To complete your registration, please check your email {form.cleaned_data.get("email")}'
369                                   f'(including spam and promotion folder) for the activation link.')
370                 except Exception as e:
371                     logger.error(f'Failed to send verification email: {str(e)}')
372                     messages.error(request, "Account created but failed to send verification email. Please contact support.")
373             else:
374                 # Log form validation errors
375                 logger.error(f'Form validation failed: {form.errors}')
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

Figure 6: Signup view

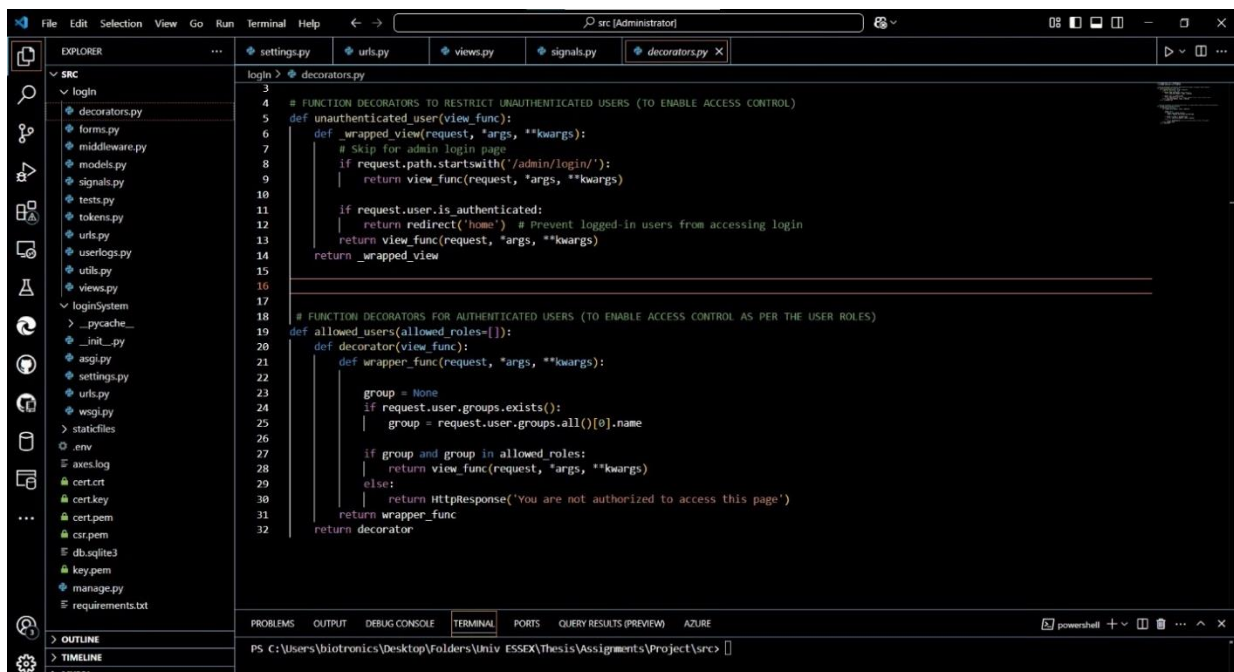
## Email Token Verification as on tokens.py



```
1 #Token generation utilities for user account activation and password reset.
2 # Includes functions to create and verify tokens for secure user actions.
3
4 from django.contrib.auth.tokens import PasswordResetTokenGenerator
5 import six, hashlib, logging
6 from datetime import datetime, timedelta
7 from django.utils.http import base36_to_int
8
9
10
11 #Custom token generator for email verification (24hrs till token expires).
12 class EmailVerificationTokenGenerator(PasswordResetTokenGenerator):
13     def _make_hash_value(self, user, timestamp):
14         return (
15             six.text_type(user.pk) + six.text_type(timestamp) + six.text_type(user.is_active)
16         )
17
18 email_token_generator = EmailVerificationTokenGenerator()
```

Figure 7: Email token on tokens.py

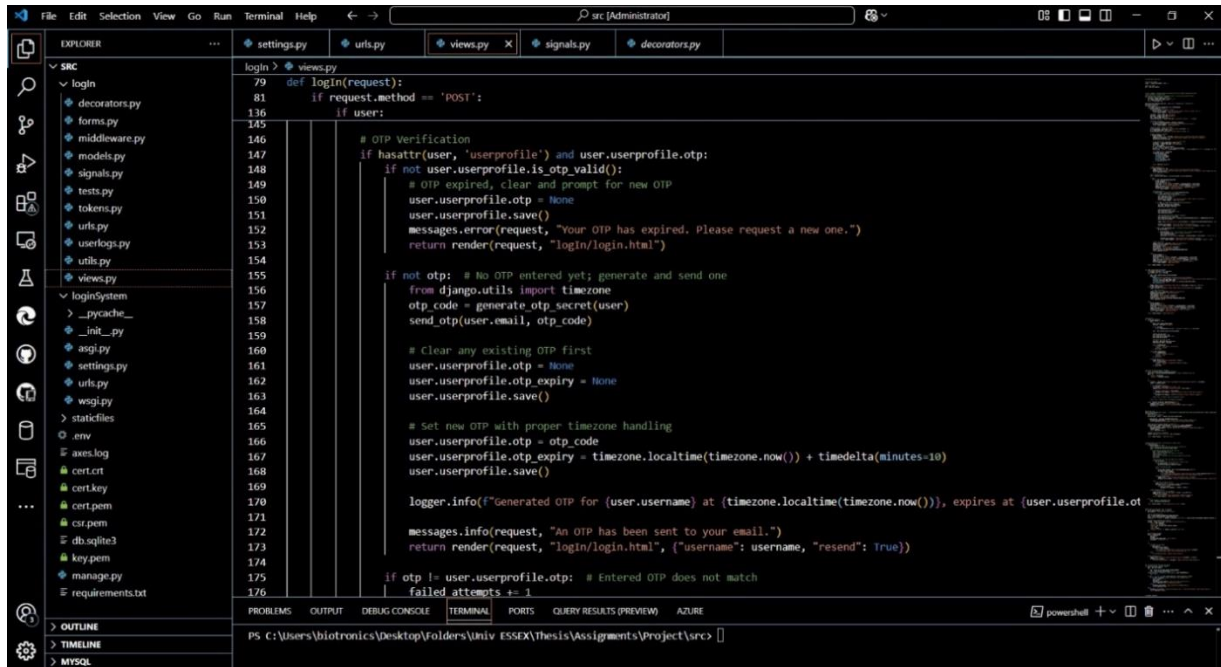
## User Access Control on Decorators.py



```
3
4 # FUNCTION DECORATORS TO RESTRICT UNAUTHENTICATED USERS (TO ENABLE ACCESS CONTROL)
5 def unauthenticated_user(view_func):
6     def _wrapped_view(request, *args, **kwargs):
7         # Skip for admin login page
8         if request.path.startswith('/admin/login/'):
9             return view_func(request, *args, **kwargs)
10
11         if request.user.is_authenticated:
12             return redirect('home') # prevent logged-in users from accessing login
13         return view_func(request, *args, **kwargs)
14     return _wrapped_view
15
16
17
18 # FUNCTION DECORATORS FOR AUTHENTICATED USERS (TO ENABLE ACCESS CONTROL AS PER THE USER ROLES)
19 def allowed_users(allowed_roles=[]):
20     def decorator(view_func):
21         def wrapper_func(request, *args, **kwargs):
22             group = None
23             if request.user.groups.exists():
24                 group = request.user.groups.all()[0].name
25
26             if group and group in allowed_roles:
27                 return view_func(request, *args, **kwargs)
28             else:
29                 return HttpResponse('You are not authorized to access this page')
30             return wrapper_func
31         return decorator
32     return decorator
```

Figure 8: User Access Control on decorators.py

## OTP-based Two-Factor Authentication (2FA) Logic on Login view

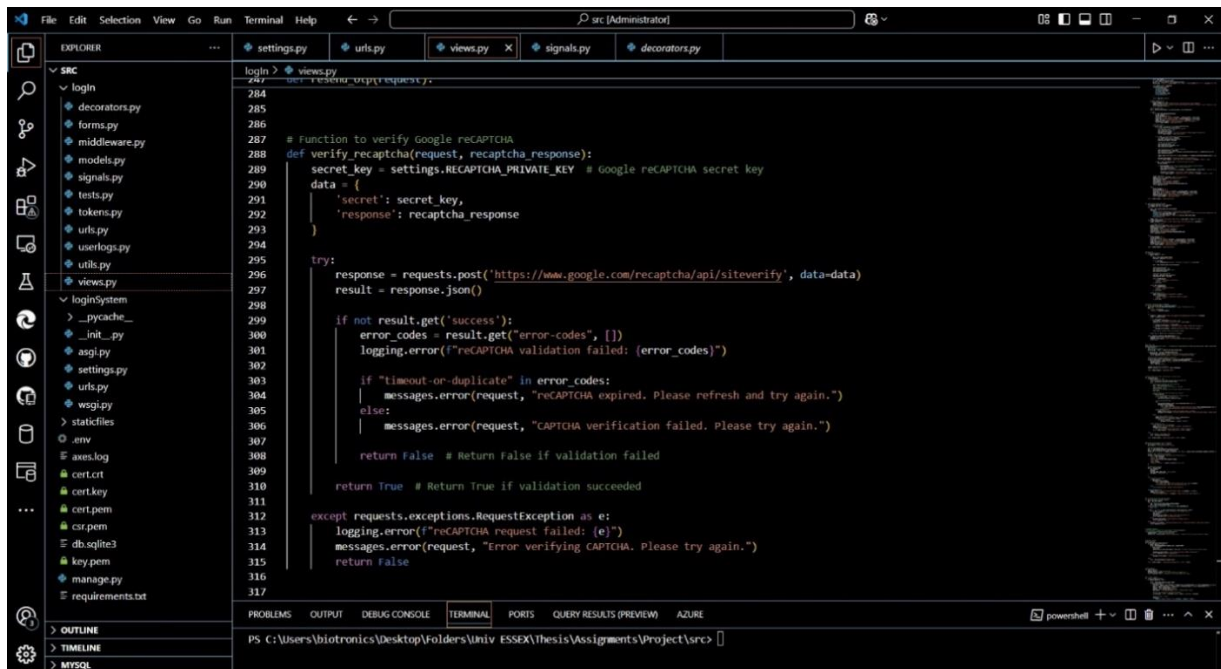


The screenshot shows a Visual Studio Code editor with a Python project. The Explorer pane on the left shows a file tree with folders like 'login' and 'views.py'. The main editor window displays the code for the 'login' view. The code implements OTP verification logic. It checks if the user has a valid OTP. If not, it generates a new OTP and sends it to the user's email. If the OTP is entered but does not match, it increments the failed attempts counter.

```
login > views.py
79 def login(request):
80     if request.method == 'POST':
81         if user:
82             # OTP Verification
83             if hasattr(user, 'userprofile') and user.userprofile.otp:
84                 if not user.userprofile.is_otp_valid():
85                     # OTP expired, clear and prompt for new OTP
86                     user.userprofile.otp = None
87                     user.userprofile.save()
88                     messages.error(request, "Your OTP has expired. Please request a new one.")
89                     return render(request, "login/login.html")
90             if not otp: # No OTP entered yet; generate and send one
91                 from django.utils import timezone
92                 otp_code = generate_otp_secret(user)
93                 send_otp(user.email, otp_code)
94             # Clear any existing OTP first
95             user.userprofile.otp = None
96             user.userprofile.otp_expiry = None
97             user.userprofile.save()
98             # Set new OTP with proper timezone handling
99             user.userprofile.otp = otp_code
100             user.userprofile.otp_expiry = timezone.localtime(timezone.now()) + timedelta(minutes=10)
101             user.userprofile.save()
102             logger.info(f"Generated OTP for {user.username} at {timezone.localtime(timezone.now())}, expires at {user.userprofile.otp_expiry}")
103             messages.info(request, "An OTP has been sent to your email.")
104             return render(request, "login/login.html", {"username": username, "resend": True})
105         if otp != user.userprofile.otp: # Entered OTP does not match
106             failed_attempts += 1
```

Figure 9: OTP verification on Login view

## CAPTCHA Verification Logic on Login view

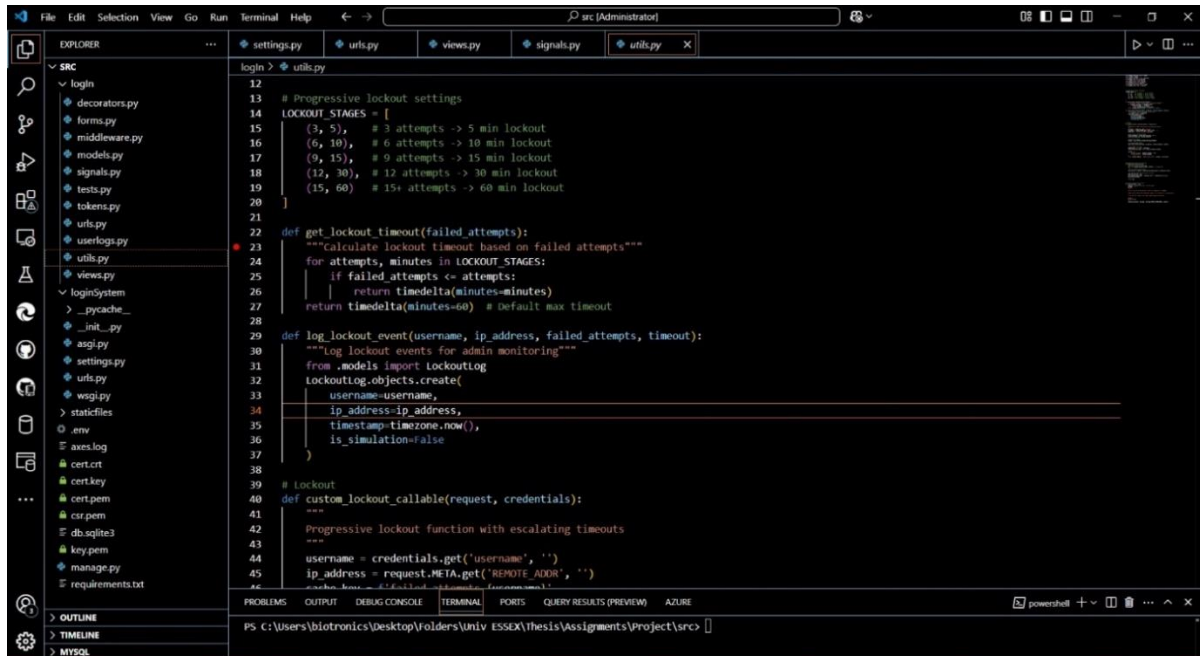


The screenshot shows a Visual Studio Code editor with a Python project. The Explorer pane on the left shows a file tree with folders like 'login' and 'views.py'. The main editor window displays the code for the 'login' view. The code implements CAPTCHA verification logic. It uses the 'requests' library to verify the CAPTCHA with Google. If the verification fails, it returns an error message. If it succeeds, it returns True.

```
login > views.py
284 def login(request):
285     if request.method == 'POST':
286         if user:
287             # function to verify Google reCAPTCHA
288             def verify_recaptcha(request, recaptcha_response):
289                 secret_key = settings.RECAPTCHA_PRIVATE_KEY # Google reCAPTCHA secret key
290                 data = {
291                     'secret': secret_key,
292                     'response': recaptcha_response
293                 }
294             try:
295                 response = requests.post('https://www.google.com/recaptcha/api/siteverify', data=data)
296                 result = response.json()
297                 if not result.get('success'):
298                     error_codes = result.get("error-codes", [])
299                     logging.error(f"reCAPTCHA validation failed: {error_codes}")
300                     if "timeout-or-duplicate" in error_codes:
301                         messages.error(request, "reCAPTCHA expired. Please refresh and try again.")
302                     else:
303                         messages.error(request, "CAPTCHA verification failed. Please try again.")
304                 return False # Return False if validation failed
305             except requests.exceptions.RequestException as e:
306                 logging.error(f"reCAPTCHA request failed: {e}")
307                 messages.error(request, "Error verifying CAPTCHA. Please try again.")
308                 return False
309             return True # Return True if validation succeeded
310         except requests.exceptions.RequestException as e:
311             logging.error(f"reCAPTCHA request failed: {e}")
312             messages.error(request, "Error verifying CAPTCHA. Please try again.")
313             return False
314         return render(request, "login/login.html", {"username": username, "resend": True})
315     failed_attempts += 1
```

Figure 10: CAPTCHA Verification on Login view

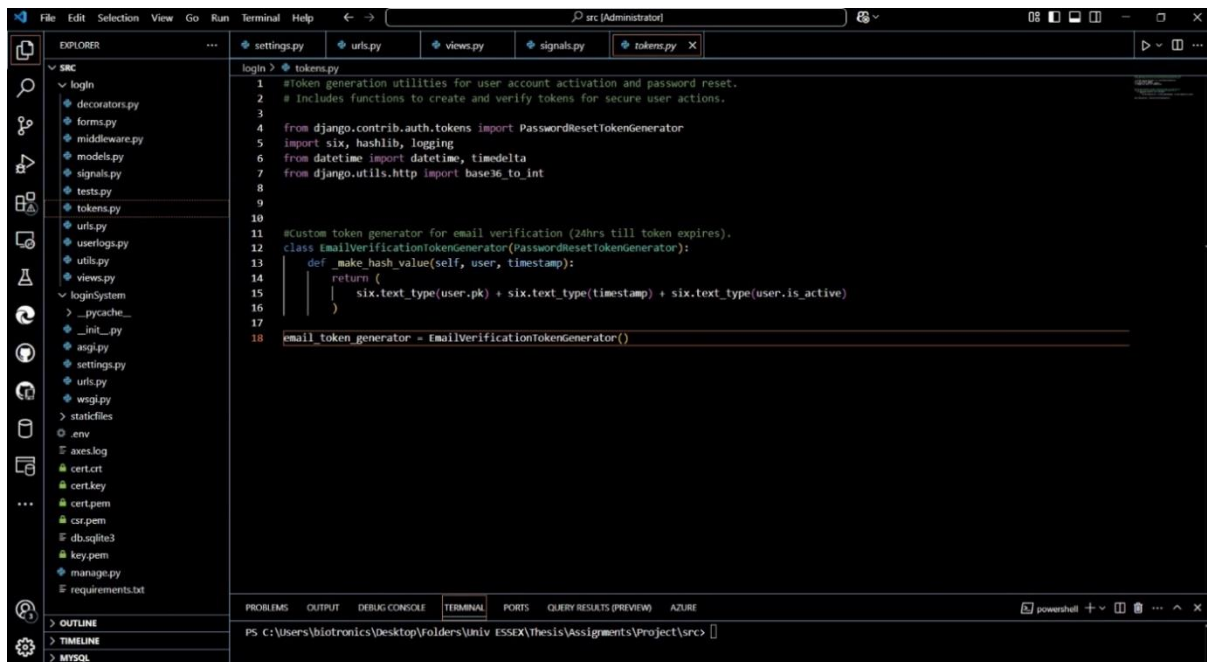
## Progressive Account Lockout Implementation Code as on utils.py



```
12 # Progressive lockout settings
13 LOCKOUT_STAGES = [
14     (3, 5), # 3 attempts -> 5 min lockout
15     (6, 10), # 6 attempts -> 10 min lockout
16     (9, 15), # 9 attempts -> 15 min lockout
17     (12, 30), # 12 attempts -> 30 min lockout
18     (15, 60) # 15+ attempts -> 60 min lockout
19 ]
20
21
22 def get_lockout_timeout(failed_attempts):
23     """Calculate lockout timeout based on failed attempts"""
24     for attempts, minutes in LOCKOUT_STAGES:
25         if failed_attempts <= attempts:
26             return timedelta(minutes=minutes)
27     return timedelta(minutes=60) # Default max timeout
28
29
30 def log_lockout_event(username, ip_address, failed_attempts, timeout):
31     """Log lockout events for admin monitoring"""
32     from .models import LockoutLog
33     LockoutLog.objects.create(
34         username=username,
35         ip_address=ip_address,
36         timestamp=timezone.now(),
37         is_simulation=False
38     )
39
40 # Lockout
41 def custom_lockout_callable(request, credentials):
42     """
43     Progressive lockout function with escalating timeouts
44     """
45     username = credentials.get('username', '')
46     ip_address = request.META.get('REMOTE_ADDR', '')
```

Figure 11: Progressive Lockout on utils.py

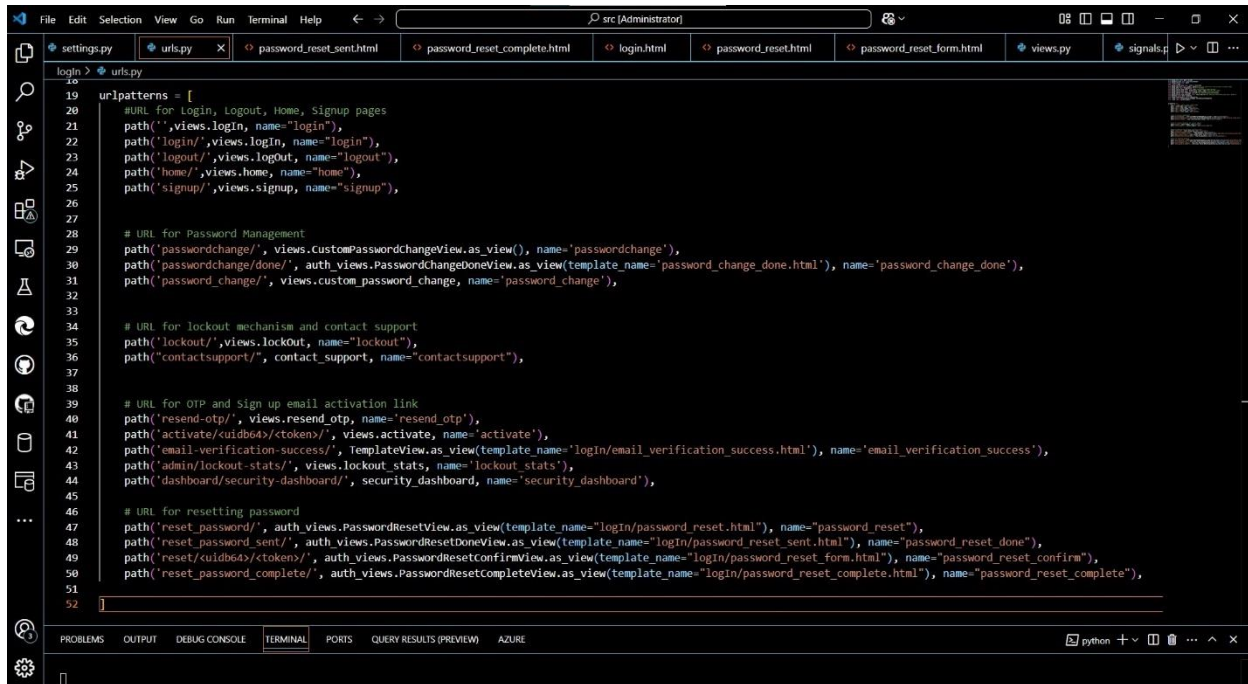
## Email Token Generator Code



```
1 #Token generation utilities for user account activation and password reset.
2 # Includes functions to create and verify tokens for secure user actions.
3
4 from django.contrib.auth.tokens import PasswordResetTokenGenerator
5 import six, hashlib, logging
6 from datetime import datetime, timedelta
7 from django.utils.http import base64_to_int
8
9
10
11 #Custom token generator for email verification (24hrs till token expires).
12 class EmailVerificationTokenGenerator(PasswordResetTokenGenerator):
13     def _make_hash_value(self, user, timestamp):
14         return (
15             six.text_type(user.pk) + six.text_type(timestamp) + six.text_type(user.is_active)
16         )
17
18 email_token_generator = EmailVerificationTokenGenerator()
```

Figure 12: Email Token Generator on tokens.py

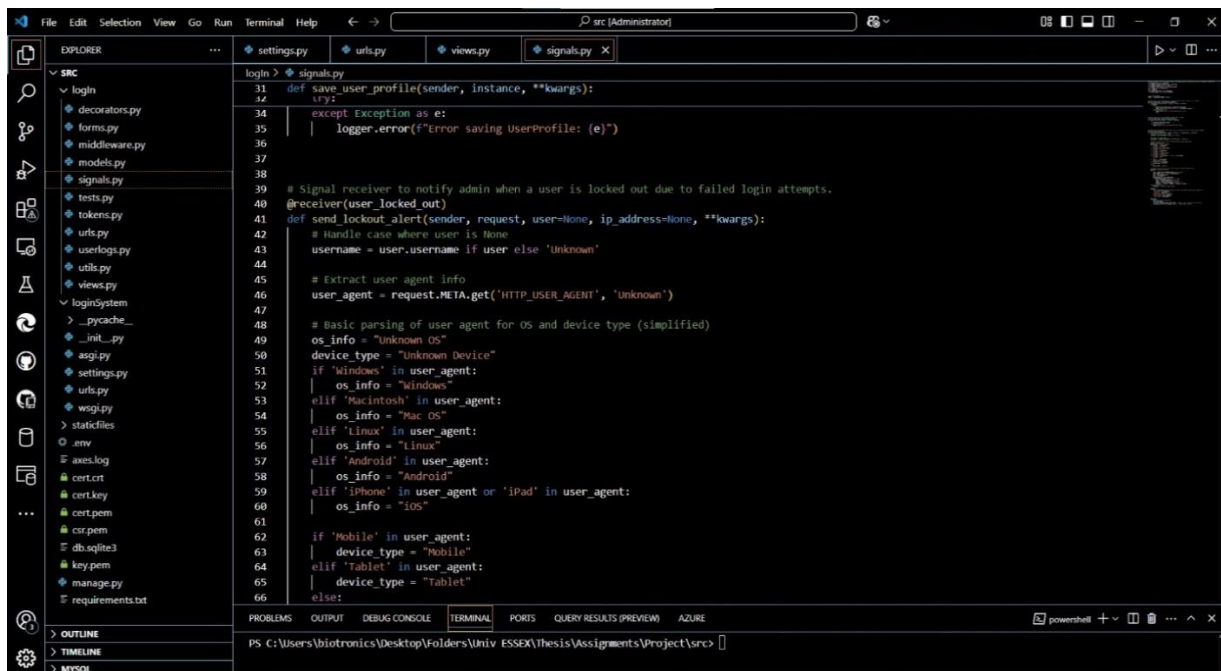
## Urls.py



```
18
19 urlpatterns = [
20     # URL for Login, Logout, Home, Signup pages
21     path('', views.login, name='login'),
22     path('login/', views.login, name='login'),
23     path('logout/', views.logout, name='logout'),
24     path('home/', views.home, name='home'),
25     path('signup/', views.signup, name='signup'),
26
27
28     # URL for Password Management
29     path('passwordchange/', views.CustomPasswordChangeView.as_view(), name='passwordchange'),
30     path('passwordchange/done/', auth_views.PasswordChangeDoneView.as_view(template_name='password_change_done.html'), name='password_change_done'),
31     path('password_change/', views.custom_password_change, name='password_change'),
32
33
34     # URL for logout mechanism and contact support
35     path('logout/', views.logout, name='logout'),
36     path('contactsupport/', contact_support, name='contactsupport'),
37
38
39     # URL for OTP and Sign up email activation link
40     path('resend-otp/', views.resend_otp, name='resend_otp'),
41     path('activate/cuid64/token/', views.activate, name='activate'),
42     path('email_verification_success/', TemplateView.as_view(template_name='login/email_verification_success.html'), name='email_verification_success'),
43     path('admin/lockout/stats/', views.lockout_stats, name='lockout_stats'),
44     path('dashboard/security-dashboard/', security_dashboard, name='security_dashboard'),
45
46
47     # URL for resetting password
48     path('reset_password/', auth_views.PasswordResetView.as_view(template_name='login/password_reset.html'), name='password_reset'),
49     path('reset_password_sent/', auth_views.PasswordResetDoneView.as_view(template_name='login/password_reset_sent.html'), name='password_reset_done'),
50     path('reset/cuid64/token/', auth_views.PasswordResetConfirmView.as_view(template_name='login/password_reset_form.html'), name='password_reset_confirm'),
51     path('reset_password_complete/', auth_views.PasswordResetCompleteView.as_view(template_name='login/password_reset_complete.html'), name='password_reset_complete'),
52 ]
```

Figure 13: urls.py

## Device Fingerprinting (User agents) implementation on Signals.py



```
31 def save_user_profile(sender, instance, **kwargs):
32     url =
33
34     except Exception as e:
35         logger.error(f"Error saving UserProfile: (e)")
36
37
38
39 # Signal receiver to notify admin when a user is locked out due to failed login attempts.
40 @receiver(user_locked_out)
41 def send_lockout_alert(sender, request, user=None, ip_address=None, **kwargs):
42     # Handle case where user is None
43     username = user.username if user else 'Unknown'
44
45     # Extract user agent info
46     user_agent = request.META.get('HTTP_USER_AGENT', 'Unknown')
47
48     # Basic parsing of user agent for OS and device type (simplified)
49     os_info = "Unknown OS"
50     device_type = "Unknown Device"
51     if 'Windows' in user_agent:
52         os_info = "Windows"
53     elif 'Macintosh' in user_agent:
54         os_info = "Mac OS"
55     elif 'Linux' in user_agent:
56         os_info = "Linux"
57     elif 'Android' in user_agent:
58         os_info = "Android"
59     elif 'iPhone' in user_agent or 'iPad' in user_agent:
60         os_info = "IOS"
61
62     if 'Mobile' in user_agent:
63         device_type = "Mobile"
64     elif 'Tablet' in user_agent:
65         device_type = "Tablet"
66     else:
```

Figure 14: User agents on signals.py

## Google reCAPTCHA v2 settings

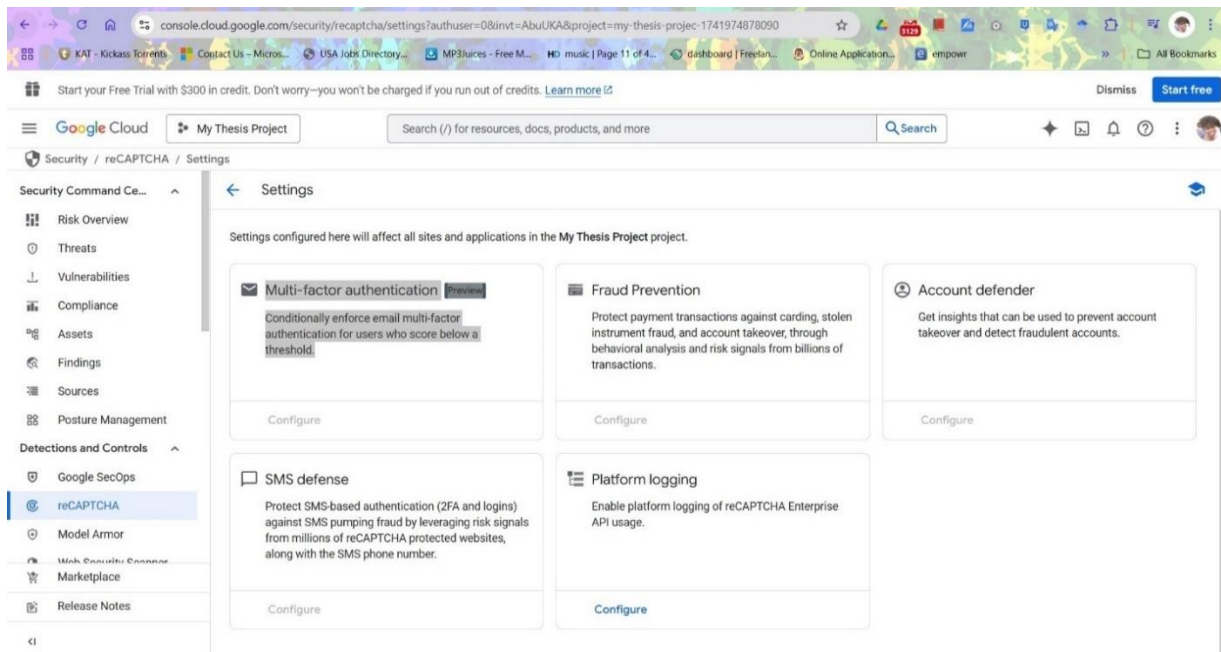


Figure 15: Google reCAPTCHA settings

## Appendix C: Functionality Tests

This section contains all the functionality tests done as well as the output/results.

### Login Logic Functionality Tests

Login page

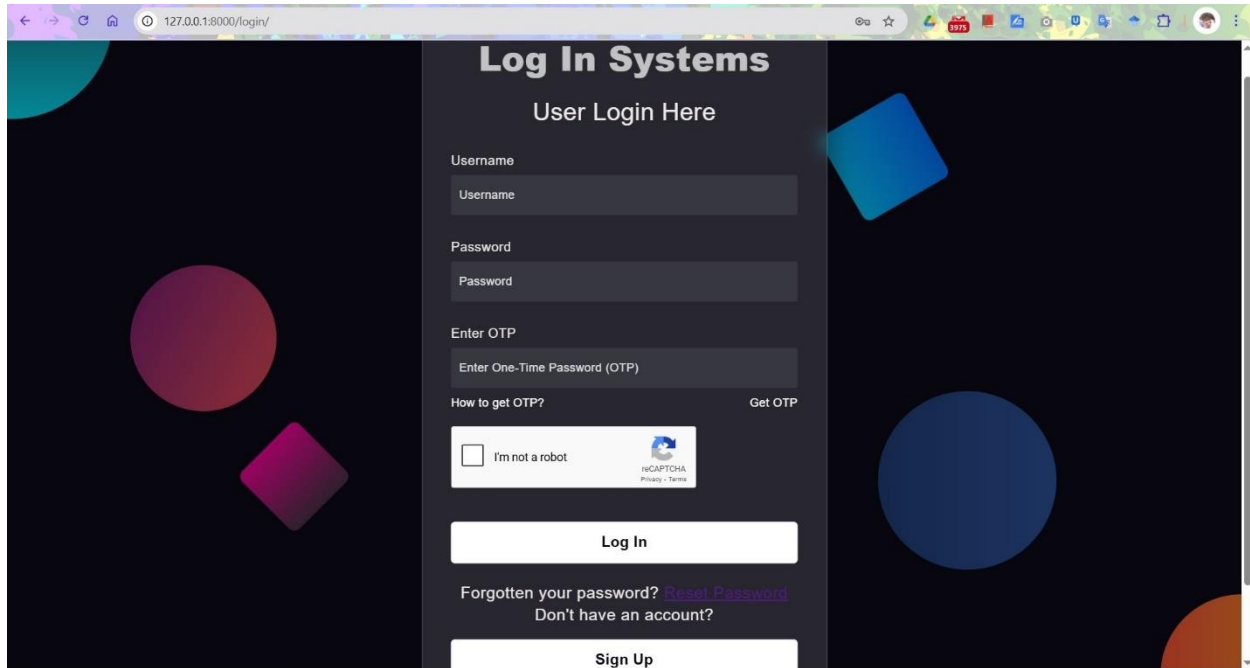


Figure 16: Login page

## Error Message when Credentials are invalid during Login

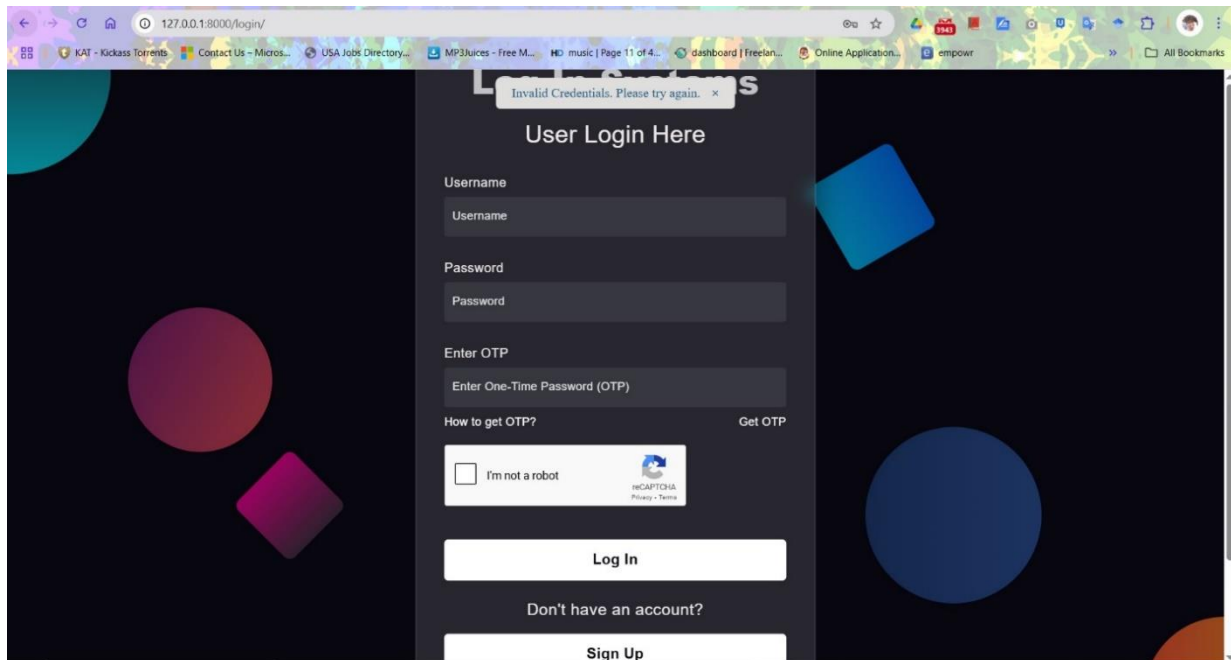


Figure 17: Login with Invalid Credentials

## Error Message when CAPTCHA is not verified during Login

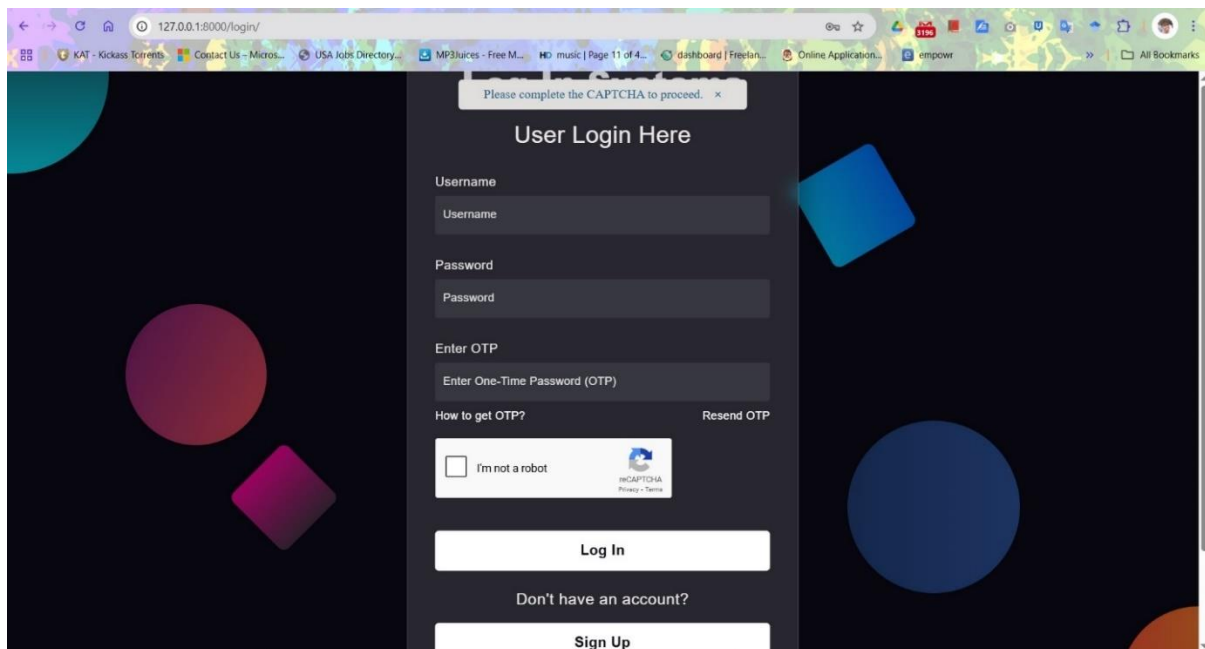
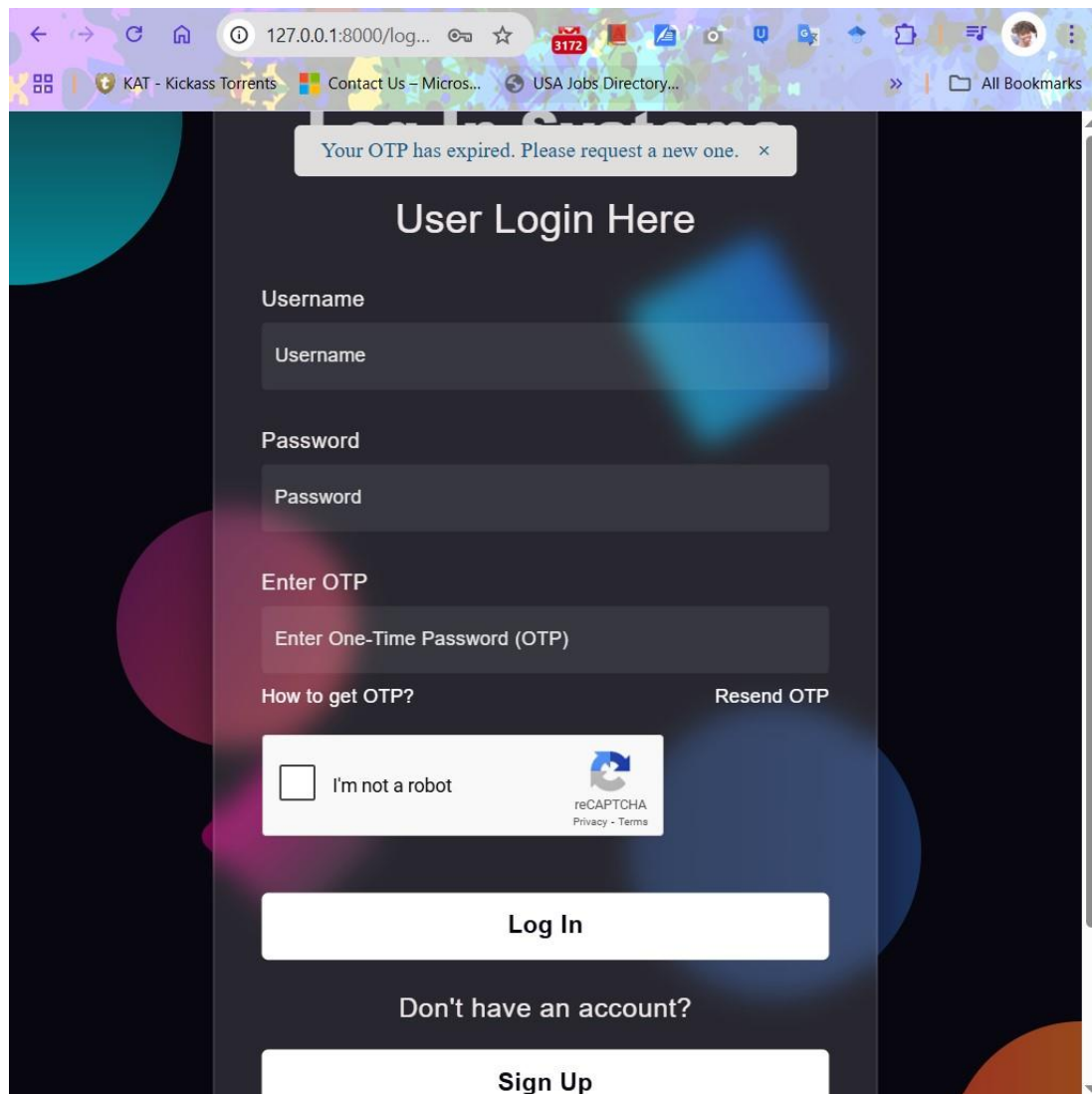


Figure 18: Invalid CAPTCHA

## Error Message when a time-based OTP is not verified due to expiration



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/log...". The browser's bookmark bar includes "KAT - Kickass Torrents", "Contact Us - Micros...", and "USA Jobs Directory...". A notification bar at the top of the page states: "Your OTP has expired. Please request a new one. x". Below this, the page is titled "User Login Here". The login form consists of the following elements:

- Username:** A text input field with the placeholder text "Username".
- Password:** A password input field with the placeholder text "Password".
- Enter OTP:** A text input field with the placeholder text "Enter One-Time Password (OTP)".
- How to get OTP?** A link located below the OTP input field.
- Resend OTP:** A link located to the right of the "How to get OTP?" link.
- reCAPTCHA:** A checkbox labeled "I'm not a robot" next to the reCAPTCHA logo and "reCAPTCHA Privacy - Terms" link.
- Log In:** A large white button with the text "Log In".
- Don't have an account?:** A link located below the "Log In" button.
- Sign Up:** A large white button with the text "Sign Up".

Figure 19: Unverified OTP

Successfully resent a time-based OTP to user's email

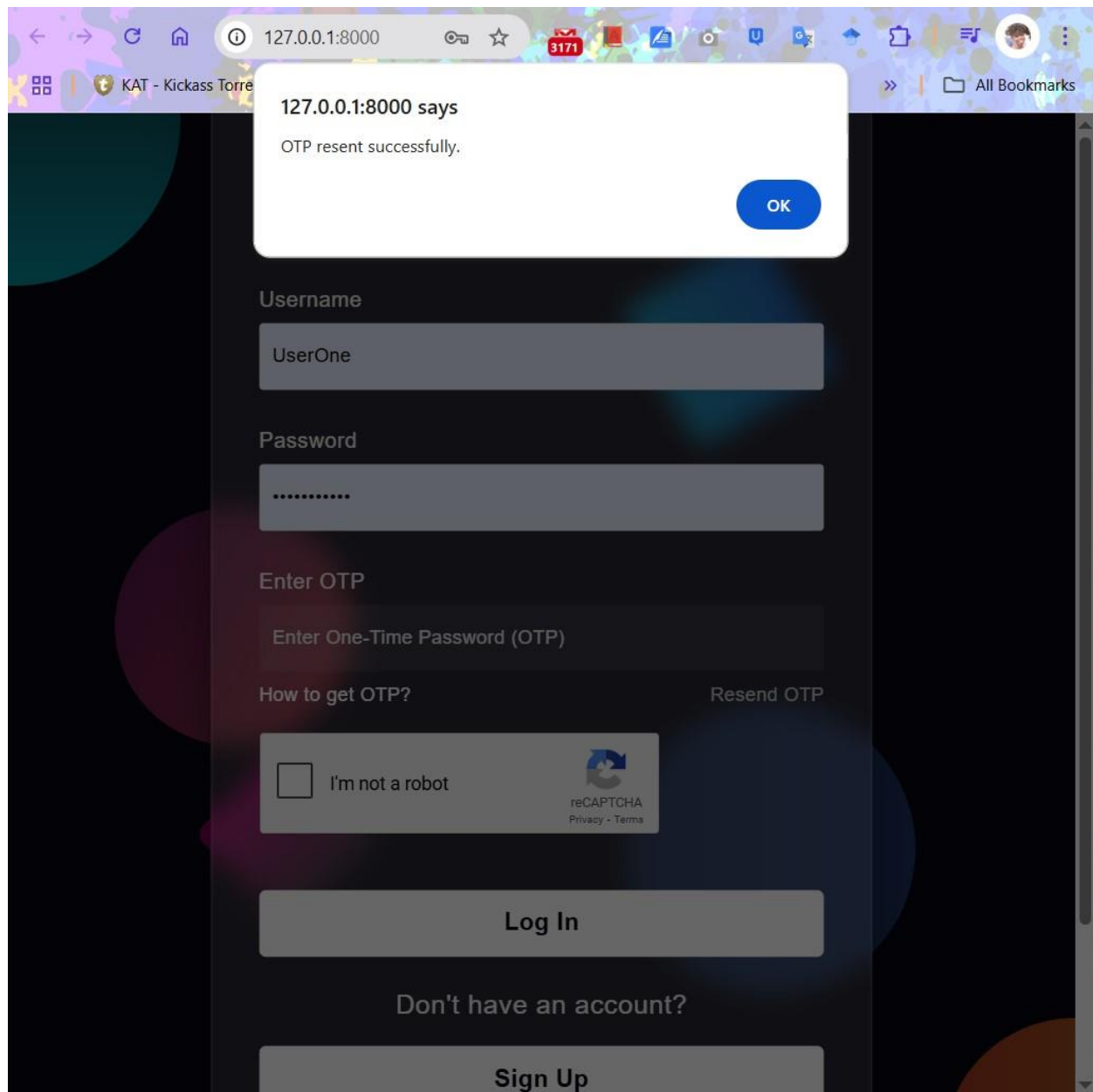


Figure 20: OTP resent successfully

## Email containing the OTP

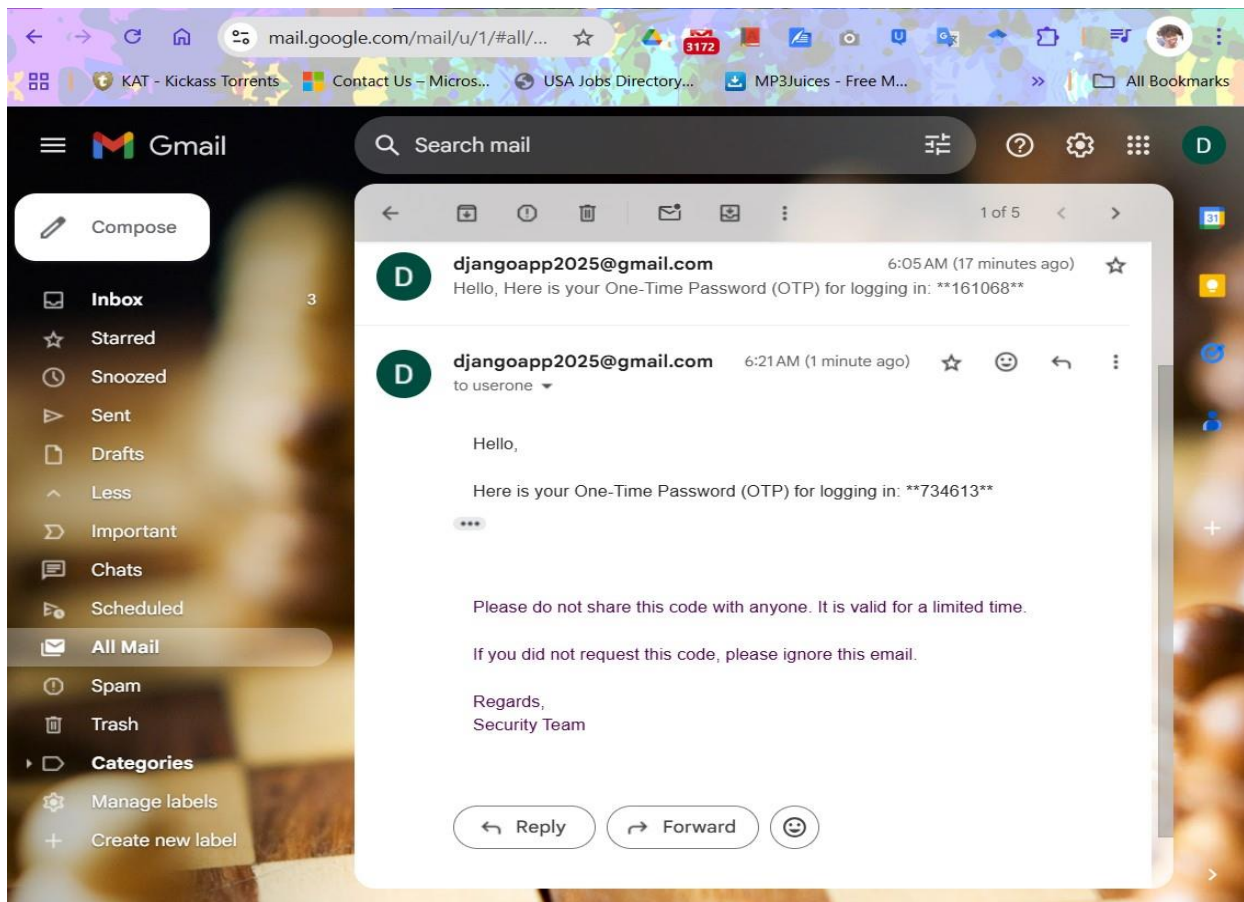


Figure 21: Email containing OTP

## Successful Login as Admin

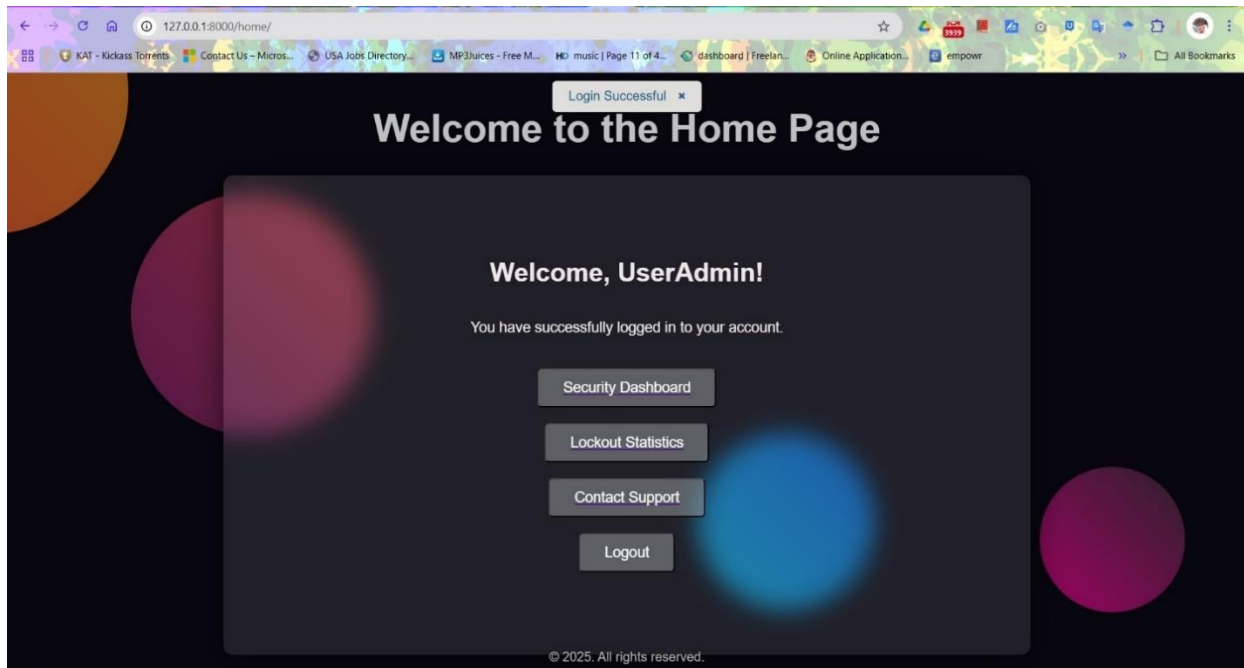


Figure 22: Login Successfully

## Admin Successfully logged out

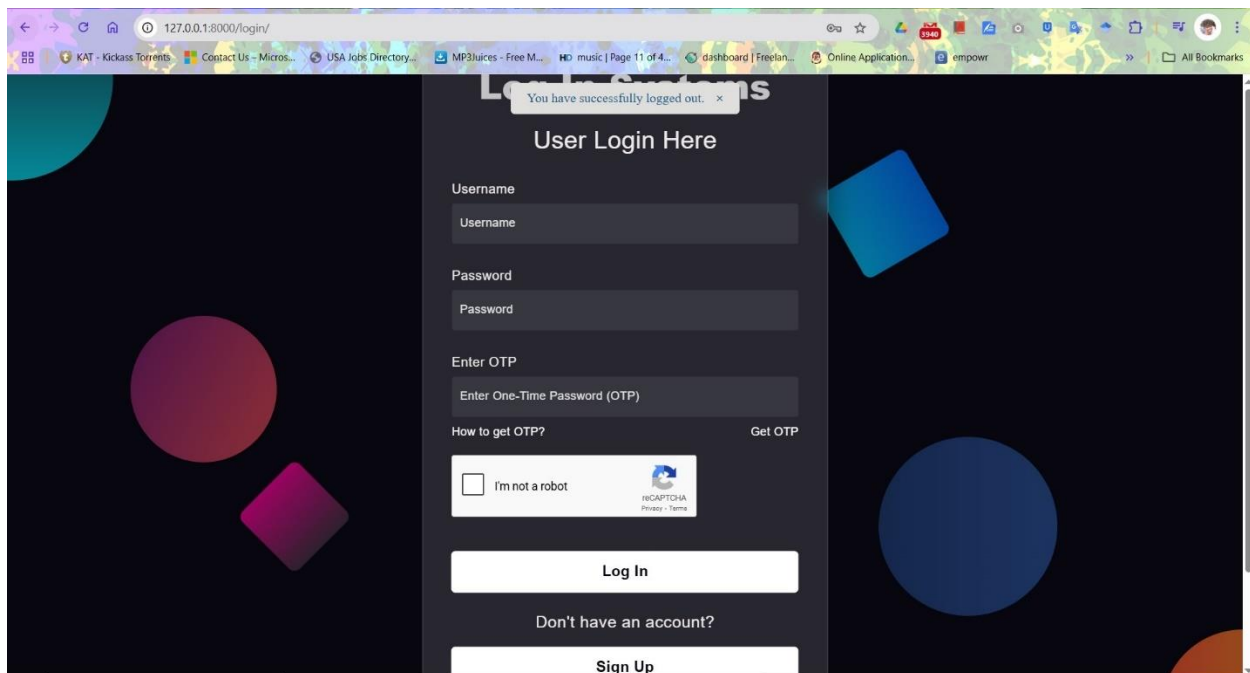
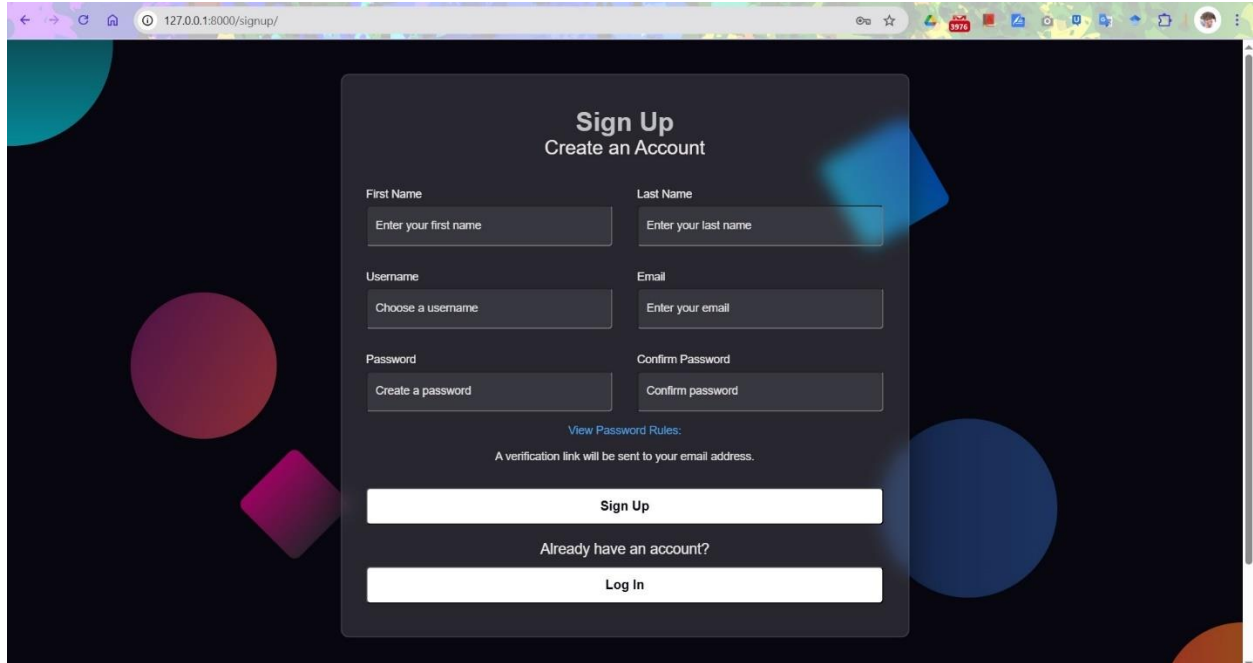


Figure 23: Successfully Logged Out

## Sign-up Logic Functionality Tests

### Signup page



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/signup/". The page features a dark background with colorful geometric shapes. A central white form titled "Sign Up" and "Create an Account" contains the following fields:

- First Name: Enter your first name
- Last Name: Enter your last name
- Username: Choose a username
- Email: Enter your email
- Password: Create a password
- Confirm Password: Confirm password

Below the fields, there is a link "View Password Rules:" and a note "A verification link will be sent to your email address." At the bottom of the form are two buttons: "Sign Up" and "Log In".

Figure 24: Signup page

A modal to help the user know password guidelines when creating a password

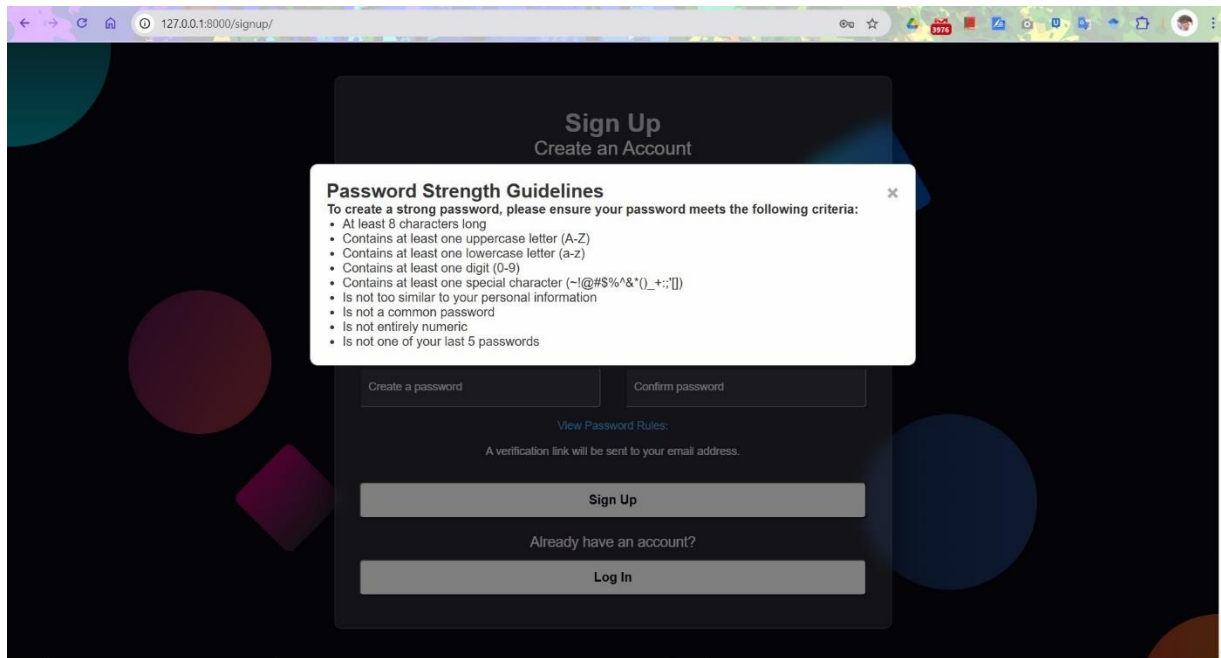


Figure 25: Password guide modal on signup page

Error handling due to Invalid Input of weak password and during Signup

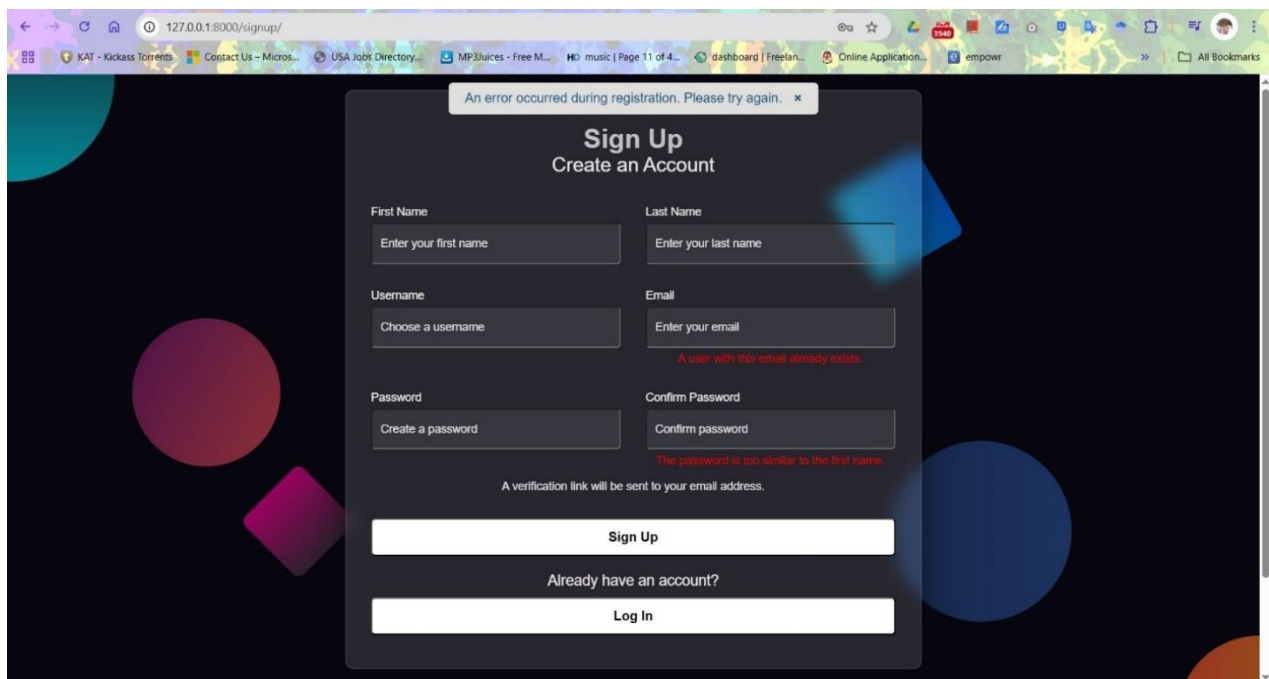


Figure 26: Signup Error Handling

After a valid input, the system notifies the user to check the email for a verification token. A verification email is sent to user so as to verify email

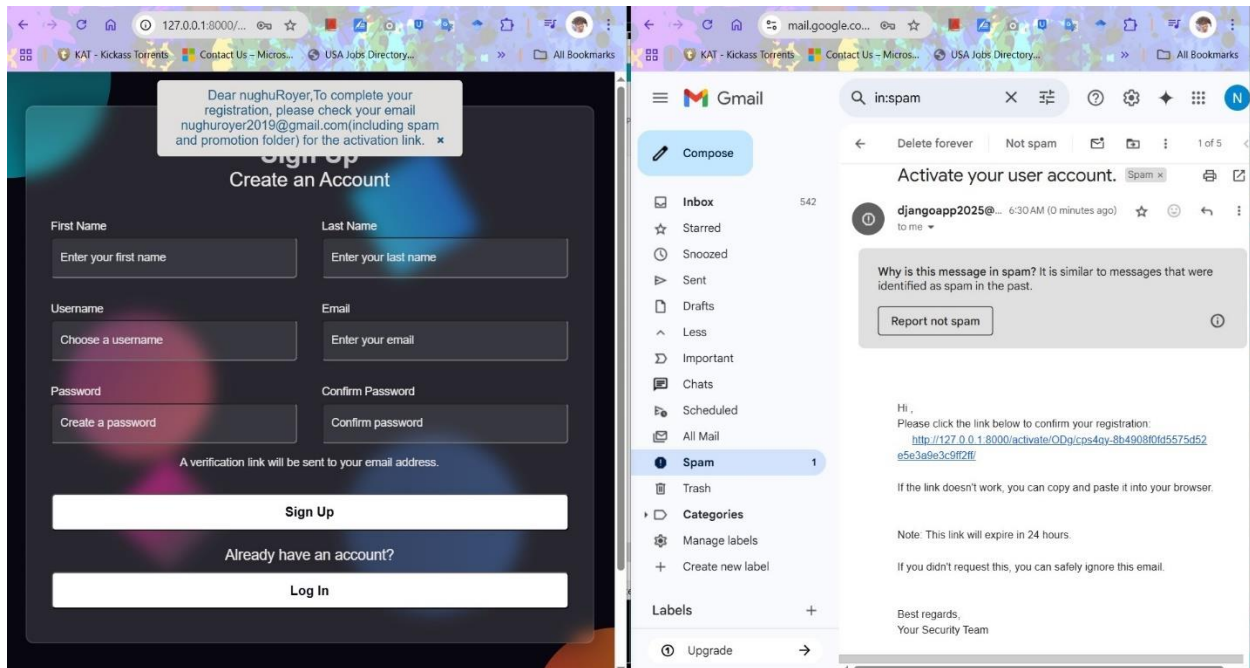


Figure 27: Email Verification

Once user has verified the email, the user is redirected to a 'Email successfully verified' page and from there the user can be redirected to login page to login.

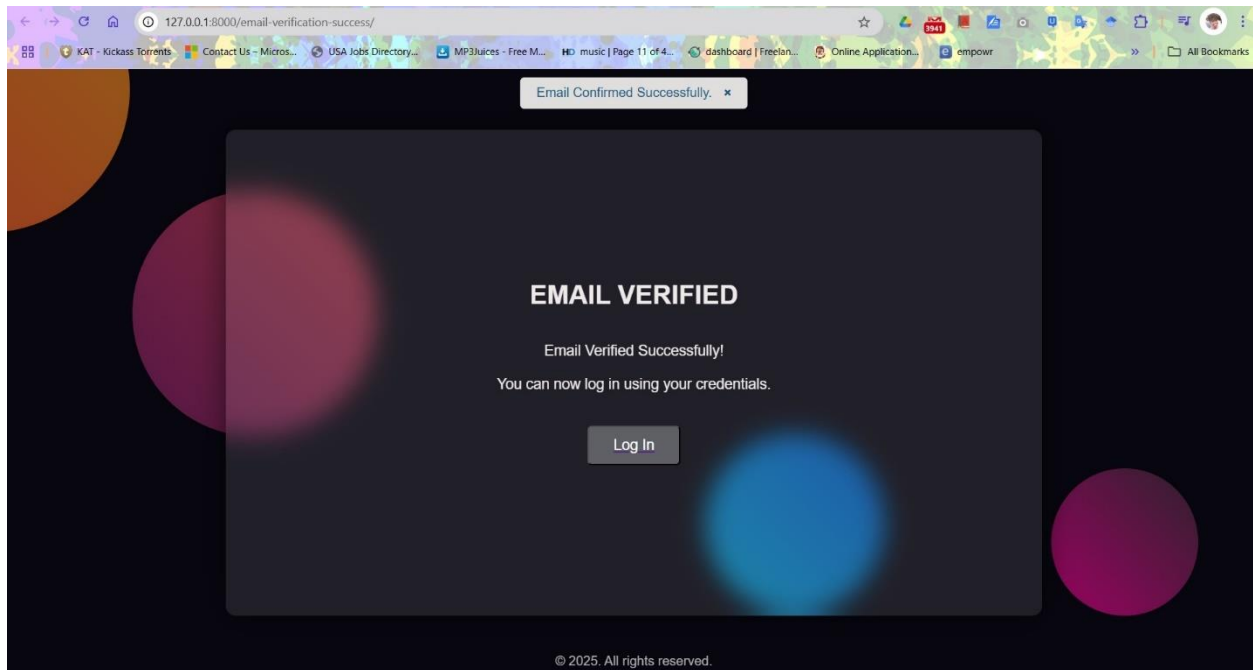


Figure 28: Email Verified Page

## Password Reset Logic Functionality Tests

From the Login page, the user can reset the password. Below is Password reset page handling errors caused by a user creating a weak password.

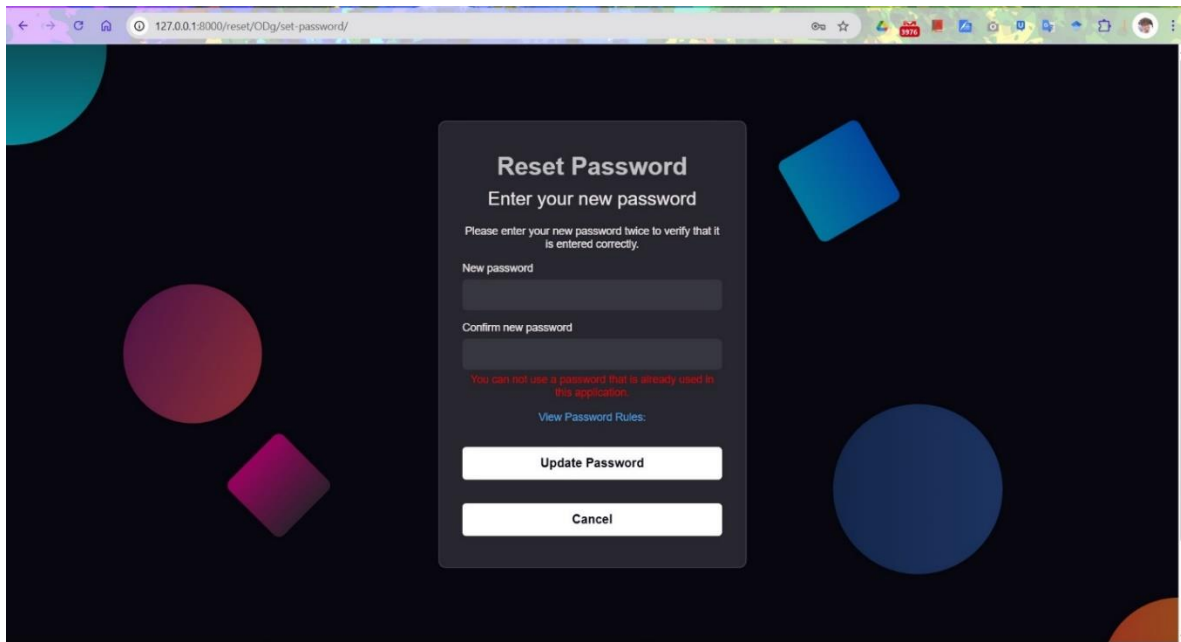


Figure 29: Password reset error handling 1

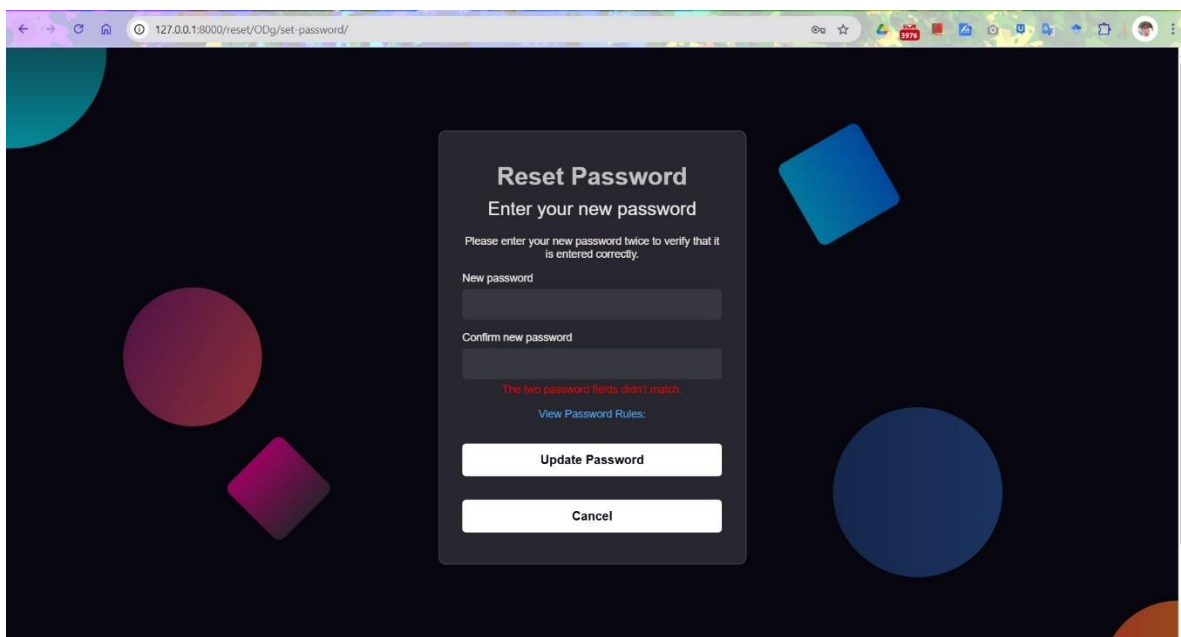


Figure 30: Password reset error handling 2

Successfully sent email with a password reset token

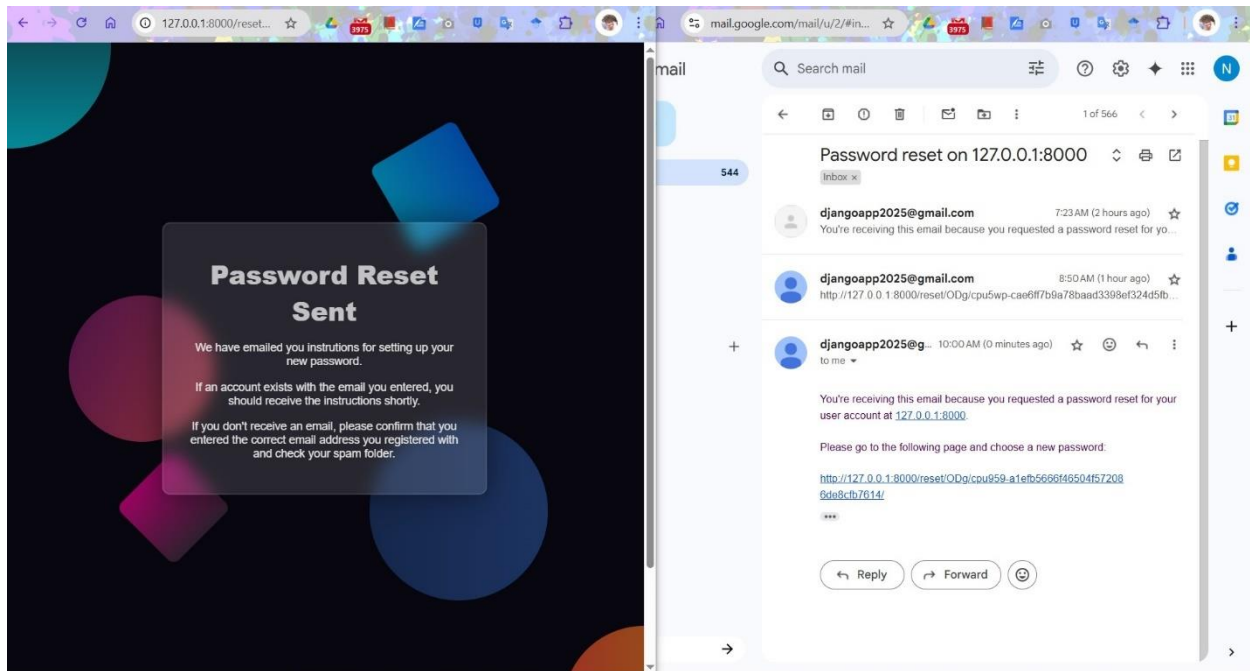


Figure 31: Password reset email

The user can see the guidelines for password so as to enhance its strength

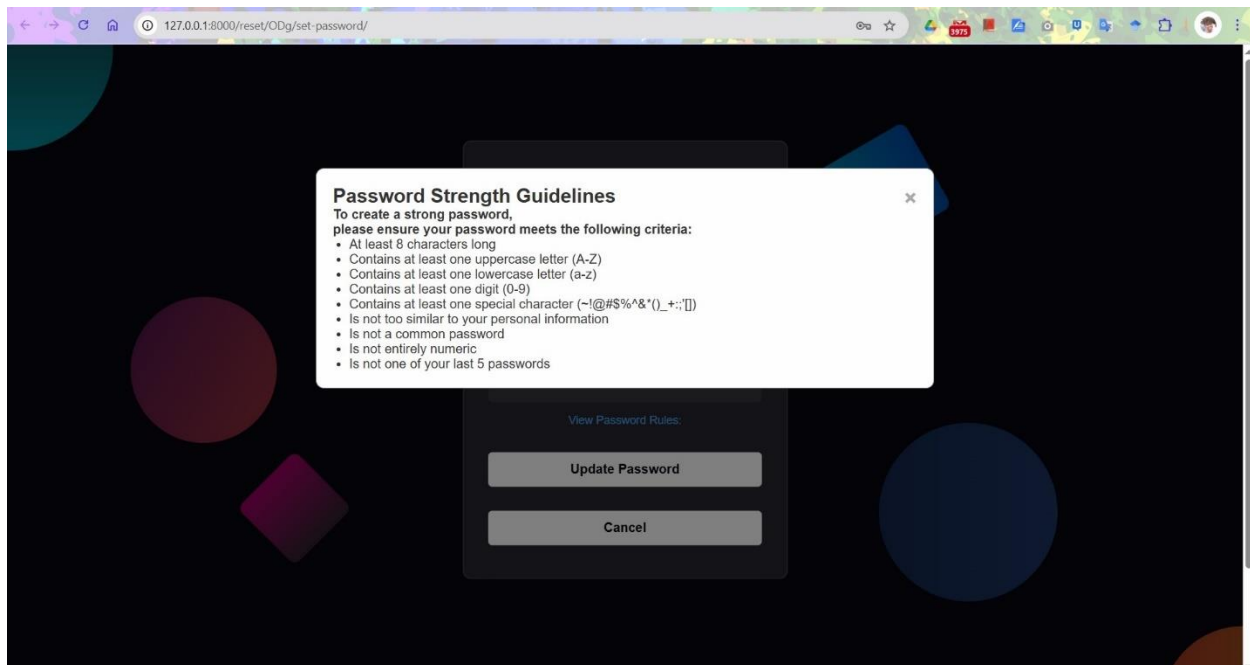


Figure 32: Password guide when resetting

User has successfully reset the password and they can now login with their new password

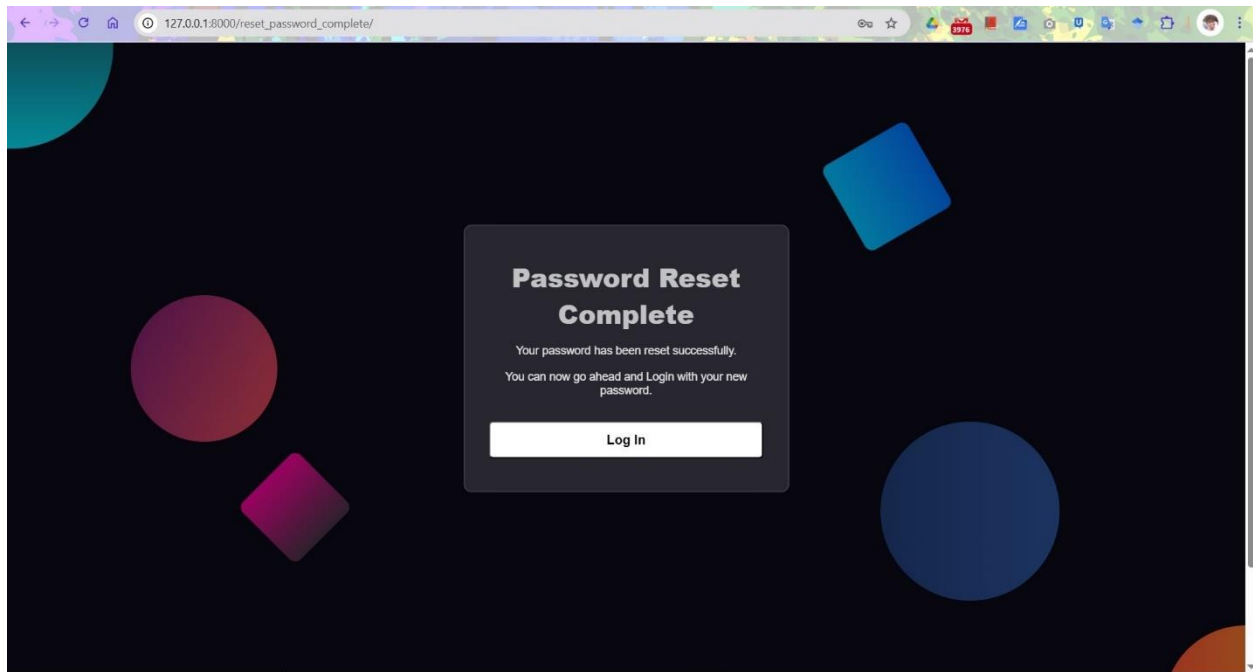


Figure 33: Password reset successful

### Lockout Logic Functionality Tests

After 3 failed login attempts, the user's account and IP are blocked then the user is redirected to the Lockout page successfully.

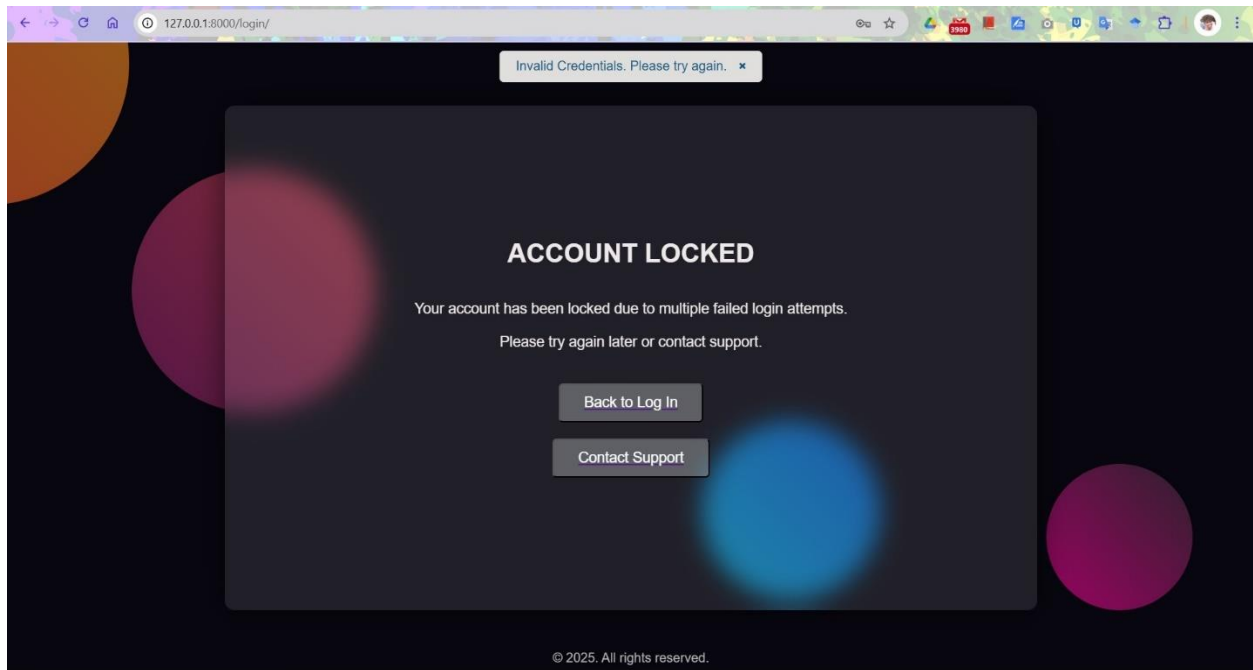


Figure 34: Account lockout page

Logging is then done and a Lockout email alert to admin is executed successfully.

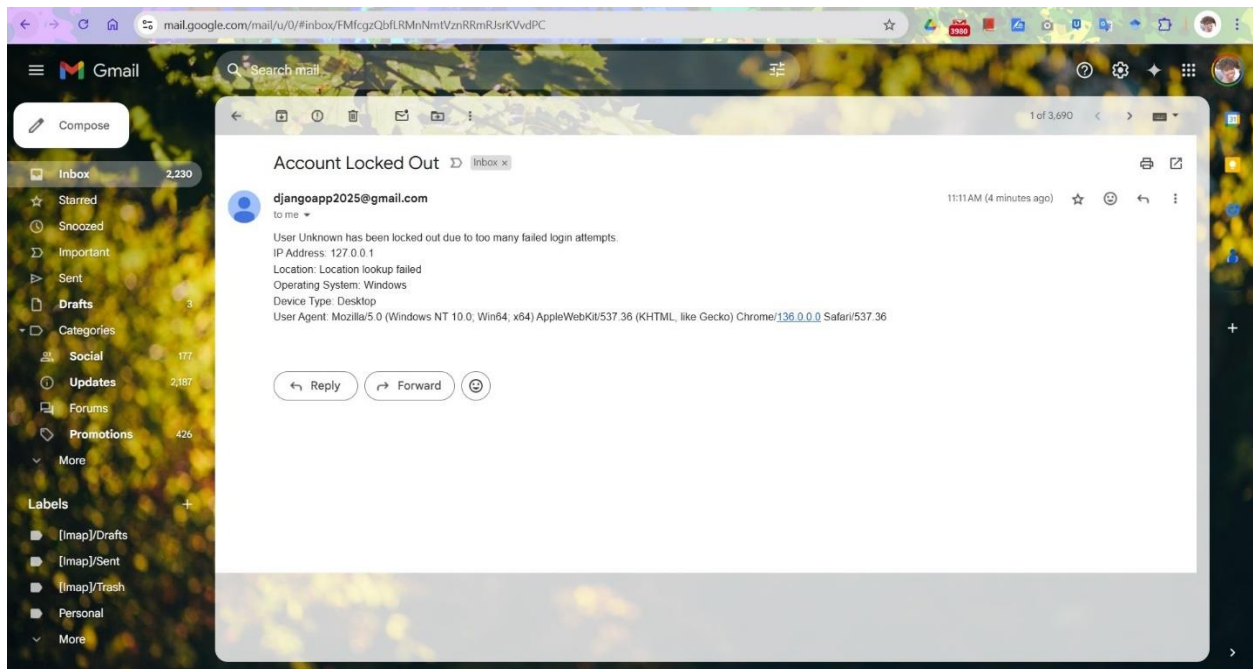


Figure 35: Lockout email to admin

When the user goes back to the Login page and tries to login again while the username and IP are locked, Login page will display a message informing that the account is locked and will be redirect to the locked-out page again.

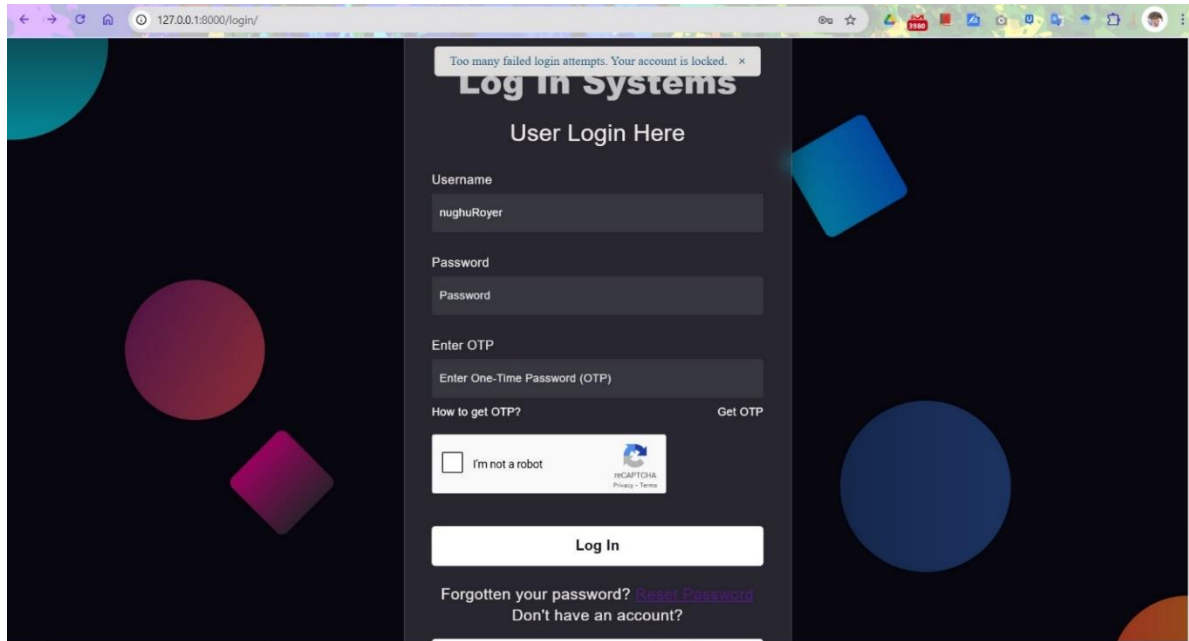


Figure 36: Account locked login page

The user will also receive an email about the 'locked account' situation.

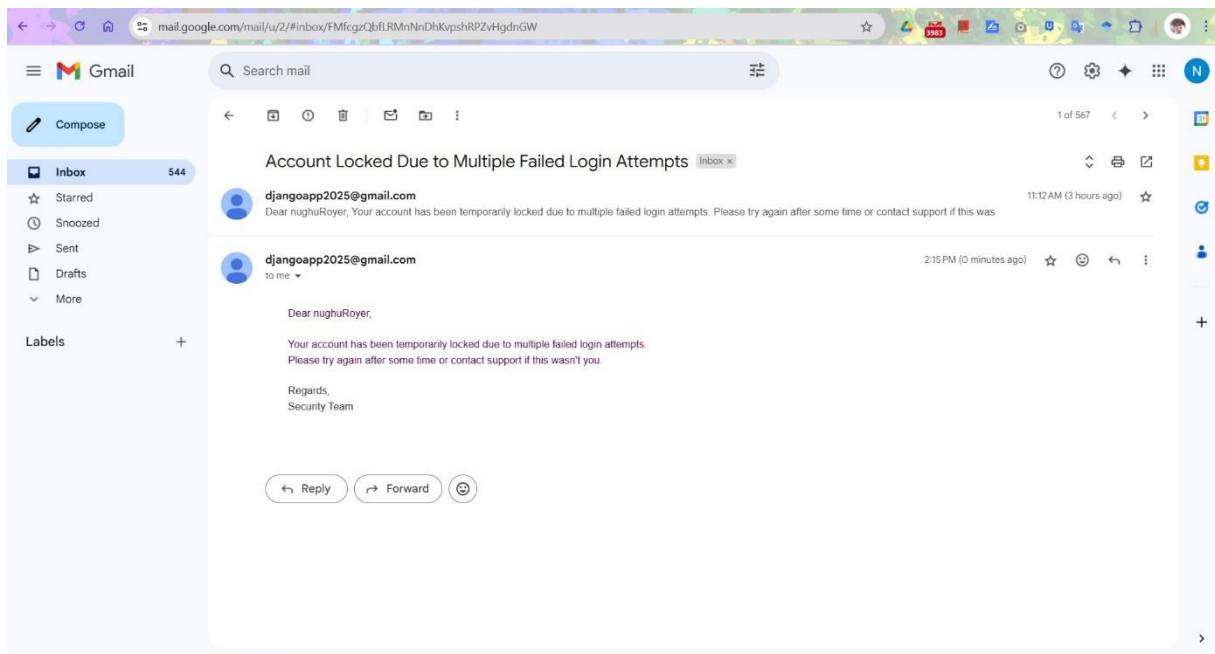
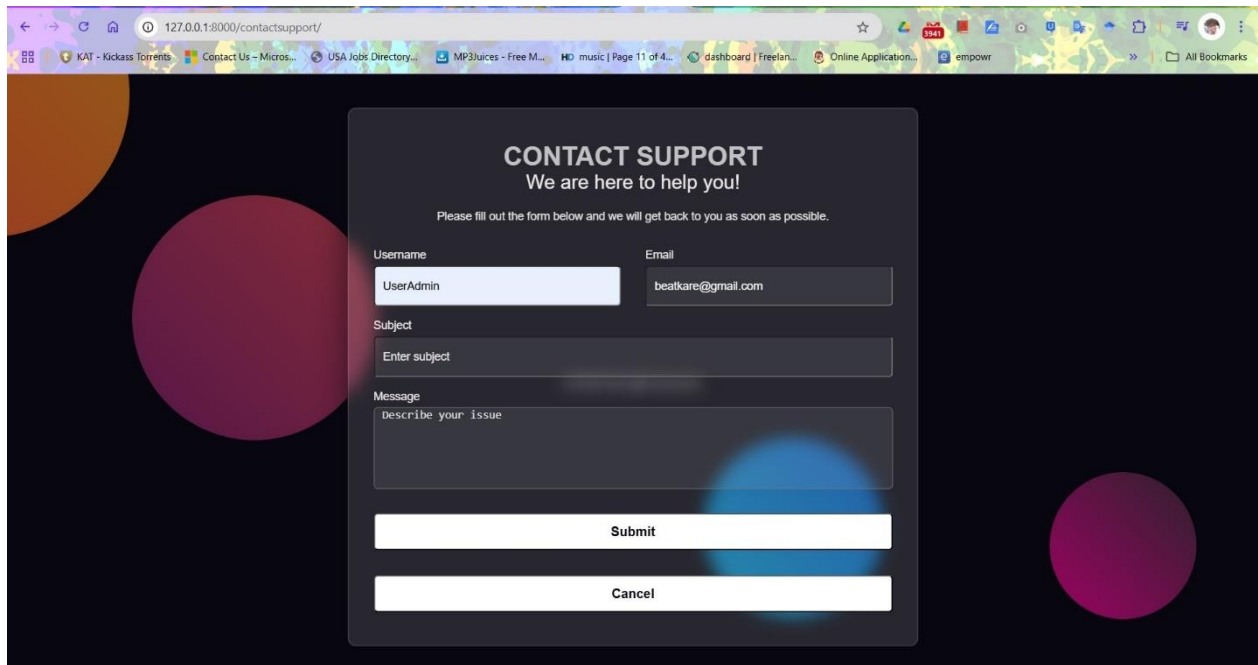


Figure 37: User notified of locked account

## Contact Support Logic Functionality Tests

Contact Support page automatically fills in the username and email if you access the page when signed in



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/contactsupport/". The page has a dark background with large, semi-transparent circles in orange, purple, and blue. A central white form titled "CONTACT SUPPORT" contains the text "We are here to help you!" and "Please fill out the form below and we will get back to you as soon as possible." The form includes fields for "Username" (pre-filled with "UserAdmin"), "Email" (pre-filled with "beatkare@gmail.com"), "Subject" (placeholder "Enter subject"), and "Message" (placeholder "Describe your issue"). At the bottom of the form are "Submit" and "Cancel" buttons.

Figure 38: Contact Support page

Contact Support Email Sent to the Admin Successfully.

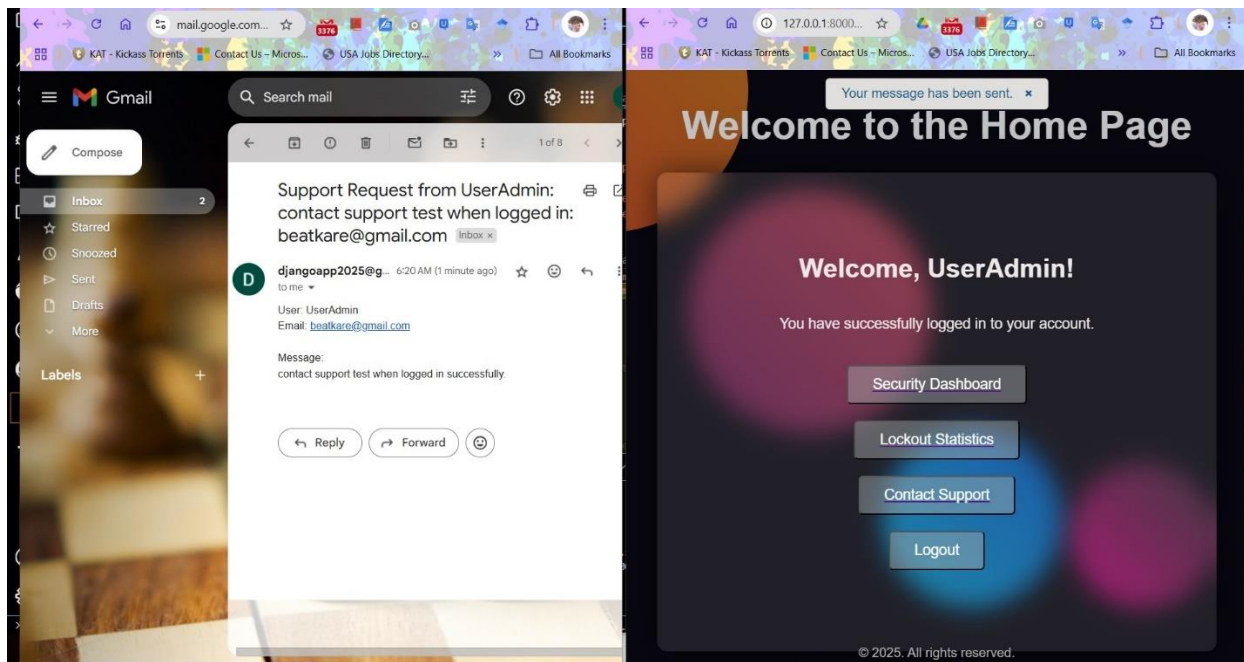


Figure 39: Customer Support Email Success

## Session Expiry Functionality Test

The session cookie is executing successfully, as it logs out the user after *25 minutes* of inactivity (as integrated in the system). This helps enhance security by reducing the risk of unauthorized access from idle sessions, ensuring that user accounts remain protected from potential session hijacking or misuse.

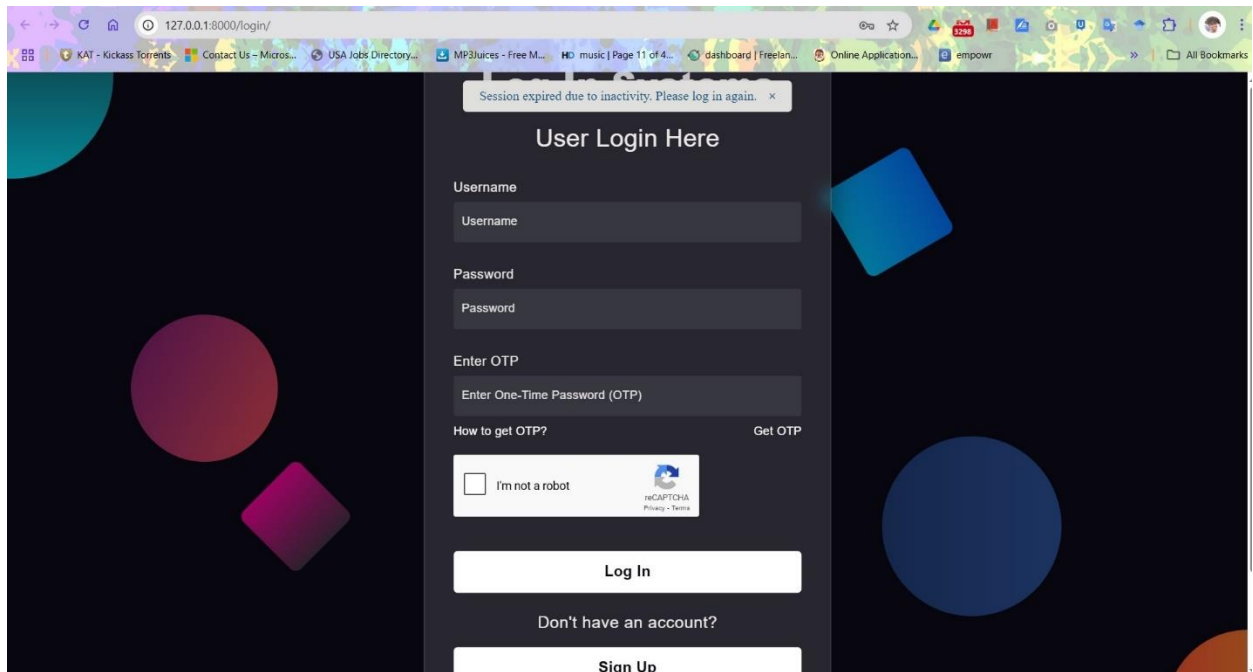


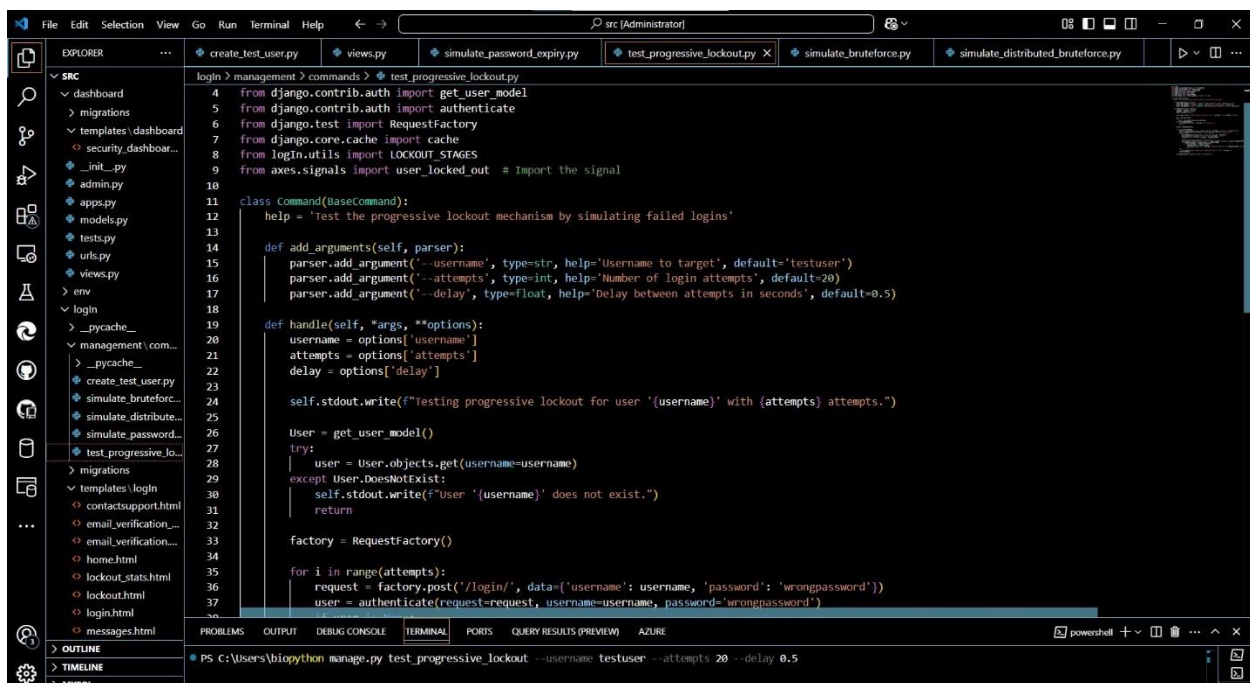
Figure 40: Session Expiry Test

## Appendix D: Simulation Tests and Results

All the simulations tests done as well as their results/outputs are described in this section.

### Progressive Lockout Simulation Tests and Results

A simulation was created and ran (*python manage.py test\_progressive\_lockout --username testuser --attempts 20 --delay 0.5*), below is the source code



```
login > management > commands > test_progressive_lockout.py
4 from django.contrib.auth import get_user_model
5 from django.contrib.auth import authenticate
6 from django.test import RequestFactory
7 from django.core.cache import cache
8 from logIn.utils import LOCKOUT_STAGES
9 from axes.signals import user_locked_out # Import the signal
10
11 class Command(BaseCommand):
12     help = 'test the progressive lockout mechanism by simulating failed logins'
13
14     def add_arguments(self, parser):
15         parser.add_argument('--username', type=str, help='Username to target', default='testuser')
16         parser.add_argument('--attempts', type=int, help='Number of login attempts', default=20)
17         parser.add_argument('--delay', type=float, help='Delay between attempts in seconds', default=0.5)
18
19     def handle(self, *args, **options):
20         username = options['username']
21         attempts = options['attempts']
22         delay = options['delay']
23
24         self.stdout.write(f"Testing progressive lockout for user '{username}' with {attempts} attempts.")
25
26         User = get_user_model()
27         try:
28             user = User.objects.get(username=username)
29         except User.DoesNotExist:
30             self.stdout.write(f"User '{username}' does not exist.")
31             return
32
33         factory = RequestFactory()
34
35         for i in range(attempts):
36             request = factory.post('/login/', data={'username': username, 'password': 'wrongpassword'})
37             user = authenticate(request=request, username=username, password='wrongpassword')
```

Figure 41:simulate\_progressive\_lockout code

The results below shows that the lockout test passed, and the progressive lockout mechanism is working as intended.

```

login > management > commands > test progressive lockout.py
9 from axes.signals import user_locked_out # Import the signal
10
11 class Command(BaseCommand):
12     help = 'Test the progressive lockout mechanism by simulating failed logins'
13
14     def add_arguments(self, parser):
15         parser.add_argument('--username', type=str, help='Username to target', default='testuser')
16         parser.add_argument('--attempts', type=int, help='Number of login attempts', default=30)
17
18 PS C:\Users\biopython manage.py test progressive_lockout --username testuser --attempts 20 --delay 0.5
19
20 System check identified some issues:
21
22 WARNINGS:
23 ? (urls.uses) URL namespace 'admin' isn't unique. You may not be able to reverse all URLs in this namespace
24
25 Testing progressive lockout for user 'testuser' with 20 attempts.
26 Attempt 1: Failed login for user 'testuser'
27 Attempt 2: Failed login for user 'testuser'
28 Attempt 3: Failed login for user 'testuser'
29 User 'testuser' should be locked out for 5 minutes after 3 failed attempts.
30 Attempt 4: Failed login for user 'testuser'
31 User 'testuser' should be locked out for 5 minutes after 4 failed attempts.
32 Attempt 5: Failed login for user 'testuser'
33 User 'testuser' should be locked out for 5 minutes after 5 failed attempts.
34 Attempt 6: Failed login for user 'testuser'
35 User 'testuser' should be locked out for 5 minutes after 6 failed attempts.
36 Attempt 7: Failed login for user 'testuser'
37 User 'testuser' should be locked out for 5 minutes after 7 failed attempts.
38 Attempt 8: Failed login for user 'testuser'
39 User 'testuser' should be locked out for 5 minutes after 8 failed attempts.
40 Attempt 9: Failed login for user 'testuser'
41 User 'testuser' should be locked out for 5 minutes after 9 failed attempts.
42 Attempt 10: Failed login for user 'testuser'
43 User 'testuser' should be locked out for 5 minutes after 10 failed attempts.
44 Attempt 11: Failed login for user 'testuser'
45 User 'testuser' should be locked out for 5 minutes after 11 failed attempts.
46 Attempt 12: Failed login for user 'testuser'
47 User 'testuser' should be locked out for 5 minutes after 12 failed attempts.
48 Attempt 13: Failed login for user 'testuser'
49 User 'testuser' should be locked out for 5 minutes after 13 failed attempts.

```

Figure 42: simulate\_progressive\_lockout result1

```

Attempt 14: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 14 failed attempts.
Attempt 15: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 15 failed attempts.
Attempt 16: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 16 failed attempts.
Attempt 17: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 17 failed attempts.
Attempt 18: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 18 failed attempts.
Attempt 19: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 19 failed attempts.
Attempt 20: Failed login for user 'testuser'
User 'testuser' should be locked out for 5 minutes after 20 failed attempts.
Progressive lockout test completed.
o User 'testuser' should be locked out for 5 minutes after 20 failed attempts.
User 'testuser' should be locked out for 5 minutes after 20 failed attempts.
Progressive lockout test completed.
PS C:\Users\biopython\Desktop\Folders\Univ ESSEX\Thesis\Assignments\Project\src>

```

Figure 43:simulate\_progressive\_lockout result2

Here are the key points from the output:

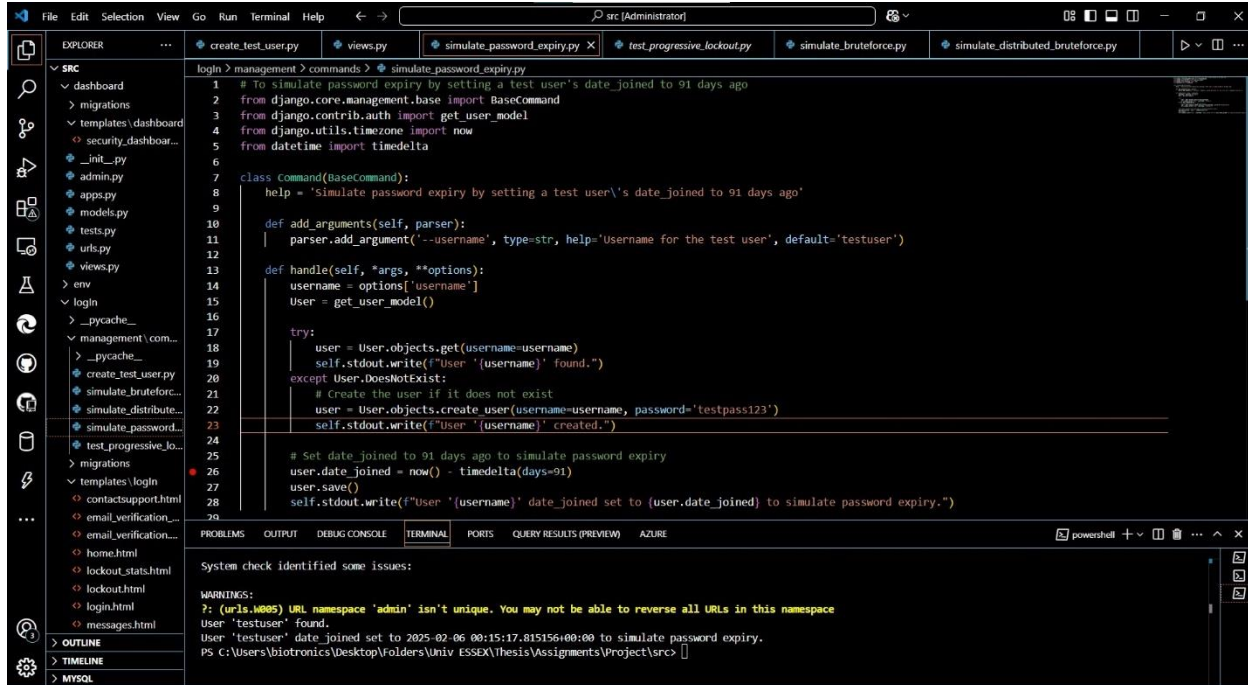
- **Failed Login Attempts:** The output shows that for each failed login attempt, the system correctly identifies that the user is locked out after a certain number of failed attempts.
- **Lockout Duration:** The message "User 'testuser' should be locked out for 5 minutes after X failed attempts" confirms that the lockout duration is being applied progressively as expected.
- **Completion of Test:** The message "Progressive lockout test completed." indicates that the test ran to completion without any unhandled exceptions.

The system is correctly tracking failed login attempts and applying the lockout policy based on the defined LOCKOUT\_STAGES.

### **Password Expiration and Change Simulation Tests and Results**

To test forced password expiration after 90 days, a simulation was created and run using the command `(python manage.py simulate_password_expiry --username testuser)`. This command sets the *testuser* account's join date to 91 days ago, which

triggers the password expiration (after 90 days), prompting a password change.



```
login > management > commands > simulate_password_expiry.py
1 # To simulate password expiry by setting a test user's date_joined to 91 days ago
2 from django.core.management.base import BaseCommand
3 from django.contrib.auth import get_user_model
4 from django.utils.timezone import now
5 from datetime import timedelta
6
7 class Command(BaseCommand):
8     help = 'Simulate password expiry by setting a test user's date_joined to 91 days ago'
9
10    def add_arguments(self, parser):
11        | parser.add_argument('--username', type=str, help='Username for the test user', default='testuser')
12
13    def handle(self, *args, **options):
14        username = options['username']
15        User = get_user_model()
16
17        try:
18            user = User.objects.get(username=username)
19            self.stdout.write(f"User '{username}' found.")
20        except User.DoesNotExist:
21            # Create the user if it does not exist
22            user = User.objects.create_user(username=username, password='testpass123')
23            self.stdout.write(f"User '{username}' created.")
24
25        # Set date_joined to 91 days ago to simulate password expiry
26        user.date_joined = now() - timedelta(days=91)
27        user.save()
28        self.stdout.write(f"User '{username}' date_joined set to {user.date_joined} to simulate password expiry.")
```

System check identified some issues:

WARNINGS:

? (urls.W005) URL namespace 'admin' isn't unique. You may not be able to reverse all URLs in this namespace

User 'testuser' found.

User 'testuser' date\_joined set to 2025-02-06 00:15:17.815156+00:00 to simulate password expiry.

PS C:\Users\biotronics\Desktop\Folders\Univ ESSEX\Thesis\Assignments\Project\src> []

Figure 44: simulate\_password\_expiry Code

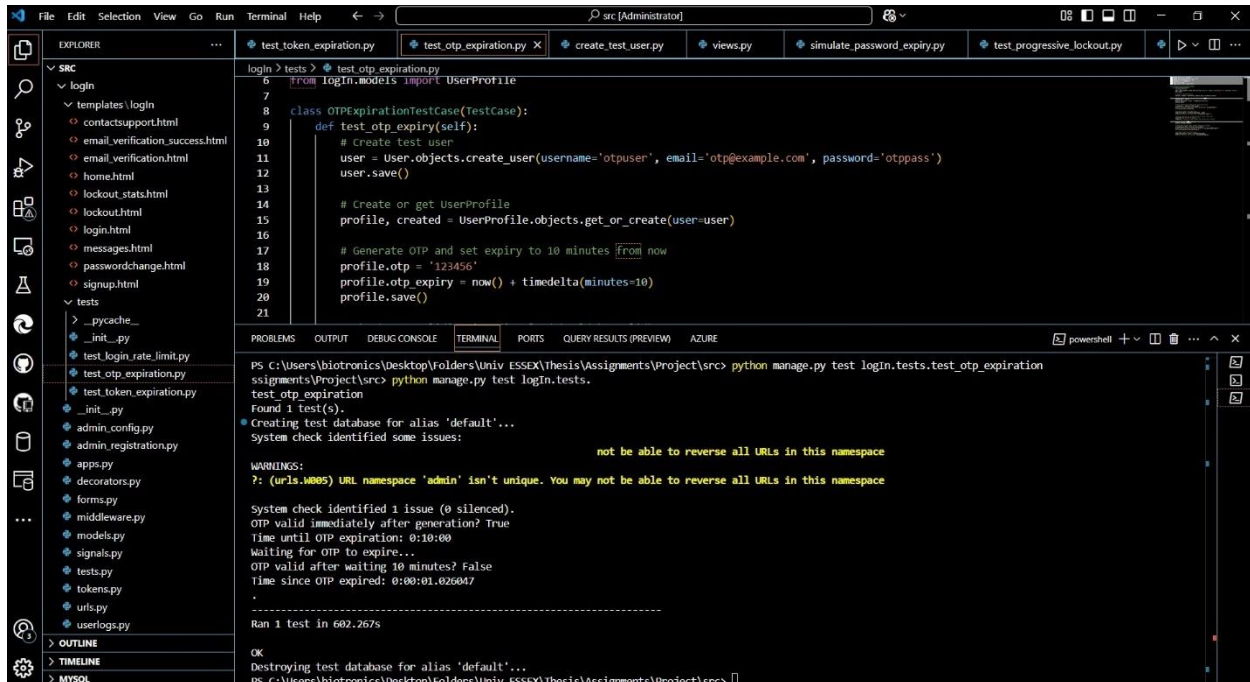
The simulation ran successfully, and the system correctly prompted the user to change their password after 90 days, as expected.

## Time-Based OTP Expiration (Login) Simulation Tests and Results

To test how long OTP takes to expire during login, a simulation was created and ran.

The command used: `python manage.py test login.tests.test_otp_expiration`

Below are the code snippets and results/output:



```
login > tests > test_otp_expiration.py
6 from login.models import UserProfile
7
8 class OTPExpirationTestCase(TestCase):
9     def test_otp_expiry(self):
10         # Create test user
11         user = User.objects.create_user(username='otpuser', email='otp@example.com', password='otppass')
12         user.save()
13
14         # Create or get UserProfile
15         profile, created = UserProfile.objects.get_or_create(user=user)
16
17         # Generate OTP and set expiry to 10 minutes from now
18         profile.otp = '123456'
19         profile.otp_expiry = now() + timedelta(minutes=10)
20         profile.save()
21
```

```
PS C:\Users\biotronics\Desktop\Folders\Univ ESSEX\Thesis\Assignments\Project\src> python manage.py test login.tests.test_otp_expiration
ssignments\Project\src> python manage.py test login.tests.
test_otp_expiration
Found 1 test(s).
Creating test database for alias 'default'...
System check identified some issues:

WARNINGS:
?: (urls.W005) URL namespace 'admin' isn't unique. You may not be able to reverse all URLs in this namespace

System check identified 1 issue (0 silenced).
OTP valid immediately after generation? True
Time until OTP expiration: 0:10:00
Waiting for OTP to expire...
OTP valid after waiting 10 minutes? False
Time since OTP expired: 0:00:01.026047
.
-----
Ran 1 test in 602.267s

OK
Destroying test database for alias 'default'...
PS C:\Users\biotronics\Desktop\Folders\Univ ESSEX\Thesis\Assignments\Project\src>
```

Figure 45: test\_otp\_expiration code and results

Here are the key points from the output:

- The message *OTP valid immediately after generation? True* confirms that the OTP is functional and valid right after it is created, as expected.
- The output *Time until OTP expiration: 0:10:00* shows that the OTP is configured to expire after exactly 10 minutes, which aligns with the defined security settings.
- The message *OTP valid after waiting 10 minutes? False* indicates that the OTP is invalid after the expiration period, demonstrating the expiration logic works correctly.

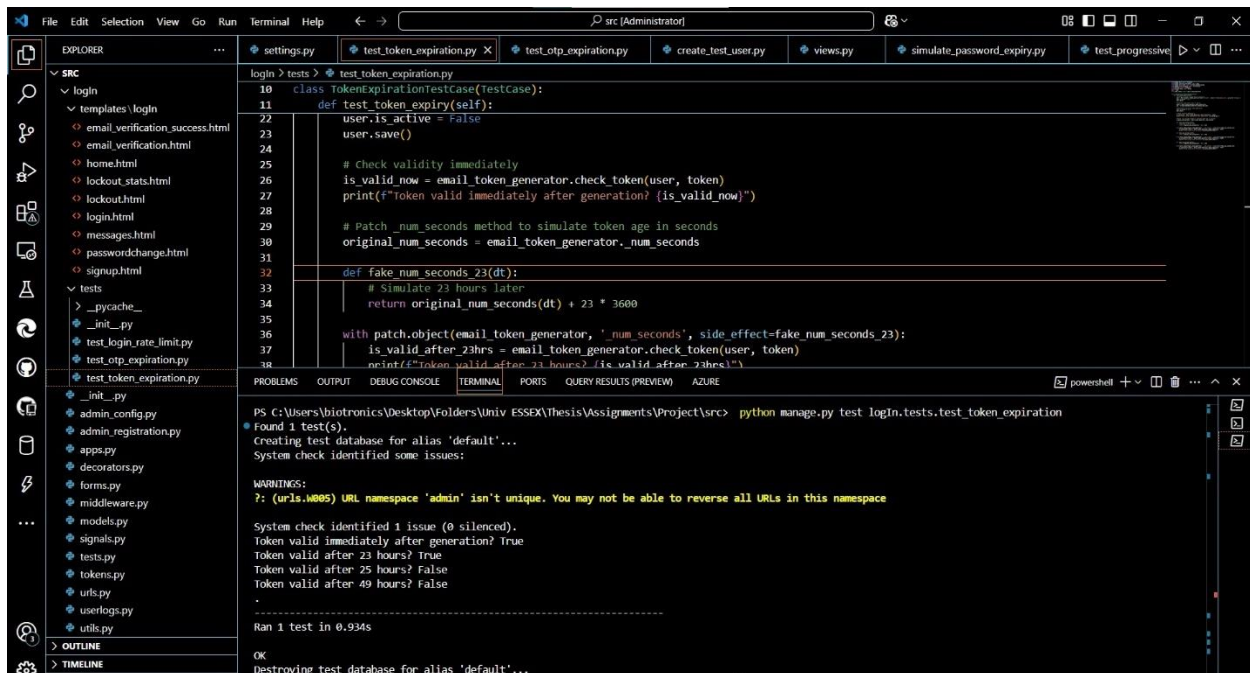
- The line *Ran 1 test in 0.934s with a result of OK* confirms the test executed fully with no errors or exceptions.

The system successfully enforces the 10-minute OTP expiration rule. It validates OTPs immediately after generation and correctly invalidates them after the configured timeout, ensuring strong time-based security control.

## Time-Based Email Token Expiration (Signup) Simulation Tests and Results

To test how long email token takes to expire during signup, a simulation was created and ran. The command used: *python manage.py test login.tests.test\_token\_expiration*

Below are the code snippets and results/output:



The screenshot shows a code editor with the file `test_token_expiration.py` open. The code defines a `TokenExpirationTestCase` class with a `test_token_expiry` method. The method tests the token's validity immediately after generation and after 23 hours. It uses a patch to simulate the token's age in seconds.

```

10 class TokenExpirationTestCase(TestCase):
11     def test_token_expiry(self):
12         user.is_active = False
13         user.save()
14
15         # Check validity immediately
16         is_valid_now = email_token_generator.check_token(user, token)
17         print(f"Token valid immediately after generation? {is_valid_now}")
18
19         # Patch _num_seconds method to simulate token age in seconds
20         original_num_seconds = email_token_generator._num_seconds
21
22         def fake_num_seconds_23(dt):
23             # Simulate 23 hours later
24             return original_num_seconds(dt) + 23 * 3600
25
26         with patch.object(email_token_generator, '_num_seconds', side_effect=fake_num_seconds_23):
27             is_valid_after_23hrs = email_token_generator.check_token(user, token)
28             print(f"Token valid after 23 hours? {is_valid_after_23hrs}")
29
30         self.assertTrue(is_valid_now)
31         self.assertFalse(is_valid_after_23hrs)
32
33     def test_token_expiry_25hrs(self):
34         # Simulate 25 hours later
35         return original_num_seconds(dt) + 25 * 3600
36
37         with patch.object(email_token_generator, '_num_seconds', side_effect=fake_num_seconds_25hrs):
38             is_valid_after_25hrs = email_token_generator.check_token(user, token)
39             print(f"Token valid after 25 hours? {is_valid_after_25hrs}")
40
41         self.assertTrue(is_valid_now)
42         self.assertFalse(is_valid_after_25hrs)
43
44     def test_token_expiry_49hrs(self):
45         # Simulate 49 hours later
46         return original_num_seconds(dt) + 49 * 3600
47
48         with patch.object(email_token_generator, '_num_seconds', side_effect=fake_num_seconds_49hrs):
49             is_valid_after_49hrs = email_token_generator.check_token(user, token)
50             print(f"Token valid after 49 hours? {is_valid_after_49hrs}")
51
52         self.assertTrue(is_valid_now)
53         self.assertFalse(is_valid_after_49hrs)

```

The terminal output shows the command `python manage.py test login.tests.test_token_expiration` being executed. It reports that 1 test was found and passed. The output also includes a warning about a non-unique URL namespace and a system check identifying 1 issue.

```

PS C:\Users\biotronics\Desktop\Folders\Univ ESSEX\Thesis\Assignments\Project\src> python manage.py test login.tests.test_token_expiration
Found 1 test(s).
Creating test database for alias 'default'...
System check identified some issues:

WARNINGS:
?: (urls.W005) URL namespace 'admin' isn't unique. You may not be able to reverse all URLs in this namespace

System check identified 1 issue (0 silenced).
Token valid immediately after generation? True
Token valid after 23 hours? True
Token valid after 25 hours? False
Token valid after 49 hours? False
.
Ran 1 test in 0.934s
OK
Destroying test database for alias 'default'...

```

Figure 46: *test\_token\_expiration* code and results

Here are the key points from the output:

- The message *Token valid immediately after generation? True* confirms that the Token is functional and valid right after it is created, as expected.
- The output *Token valid after 23 hours? True* shows that the Token is still valid after 23 hours, which aligns with the defined security settings of 24hours validity.
- The output *Token valid after 25 hours? False* shows that the Token is still invalid after 25 hours, demonstrating the expiration logic works correctly.
- The line *Ran 1 test in 0.934s OK* confirms the test executed fully with no errors or exceptions.

The system successfully enforces the 24-hour Token expiration rule. It validates Tokens immediately after generation and correctly invalidates them after the configured timeout (24-hours), ensuring strong time-based security control.

### **Brute Force Attack Simulation Tests and Results**

A simulation was created and run to test whether the system can prevent a brute force attack originating from a single IP address. The command used: *python manage.py simulate\_bruteforce --username=testuser --attempts=10 --delay=0.5* Below is the code snippet.

```

8 from axes.handlers.proxy import AxesProxyHandler
9 import logging
10
11 class Command(BaseCommand):
12     help = 'simulate brute force attack with detailed lockout status and features'
13
14     def add_arguments(self, parser):
15         # Define command line arguments for the management command
16         parser.add_argument('--username', type=str, help='Username to target', default='testuser')
17         parser.add_argument('--attempts', type=int, help='Number of login attempts', default=10)
18         parser.add_argument('--delay', type=float, help='Delay between attempts in seconds', default=0.5)
19         parser.add_argument('--ip', type=str, help='IP address to simulate from', default='127.0.0.1')
20         parser.add_argument('--simulate-success-after-lockout', action='store_true', help='Simulate a successful login after lockout expires')
21
22     def handle(self, *args, **options):
23         # Extract options from command line arguments
24         username = options['username']
25         attempts = options['attempts']
26         delay = options['delay']
27         ip = options['ip']
28         simulate_success_after_lockout = options['simulate_success_after_lockout']
29
30         self.stdout.write(f"Starting brute force simulation for user '{username}' from IP {ip} with {attempts} attempts.")
31
32         # Get the user model and fetch the target user
33         user = get_user_model()
34         try:
35             user = User.objects.get(username=username)
36         except User.DoesNotExist:
37             self.stdout.write(f"User '{username}' does not exist.")
38             return
39
40         # Setup request factory and logger
41         factory = RequestFactory()
42         logger = logging.getLogger(__name__)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS (PREVIEW) AZURE

powerShell + - ... ^ x

IP 127.0.0.1 is NOT locked out after attempt 2.  
Lockout triggered at attempt 3 after 6.76 seconds.

Figure 47: simulate\_bruteforce code

The simulation tested 10 login attempts from a single IP address (127.0.0.1) with a 0.5-second delay between attempts. Below are the results/output.



Key points from the output/results:

- The system successfully triggered a lockout at attempt 3, after 6.76 seconds.
- All subsequent attempts (4–10) were correctly blocked, confirming that the account remained locked out.
- Multiple lockout confirmation messages indicate that the account remained locked during repeated post-lockout attempts.
- The total time before final lockout confirmation was 18.68 seconds.

The system effectively enforced a lockout after repeated failed login attempts, demonstrating resistance against brute force attacks from a single IP address.

### **Distributed Brute Force Attack Simulation Tests and Results**

A simulation was created and run to test whether the system can prevent a brute force attack originating from multiple IP addresses simultaneously. The command used:

```
python manage.py simulate_distributed_bruteforce --username testuser --attempts 10 --delay 0.5 --ip-list 127.0.0.1 127.0.0.2 127.0.0.3
```

The system processed attack attempts from three IPs: 127.0.0.1, 127.0.0.2, and 127.0.0.3. Below are the code snippet and the outputted results.

```

login > management > commands > simulate_distributed_bruteforce.py
8 from axes.handlers.proxy import AxesProxyHandler
9 import logging
10
11 class Command(BaseCommand):
12     help = 'simulate distributed brute force attack from multiple IP addresses concurrently'
13
14     def add_arguments(self, parser):
15         parser.add_argument('--username', type=str, help='Username to target', default='testuser')
16         parser.add_argument('--attempts', type=int, help='Number of login attempts per IP', default=10)
17         parser.add_argument('--delay', type=float, help='Delay between attempts in seconds', default=0.5)
18         parser.add_argument('--ip-list', nargs='+', type=str, required=True, help='List of IP addresses to simulate from')
19         parser.add_argument('--simulate-success-after-lockout', action='store_true', help='Simulate a successful login after lockout expires')
20
21     def handle(self, *args, **options):
22         username = options['username']
23         attempts = options['attempts']
24         delay = options['delay']
25         ip_list = options['ip_list']
26         simulate_success_after_lockout = options['simulate_success_after_lockout']
27
28         self.stdout.write(f"Starting distributed brute force simulation for user '{username}' from IPs: {ip_list} with {attempts} attempts each.")
29
30         User = get_user_model()
31         try:
32             user = User.objects.get(username=username)
33         except User.DoesNotExist:
34             self.stdout.write(f"User '{username}' does not exist.")
35             return
36
37         factory = RequestFactory()
38         logger = logging.getLogger(__name__)
39         proxy_handler = AxesProxyHandler()
40
41         lockout_triggered = threading.Event()
42         lockout_info = {'attempt': None, 'ip': None}

```

Figure 50: simulate\_distributed\_bruteforce code

```

python manage.py simulate_distributed_bruteforce --username testuser --attempts 10 --delay 0.5 --ip-list 127.0.0.1 127.0.0.2 127.0.0.35SEX\Thesis\Assignments\Project\src>
Brute force simulation completed.
System check identified some issues:

WARNINGS:
? cked
Attempt 1 from IP 127.0.0.2: Authentication error
IP 127.0.0.1 is NOT locked out after attempt 1.
Lockout triggered by IP 127.0.0.2 at attempt 2.
IP 127.0.0.2 is currently LOCKED OUT after attempt 2.
IP 127.0.0.3 is currently LOCKED OUT after attempt 2.
IP 127.0.0.1 is currently LOCKED OUT after attempt 2.
IP 127.0.0.2 is currently LOCKED OUT after attempt 3.
IP 127.0.0.3 is currently LOCKED OUT after attempt 3.
IP 127.0.0.1 is currently LOCKED OUT after attempt 3.
IP 127.0.0.2 is currently LOCKED OUT after attempt 4.
IP 127.0.0.3 is currently LOCKED OUT after attempt 4.
IP 127.0.0.1 is currently LOCKED OUT after attempt 4.
IP 127.0.0.2 is currently LOCKED OUT after attempt 5.
IP 127.0.0.3 is currently LOCKED OUT after attempt 5.
IP 127.0.0.1 is currently LOCKED OUT after attempt 5.
IP 127.0.0.2 is currently LOCKED OUT after attempt 6.
IP 127.0.0.3 is currently LOCKED OUT after attempt 6.
IP 127.0.0.1 is currently LOCKED OUT after attempt 6.
IP 127.0.0.2 is currently LOCKED OUT after attempt 8.
IP 127.0.0.3 is currently LOCKED OUT after attempt 7.
? cked
Attempt 1 from IP 127.0.0.2: Authentication error
IP 127.0.0.1 is NOT locked out after attempt 1.
Lockout triggered by IP 127.0.0.2 at attempt 2.
IP 127.0.0.2 is currently LOCKED OUT after attempt 2.
IP 127.0.0.3 is currently LOCKED OUT after attempt 2.
IP 127.0.0.1 is currently LOCKED OUT after attempt 2.
IP 127.0.0.2 is currently LOCKED OUT after attempt 3.
IP 127.0.0.3 is currently LOCKED OUT after attempt 3.
IP 127.0.0.1 is currently LOCKED OUT after attempt 3.

```

Figure 51: simulate\_distributed\_bruteforce result1

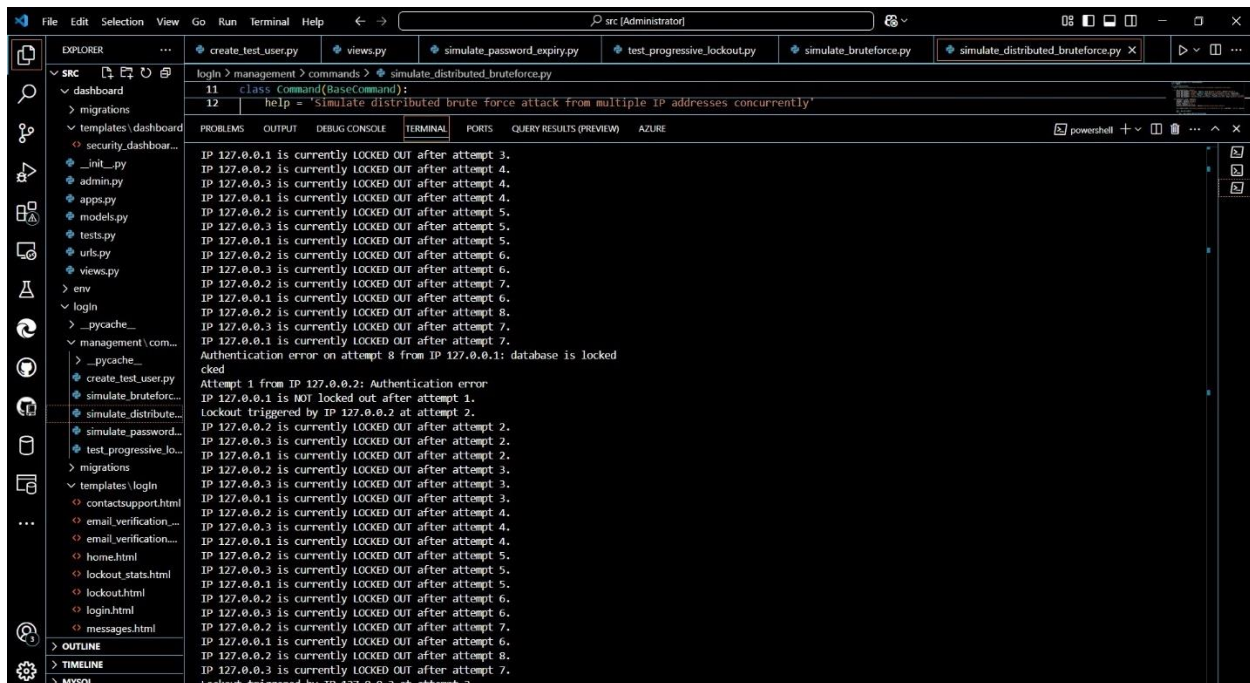


Figure 52: simulate\_distributed\_bruteforce result2

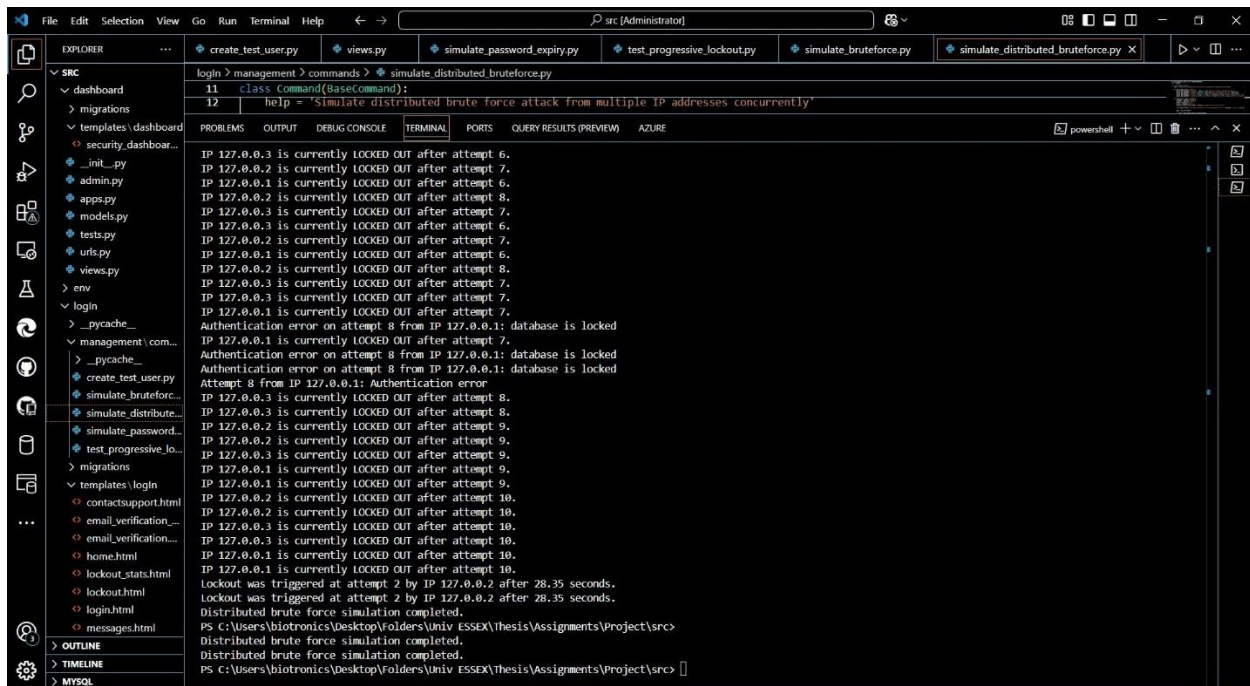


Figure 53: simulate\_distributed\_bruteforce result3

Key points from the output/results:

- Lockout was triggered early by IP 127.0.0.2 on the 2nd attempt, effectively locking all listed IPs out immediately.
- After lockout, subsequent attempts from all IPs were blocked, indicating that the lockout mechanism works globally across distributed sources.
- Some authentication errors occurred (e.g., "database is locked"), which suggests concurrent write contention — a typical issue in SQLite under parallel operations.
- The total time to trigger and confirm lockout across all sources was 28.35 seconds.

The system demonstrated the ability to detect and block a coordinated brute-force attack from multiple IP addresses, enforcing a global account lockout policy. This confirms the effectiveness of the distributed brute-force mitigation strategy.

### **Concurrent Session Test and Results**

A test was created and ran to test whether the system prevents concurrent session. The command used: `pytest login/tests/test_concurrent_session.py --disable-warnings -q`

From the output it showed that it ran successfully and concurrent session is prevented.

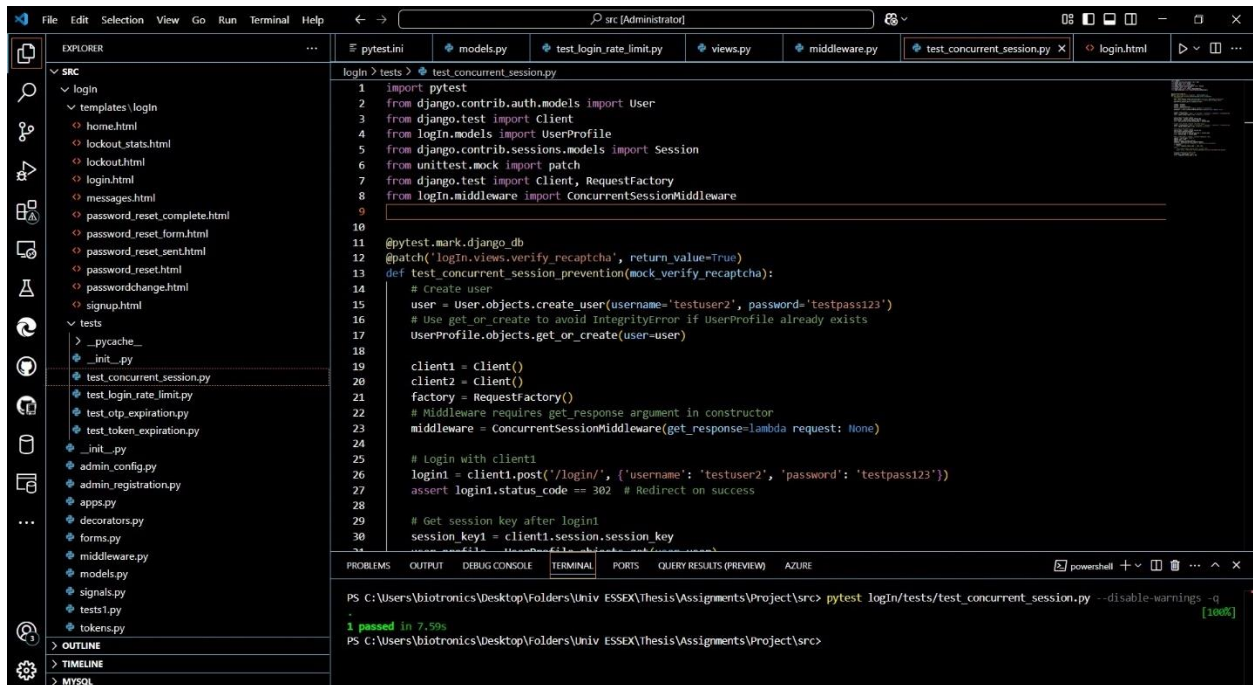


Figure 54: Concurrent session test and results

## pytest and Results

pytest was ran to test all the tests of the system. The command used: `pytest`

All the tests were succesfull except for the ratelimit one because it was been blocked by the ratelimit decorator in the login view. Thus suggesting that the block works.

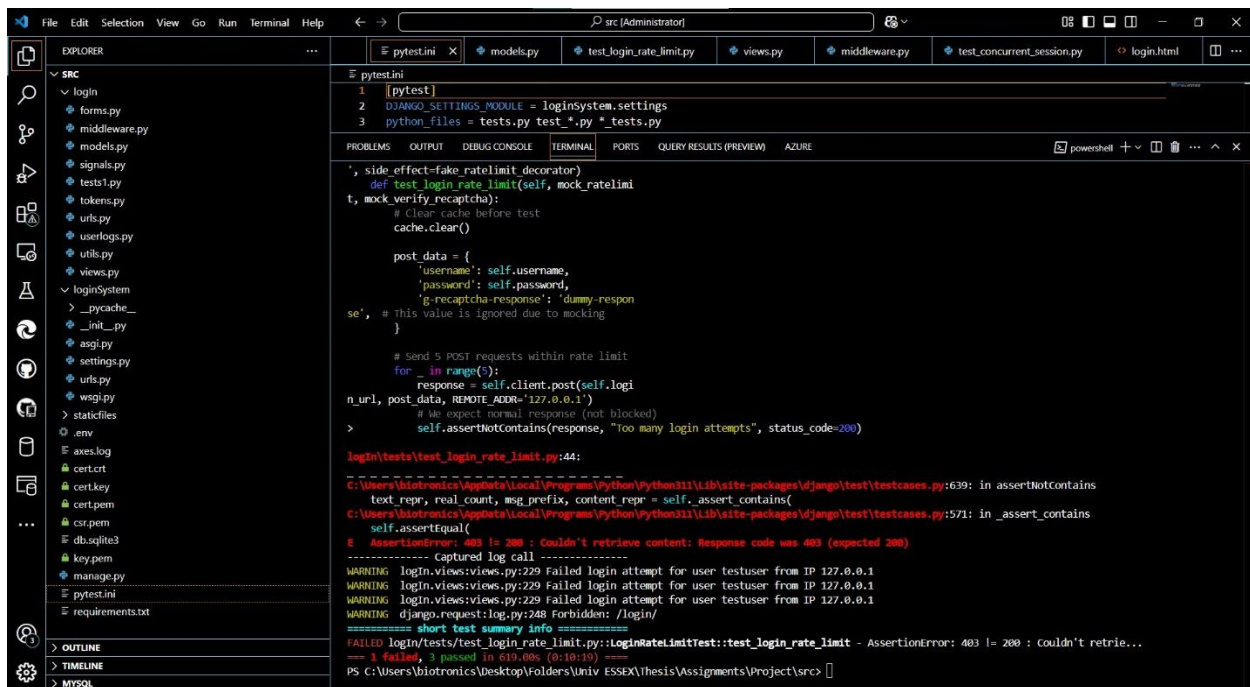


Figure 55: pytest results

## Appendix E: Dashboard Analytics and Lockout stats Logs Test

Dashboard Analytics View

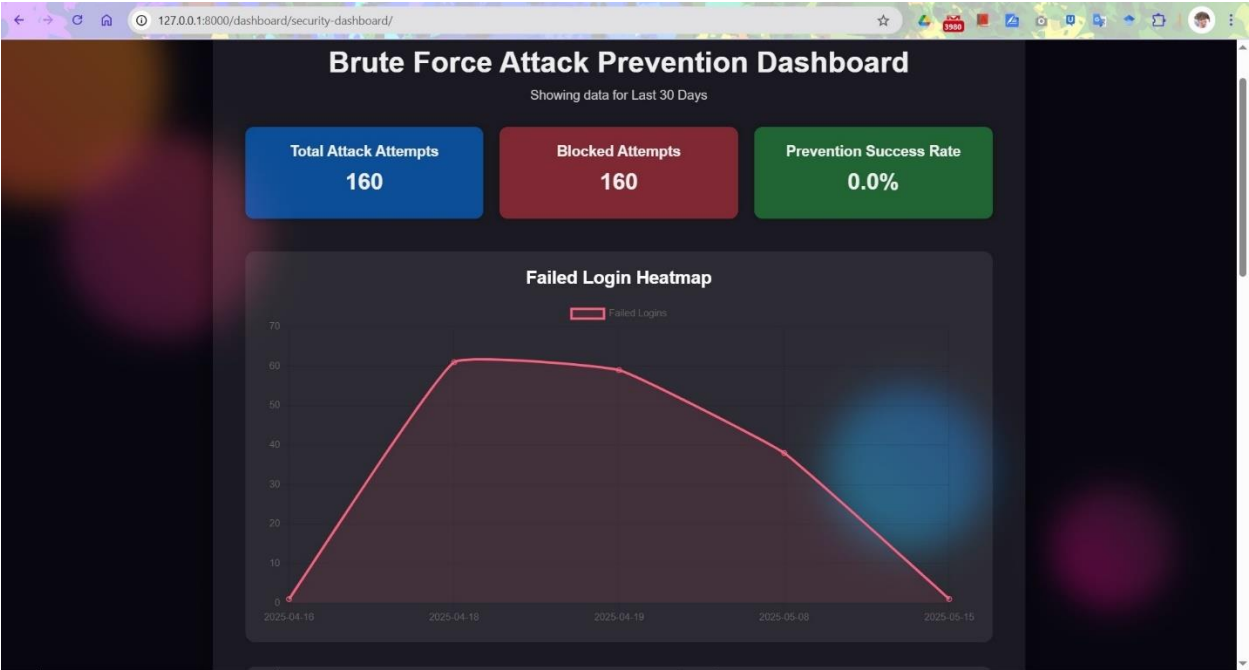


Figure 56: Security Dashboard1

The table lists details for IP threats, including IP Address, Attempts, Last Attempt, Location, Operating System, Device Type, and User Agent. It also includes navigation buttons for Home and Logout, and a copyright notice.

IP Address	Attempts	Last Attempt	Location	Operating System	Device Type	User Agent
127.0.0.1	122	2025-05-15 09:11:00	Unknown Location	Unknown OS	Desktop	Unknown
127.0.0.3	19	2025-05-08 01:15:07	Unknown Location	Unknown OS	Desktop	Unknown
127.0.0.2	19	2025-05-08 01:15:07	Unknown Location	Unknown OS	Desktop	Unknown

Home Logout

© 2025. All rights reserved.

Figure 57: Security Dashboard3