Developing A Password Management System With A "Shift-Left" Approach To Security

NIVEDHIDHA ILANGOVAN, Hochschule Offenburg, Germany

This document comprehensively outlines the secure software development lifecycle (SSDLC) followed in the creation of a custom Password Management System (PMS) for ACME Corporation. A "shift-left" security approach is emphasized throughout the entire process, integrating security considerations from the earliest stages of development. The PMS is designed to generate passwords that adhere to dynamically changeable policies, securely store the data required for password validation, and interface with external services, such as the Have I Been Pwned (HIBP) database, to check for compromised credentials. This report meticulously details each phase of the SSDLC, starting with the methods employed for requirements gathering and analysis, progressing through the C4 and UML-based design, which includes a thorough threat analysis. The design is then translated into a prototypical Python implementation, demonstrating the functionality and security considerations in practice. The implementation is rigorously validated through comprehensive unit tests, ensuring the correctness and robustness of individual components. Furthermore, static code analysis is performed to proactively identify potential security vulnerabilities and code quality issues. The document presents a detailed analysis and breakdown of the requirements, design artifacts (including diagrams and models), the Python code itself, and the results of the comprehensive testing and analysis. This holistic presentation aims to demonstrate the robust and secure development of the PMS, highlighting the focus on quality, security, and adherence to best practices.

ACM Reference Format:
Nivedhidha Ilangovan. 2025. Developing A Password Management System With A "Shift-Left" Approach To Security. *J. ACM* 1, 1, Article 1 (February 2025), 18 pages. https://doi.org/XXXXXXX.XXXXXXX

Author's Contact Information: Nivedhidha Ilangovan, nilangov@stud.hs-offenburg.de, Hochschule Offenburg, Offenburg, Baden-Württemberg, Germany.

ACM 1557-735X/2025/2-ART1
https://doi.org/XXXXXXX.XXXXXXX

Contents

1 INTRODUCTION

ACME Corporation has allocated a budget of 800,000 euros for the development of a custom Password Management System (PMS). The core objective of this initiative is to create a robust and secure solution for generating, storing, and managing passwords across the organization's diverse range of internal applications. The PMS must adhere to stringent FIPS (Federal Information Processing Standards) guidelines, ensuring compliance with industry best practices for data security. Furthermore, the system must support dynamic password policies, allowing for flexible and adaptable security configurations that can be adjusted as needed. A paramount concern is ensuring the strength of generated passwords, making them resistant to cracking attempts. Critically, the PMS must be designed and implemented with security as a primary focus, minimizing the risk of data breaches and protecting sensitive information. This comprehensive approach to password management will provide ACME Corporation with a secure and reliable solution for safeguarding access to its internal systems.

2 SOFTWARE REQUIREMENTS

2.1 USER STORIES

Example User Story: As a user, I want the PMS to generate strong, unique passwords that meet the specific requirements of each application, so that I can maintain security for my accounts.

Description of the starting situation: The user needs to create a secure password for an application and is unaware of the specific password policy of that application.

Normal flow of events:

(1) In the PMS, the user selects the desired application for which a password needs to be generated.
(2) The PMS retrieves the specific password policy associated with that application.
(3) A strong, unique password is generated by the PMS, adhering to the retrieved policy.
(4) The generated password is then cross-referenced against a database of known leaked passwords.
(5) If a match is found, the PMS generates a new password.
(6) Once a secure password is generated, it is securely delivered to the user.
(7) The user can then employ this password to log into the designated application.

What can go wrong: The user may forget the generated password. The application's password policy could be outdated. The database of leaked passwords may not always be entirely up-to-date, potentially leading to the generation of a compromised password.

Concurrent activities: Other users may be requesting password generation for different applications or System Administrators may be updating password policies.

Ending state: The user has a strong, unique password for the chosen application.

2.2 USE CASES

Actors:

- **User:** This actor represents the end-user who interacts with the PMS to generate, manage, and use passwords for various applications.
- **System Administrator:** This actor is responsible for managing the overall system, including defining policies, generating bulk passwords, and monitoring system health.
- **Legacy Applications:** These represent the existing applications that integrate with the PMS to obtain and validate user credentials.
- **PMS:** This is the system itself, which acts as a mediator between the users and the legacy applications. It handles password generation, storage, and authentication.

Usecase:

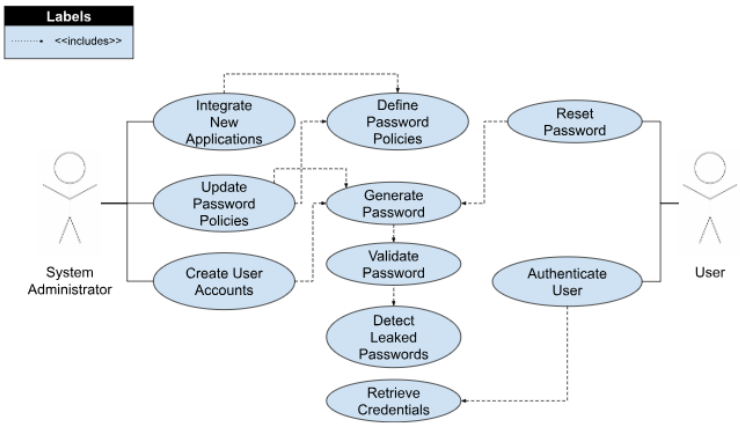

Fig. 1. Usecase Diagram

- **Integrate New Applications:** This feature is used by the System Administrator to add a new application to the PMS.
- **Update Password Policies:** This feature is used by the System Administrator to modify the password policy for an existing application connected to the PMS.
- **Create User Accounts:** This feature is used by the System Administrator to create new user accounts within a specific application.
- **Define Password Policies:** This feature is used by the PMS when a System Administrator adds a new application or updates the password policy of an existing one.
- **Generate Password:** This feature is used by the PMS whenever a new password is required, such as during a user-initiated password reset, a System Administrator's user account creation, or a password policy update affecting existing accounts.
- **Validate Password:** This feature is used by the PMS during password generation to ensure the new password complies with the selected application's current password policy.
- **Detect Leaked Passwords:** This feature is used by the PMS when a user creates a new password or the PMS generates one to check if the password has been previously compromised and leaked online.
- **Retrieve Credentials:** This feature is used by the PMS during authentication to retrieve stored credentials for comparison with user-provided values.
- **Reset Password:** This feature is used by the User to change the existing password for a specific application.
- **Authenticate User:** This feature is used by the User to authenticate on an application's login page.

2.3 ABUSE / MISUSE CASES

Understanding the potential vulnerabilities of a system requires examining not only its intended functionality but also how it might be misused or abused. Abuse cases represent specific sequences of actions that could lead to harm, damage, or loss, often focusing on unintentional or accidental misuse. They explore scenarios where legitimate functionality is employed in unintended ways, resulting in negative consequences. Misuse cases, on the other hand, describe intentional violations

of the system's security policies or intended behavior by a malicious actor or adversary. They concentrate on deliberate attempts to exploit weaknesses, bypass controls, or gain unauthorized access. By considering both abuse and misuse cases, developers can gain a more comprehensive understanding of the system's threat landscape and design more resilient and secure systems. This proactive approach to identifying potential vulnerabilities strengthens the system's defenses against both accidental misuse and malicious attacks.

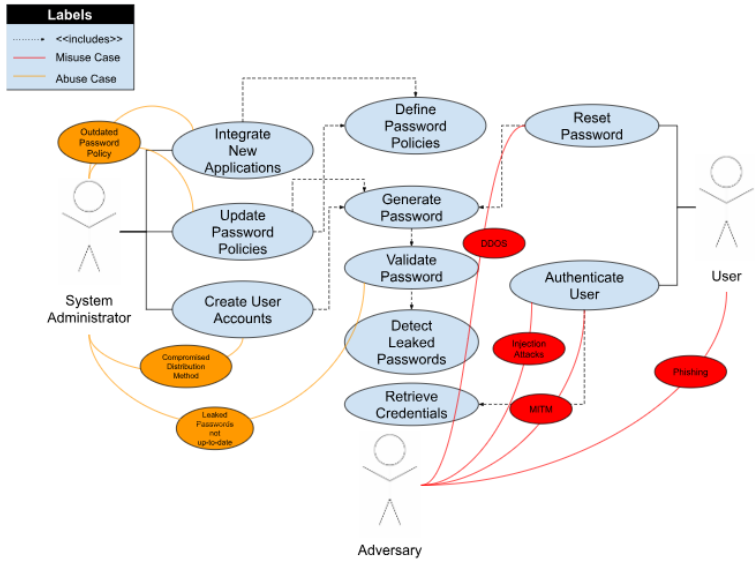

Fig. 2. Abuse/Misuse Case Diagram

Abuse Cases:

- **Outdated Password Policy:** The system administrator might fail to update password policies as per current recommendations.
- **Leaked Passwords not up-to-date:** The system administrator might fail to update the leaked passwords database.
- **Compromised Distribution Method:** The system administrator might use an unsafe/compromised distribution method for transmitting bulk-generated passwords to users.

Misuse Cases:

- **DDOS:** The adversary can send multiple requests for password reset making the PMS flooded with requests and unable to respond to legitimate traffic.
- **Injection Attacks:** The adversary can use injection attacks like SQL injection to retrieve all the passwords from the database.
- **MITM:** The adversary can try to intercept and steal the passwords being submitted during the user authentication. It becomes easier to decrypt the passwords when the encryption methods used are weak.
- **Phishing:** The adversary can send a phishing email to the users stating that a password update needs to be performed along with a malicious URL for a password reset link.

2.4 ATTACK TREES

Attacks Based on Availability:

This diagram outlines potential threats to system availability, concentrating on the database, web server, and network infrastructure as critical points of vulnerability. It shows how various exploits could compromise these essential components and lead to system downtime. The high risk associated with system unavailability underscores the necessity of robust security measures, including redundancy, failover mechanisms, and intrusion detection systems, to mitigate these diverse threats and ensure the continuous operation of the PMS. These measures are crucial for protecting the PMS against denial-of-service attacks, infrastructure failures, and other events that could disrupt service.

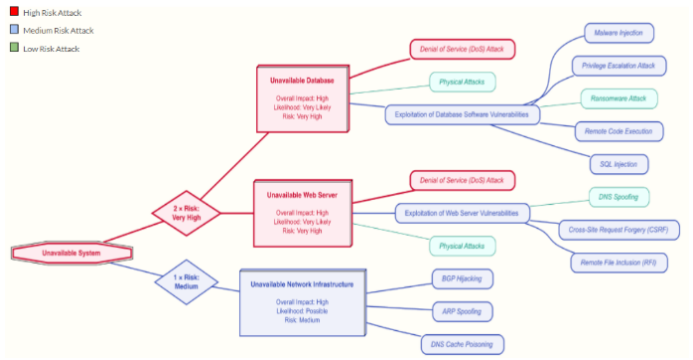

Fig. 3. Attack Tree based on Availability

Attacks Based on Confidentiality and Integrity:

This diagram summarizes potential threats to the system's confidentiality and integrity, with a primary focus on the database as a key target for attacks aimed at password compromise. The high risk associated with password compromise is evident, emphasizing the need for robust security measures such as strong password policies, multi-factor authentication, and regular security awareness training. These measures are essential to mitigate the diverse threats outlined in the diagram and protect the PMS from unauthorized access, data breaches, and other attacks that could compromise user credentials and sensitive data.

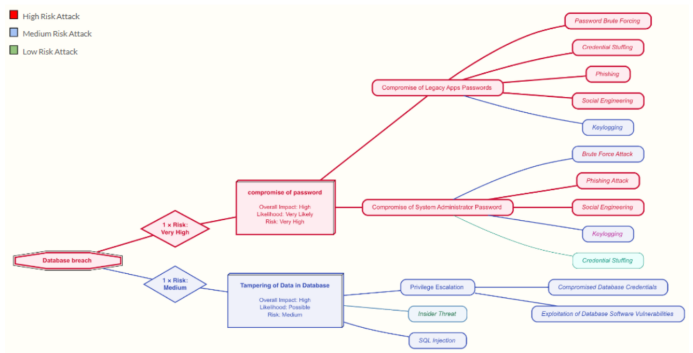

Fig. 4. Attack Tree based on Confidentiality and Integrity

2.5 REQUIREMENTS TABLE

ID	Type	Description	Related Reqs	Unit Test ID	Comments
SR 4	Non-Functional	Avoid leaked passwords by storing them securely and comparing them to leaked lists	SR 7	UT-001	Consider using bcrypt
SR 4.1	Non-Functional	Define and store comparison lists	SR 4	UT-002	Make updatable
SR 4.2	Non-Functional	Define hashing type and comparison system	SR 4, SR 4.1	UT-003	Allow for hash alg. updates

Table 1. Example for Requirements Splitting

Requirement SR 4, "Avoid leaked passwords by storing them securely and comparing them to leaked lists," mandates the protection of user passwords from unauthorized access. To achieve this overarching goal, SR 4 is broken down into more granular sub-requirements. SR 4.1, "Define and store comparison lists," dictates the creation and secure maintenance of databases of known compromised passwords, which are crucial for identifying potentially leaked credentials. Complementing this, SR 4.2, "Define hashing type and comparison system," specifies the implementation of a robust mechanism for securely hashing stored passwords and efficiently comparing them against the lists of leaked passwords. This structured decomposition of SR 4 ensures a systematic and thorough approach to password security, guaranteeing both the secure storage of credentials and their effective validation against known compromises.

3 SOFTWARE DESIGN

3.1 THREAT MODELING ON THE BASIS OF C4 AND STRIDE

STRIDE provides a systematic framework for identifying potential security vulnerabilities within software systems, enabling early detection of risks during the design phase of the Software Development Life Cycle (SDLC). By decomposing a system into its constituent components and applying the STRIDE mnemonic (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to each, analysts can methodically uncover potential threats. This structured approach has been widely adopted by threat modeling tools to streamline and enhance the threat identification process.

Complementing STRIDE, the C4 model offers a layered perspective for understanding complex software systems. It enhances threat modeling techniques by providing clear visual representations of system components, their relationships, and the flow of data. This visualization clarifies dependencies and interactions, which are crucial for identifying potential security vulnerabilities. The combined approach leverages the strengths of both methodologies, proceeding as follows:

(1) Construct a C4 model, progressing from the high-level context (C1) down to the detailed component level (C3).
(2) Apply the STRIDE framework to each layer of the C4 model, systematically analyzing potential threats at different levels of abstraction.
(3) Thoroughly document the identified threats and prioritize them based on their potential impact and likelihood of occurrence.

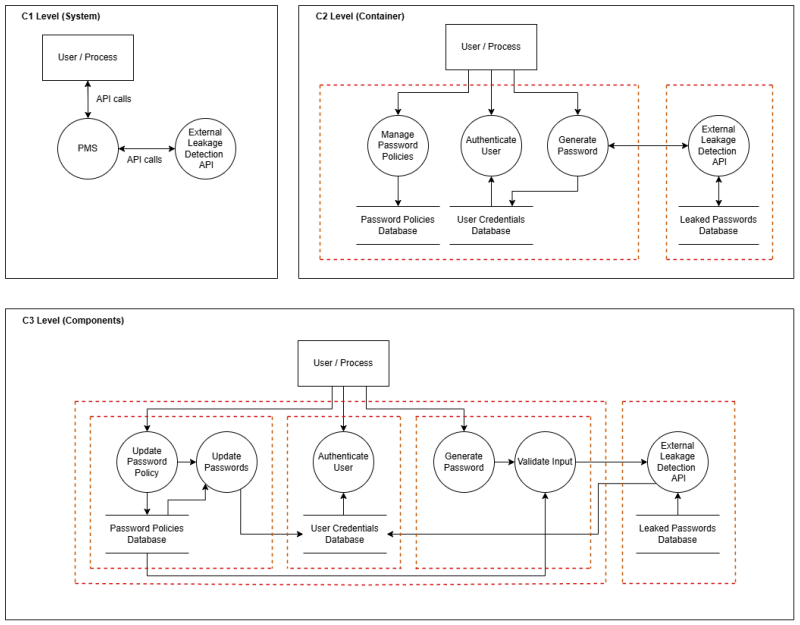

Fig. 5. C4 Diagram (C1 to C3)

S. No.	Component	STRIDE Category	Threat
1	User Authentication	Spoofed Identity	Failure of user interface to obscure entry of credentials
5	User Authentication	Tampering with Input	It is possible for attacker to tamper with cookies
9	Overall PMS	Repudiation of Action	Lack of logging showing user access to sensitive data
14	User Credentials Database	Information Disclosure	Secrets are stored in plain text and/or stored in unencrypted storage
19	User Authentication	Denial of Service	System was not designed to meet current traffic demands

Table 2. Example for Threat Analysis using C4 with STRIDE

3.2 THREAT MODELING USING IRIUS RISK TOOL

IRIUS Risk is a threat modeling tool designed to simplify and accelerate the process of identifying potential security vulnerabilities within software systems. By incorporating established threat modeling methodologies such as STRIDE, IRIUS Risk guides analysts through a structured and systematic approach to threat identification. This structured approach ensures a comprehensive analysis of the system, helping to uncover potential weaknesses early in the software development lifecycle when they are less costly and easier to address. IRIUS Risk often provides features like visual diagrams, threat libraries, and reporting capabilities to further aid in the threat modeling process.

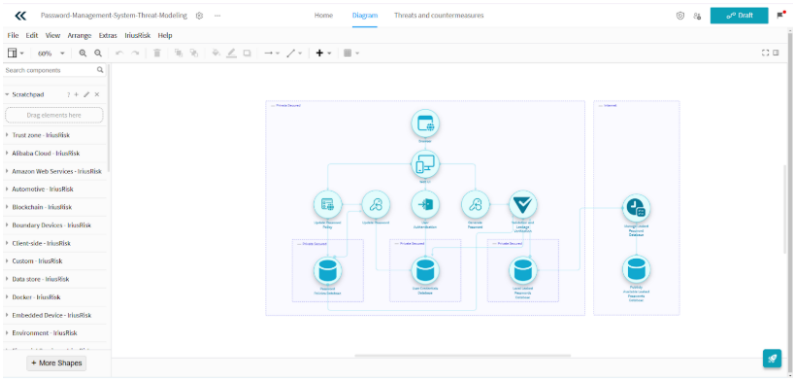

Fig. 6. Evidence for using IRIUS RISK Tool

S. No.	Component	STRIDE Category	Threat
2	Browser	Tampering	An adversary embeds malicious scripts in content that will be served to web browsers
3	Browser	Spoofing	An adversary may craft messages that appear to come from a different principle or use stolen / spoofed authentication credentials
5	User Credentials Database	Information Disclosure	An attacker examines a target system to find sensitive data that has been embedded within it
22	User Authentication	Elevation of Privilege	Attackers exploit flaws in access control systems
24	Update Password Policy	Denial of Service	Attackers flood the form with requests or exploit a vulnerable data processing functionality

Table 3. Example for Threat Analysis using IRIUS RISK Tool

3.3 THREAT MODELING USING MICROSOFT THREAT MODELING TOOL

Microsoft's Threat Modeling Tool is a robust solution for organizations seeking to proactively address security vulnerabilities in their software. Utilizing proven methodologies such as STRIDE, the tool facilitates a structured and systematic approach to threat identification, enabling early detection of potential risks within the software development lifecycle. Its seamless integration with the Microsoft development environment makes it a particularly attractive option for organizations heavily invested in Microsoft technologies. The tool empowers security analysts and developers to visualize potential threats, analyze their impact, and prioritize mitigation efforts. By identifying and addressing security concerns early, organizations can reduce the cost and complexity of fixing vulnerabilities later in the development process. This proactive approach to security helps build more resilient and secure software systems.

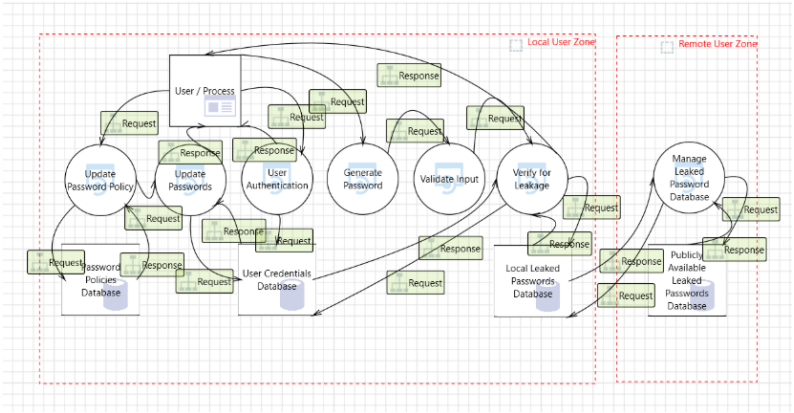

Fig. 7. Evidence for using Microsoft Threat Modeling Tool

S. No.	Components	Interaction	STRIDE Category	Threat
1	User/Process, Update Password Policy	Request	Elevation of Privileges	An adversary may gain unauthorized access to Web API due to poor access control checks
19	Update Password Policy, Update Passwords	Request	Information Disclosure	An adversary can gain access to sensitive information from an API through error messages
41	User/Process, User Authentication	Request	Tampering	An adversary may inject malicious inputs into an API and affect downstream processes
57	User/Process, Generate Password	Request	Spoofing	An adversary may spoof User / Process and gain access to Web API
101	Manage Leaked Password Database, Local Leaked Passwords Database	Request	Repudiation	An adversary can deny actions on database due to lack of auditing

Table 4. Example for Threat Analysis using Microsoft Threat Modeling Tool

3.4 MAIN FINDINGS AND RECOMMENDATIONS

The Password Management System (PMS), while designed with core security functionalities, faces several potential vulnerabilities that could compromise its integrity and the confidentiality of sensitive user data. A primary concern is the potential lack of robust authentication and authorization mechanisms. Without these, unauthorized access becomes a significant risk, allowing malicious actors to bypass security measures and gain control of the system. Furthermore, insecure data handling practices, particularly during storage, transmission, and processing, could expose sensitive information, such as password hashes and user credentials, to data breaches. Insufficient input validation and sanitization mechanisms present another critical weakness, leaving the system susceptible to various injection attacks like SQL injection or cross-site scripting. These attacks

could allow adversaries to manipulate data, execute malicious code, or gain unauthorized access. The absence of limited logging and monitoring capabilities would severely hamper the system's ability to detect and respond to threats effectively. Without proper logging, security incidents might go unnoticed, and forensic analysis would be significantly more difficult. Finally, the reliance on any custom-coded security functions introduces a potential point of failure. If these functions are not implemented correctly, they themselves could introduce new vulnerabilities, undermining the overall security posture of the PMS.

To proactively mitigate these identified risks and ensure the robust security of the PMS, a comprehensive, multi-layered security approach is strongly recommended. This approach should begin with a thorough review and strengthening of authentication and authorization protocols. Implementing role-based access control (RBAC) is crucial to limit user privileges to the minimum necessary for their assigned roles. Regular reviews and updates of access permissions are essential to adapt to changing needs and minimize the risk of privilege creep. Robust access controls should be implemented to protect sensitive data and critical systems from unauthorized access. Data handling practices must be secured through the implementation of encryption for sensitive data both at rest and in transit. Utilizing HTTPS for all communication channels is paramount to protect data during transmission. Secure session management practices, including appropriate session timeouts and secure session cookies, are necessary to prevent session hijacking. Regular backups of critical data, coupled with regular testing of the restore process, are vital for business continuity in the event of data loss or system compromise. Input validation and sanitization mechanisms must be significantly enhanced to prevent injection attacks. Implementing robust input validation techniques and using a web application firewall (WAF) to filter malicious traffic can significantly reduce the risk of these attacks. Comprehensive logging for all critical system activities is essential for security monitoring and incident response. Integrating these logs with a security information and event management (SIEM) tool will enable effective correlation and analysis of security events. Finally, secure configuration practices and tools should be employed to harden all systems and applications. Automating configuration tasks using security configuration management tools can help ensure consistent and secure configurations across the environment. By implementing these recommendations, ACME Corporation can significantly improve the security posture of the PMS and protect its sensitive data assets.

4 SOFTWARE CODING

4.1 KEY FUNCTIONALITIES

- **new app password policy:** Allows users to create and store new password policies for different applications. It takes the application name and desired password characteristics as input and stores this information in a JSON file (passPolicyDb.json).
- **change password policy:** Enables users to update existing password policies for applications. It takes the application ID and updated policy parameters as input and modifies the corresponding policy in the passPolicyDb.json file.
- **generate password:** Generates a strong password based on the specified application's policy and stores its hash value in passDb.json. It checks the passPolicyDb.json for the relevant policy, generates a password that meets the criteria, and checks it against the Have I Been Pwned (HIBP) database to avoid compromised passwords.
- **verify password:** Verifies user login credentials. It takes the user ID, application ID, and password as input. It retrieves the hashed password for the given user and application from the passDb.json file and compares it with the provided password using a secure hashing algorithm (Argon2).

4.2 DATA STORAGE

The system utilizes two JSON files for data storage:

- **passPolicyDb.json:** Stores password policies for different applications, including rules for password length, character requirements, and other constraints.
- **passDb.json:** Stores login information, including user ID, application ID, and the hashed password.

4.3 UML DESIGN

Admin Updates Password Policy

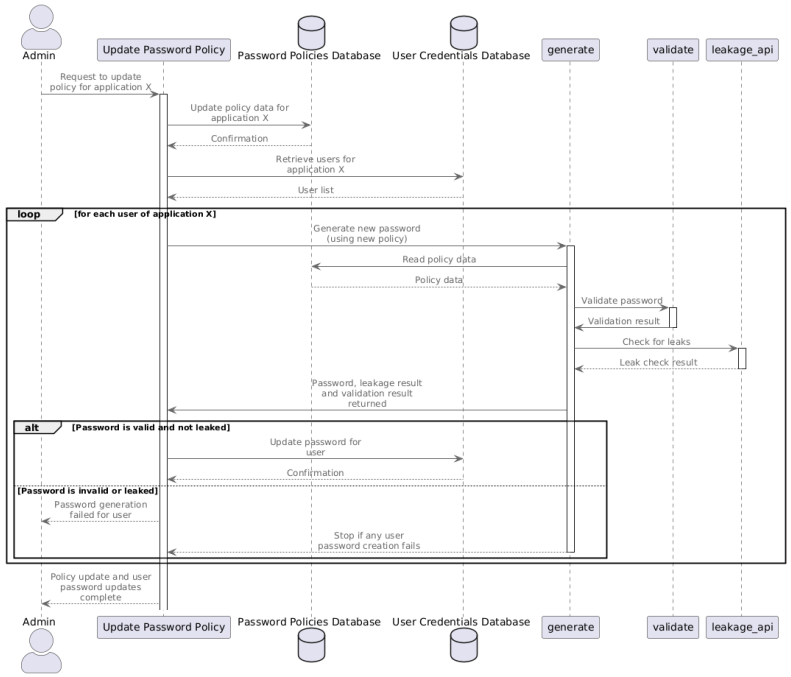

Fig. 8. Sequence Diagram for Password Policy Update

This sequence diagram illustrates the detailed steps involved in an administrator updating a password policy for a specific application within the Password Management System (PMS), from the initial request to the final confirmation. The administrator's update request triggers the retrieval of the new policy from the Password Policies Database, which is then applied iteratively to each user of the application. For every user, the system generates a policy-compliant password, validates it against the new rules and (optionally) a leak check API, and securely stores the new password in the User Credentials Database if it passes validation. The diagram emphasizes the iterative nature of updating each user's password and the error handling mechanism that halts the process for a specific user if password generation or validation fails, ensuring only valid and secure passwords are stored. This entire process culminates in a confirmation message to the administrator upon successful completion of the policy and password updates. The diagram also highlights the interaction with external databases and APIs, showcasing the system's reliance on these components for policy enforcement, user data, and security checks, ultimately ensuring the secure and consistent application of password policies across the user base.

User Authentication

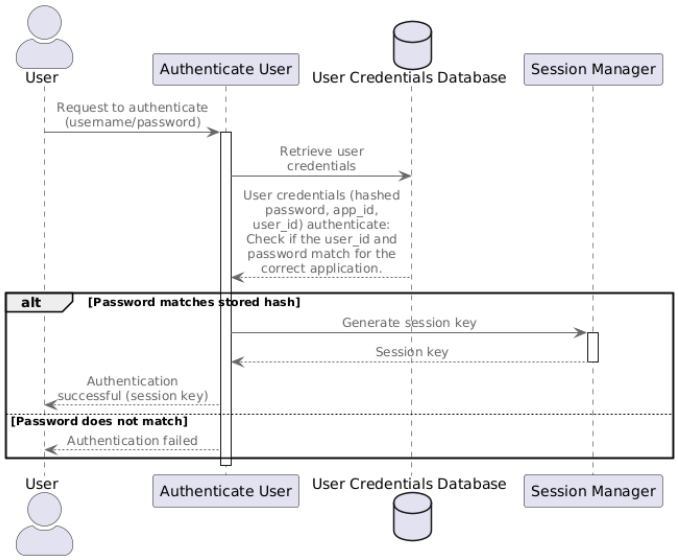

Fig. 9. Sequence Diagram for Authentication

This sequence diagram illustrates the user authentication flow within the Password Management System (PMS). A user attempting to access a protected application submits their username and password, triggering the PMS authentication process. The PMS retrieves the corresponding stored password hash and salt from the User Credentials Database. It then hashes the user's provided password using the retrieved salt and compares it to the stored hash. Upon a successful match, the PMS generates a unique session key, which is returned to the application, granting the user access. Conversely, a password mismatch results in authentication failure, and the user is notified. This process ensures password security by handling only hashes, not plaintext passwords, and utilizes session keys for secure, convenient access.

User Requests Password Reset

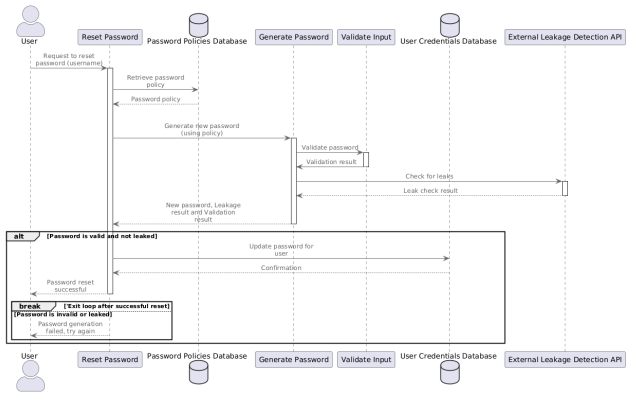

Fig. 10. Sequence Diagram for Password Reset

This sequence diagram illustrates the interactions between a user and the core components of a system, specifically focusing on password-related operations. The diagram depicts a user interacting with several key functionalities: updating the password policy, updating user passwords, authenticating, generating a new password, and validating input. The diagram also highlights the system's reliance on external databases, namely the Password Policies Database and the User Credentials Database, for storing and retrieving critical information. Furthermore, it shows the interaction with an external system, the External Leakage Detection API, which is likely used for security checks, such as verifying if a password has been compromised. The diagram visually represents the flow of requests and responses between the user and these various components, providing a clear understanding of how these functionalities interact with each other and with the underlying data stores and external security services. The diagram's structure, with the user initiating actions and the components responding, emphasizes the user-centric nature of these password-related operations.

4.4 DISCUSSION ON THE SECURE STORAGE OF PASSWORDS AND PASSWORD VALIDATION

In this section, the secure storage of passwords is discussed and their validation, crucial aspects for the integrity and confidentiality of the Password Management System (PMS). A robust PMS must protect user credentials from unauthorized access, both internally and externally.

4.4.1 Secure Password Storage: It is recommended to adhere to the industry standard practice of using a one-way hashing function to transform passwords into hashes. These hashes, rather than plain text passwords, should be stored in the database. Specifically, it is suggested to use stronger hashing algorithms like bcrypt, Argon2, or scrypt. These algorithms incorporate "salting"—adding a unique random string to each password before hashing—which further enhances security by hindering attackers from using pre-computed hash tables. This salting process makes each password hash unique, even if users choose the same password, further complicating cracking attempts.

Furthermore, it is emphasized that the storage mechanism itself must be secured. Access to the password database should be strictly limited, following the principle of least privilege. Data at rest should be encrypted, and we recommend conducting regular security audits to ensure ongoing protection. These audits should include penetration testing to actively search for vulnerabilities and ensure the effectiveness of the implemented security measures.

4.4.2 Password Validation: For user login attempts, the entered password must be validated against the stored hash. This should be done by hashing the entered password using the same algorithm and salt used during storage. The resulting hash is then compared to the stored hash. A match authenticates the user.

Critically, the validation process must be performed securely and efficiently. Direct string comparison of hashes should be avoided due to its vulnerability to timing attacks. Hence it is recommended to use constant-time comparison functions, which take the same amount of time regardless of whether the hashes match. This helps mitigate timing-based attacks. Furthermore, the system should implement appropriate logging and monitoring of authentication attempts, including failed attempts, to detect suspicious activity. Regular security audits and penetration testing are also essential to ensure the continued effectiveness of the password validation process and identify any potential vulnerabilities.

5 SOFTWARE TESTING

5.1 UNIT TESTING

testNewAppPasswordPolicy.py

Fig. 11. Result of testNewAppPasswordPolicy.py

- **Successful Policy Creation:** This test verifies if the endpoint correctly creates a new password policy when provided with valid application details and policy parameters.
- **Existing Policy Handling:** This test ensures the endpoint rejects attempts to create a new policy for an application that has already been registered.
- **Missing Input Validation:** This test checks if the endpoint handles missing required data (e.g., app name) gracefully and returns appropriate error messages.
- **Invalid Policy Parameters:** This test verifies if the endpoint detects and rejects requests with nonsensical policy parameters (e.g., zero password length).

testChangePasswordPolicy.py

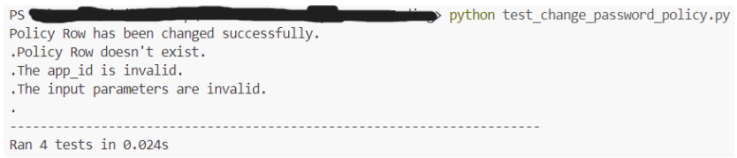

Fig. 12. Result of testChangePasswordPolicy.py

- **Successful Policy Update:** This test verifies if the endpoint correctly updates an existing password policy when provided with a valid application ID and new policy parameters.
- **Non-existent Policy Handling:** This test ensures the endpoint gracefully handles attempts to update a policy for a non-existent application ID.
- **Invalid Application ID:** This test checks if the endpoint handles invalid application ID formats appropriately and returns informative error messages.
- **Invalid Policy Parameters:** This test verifies if the endpoint detects and rejects requests with nonsensical policy parameters (e.g., negative password length).

testGeneratePassword.py

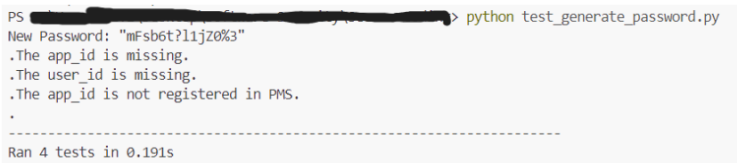

Fig. 13. Result of testGeneratePassword.py

- **Successful Password Generation:** This test verifies if the endpoint correctly generates a password when provided with valid application and user IDs.
- **Missing Input Validation:** The two tests ensure that the endpoint handles missing required data (e.g., app ID or user ID) gracefully and returns appropriate error messages.
- **Non-existent Application ID:** This test checks if the endpoint detects attempts to generate a password for an unregistered application and returns an informative error message.

testVerifyPassword.py

```
PS [redacted] > python test_verify_password.py
Authentication succeeded Successful
.Authentication failed Successfully
.The user_id is from the database.
.The user_id is missing from the parameters.
.
----------------------------------------------------------------------
Ran 4 tests in 0.161s
```

Fig. 14. Result of testVerifyPassword.py

- **Successful Login:** This test verifies if the endpoint correctly authenticates a user with a valid password for the specified application.
- **Incorrect Password:** This test checks if the endpoint rejects login attempts with incorrect passwords and returns an appropriate error message.
- **Non-existent User:** This test ensures the endpoint handles attempts to login with a non-existent user ID and returns an informative error message.
- **Missing Input Validation:** This test verifies if the endpoint handles missing required data (e.g., user ID) gracefully and returns appropriate error messages.

5.2 STATIC CODE ANALYSIS USING BANDIT

Bandit is a Python static analysis tool that helps developers find common security vulnerabilities in their code. By parsing the code and building an Abstract Syntax Tree, Bandit identifies potential issues like insecure data handling, improper input validation, and weak cryptographic practices. This allows developers to proactively address security concerns early in the development process.

```
Code scanned:
        Total lines of code: 382
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 7
                Medium: 17
                High: 2
        Total issues (by confidence):
                Undefined: 0
                Low: 17
                Medium: 3
                High: 6
Files skipped (0):
```

Fig. 15. Evidence for using Bandit Tool

5.3 STATIC CODE ANALYSIS USING SONARQUBE

SonarQube is an open-source platform that enables continuous code quality and security analysis. Through static code analysis, it identifies bugs and security vulnerabilities and measures code coverage across numerous programming languages, including Python. Its web-based interface provides

S. No.	Issue	Severity	Confidence	CWE
1	[B113:request_without_timeout] Call to requests without timeout	Medium	Low	CWE-400
2	[B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.	Low	High	CWE-330
3	[B324:hashlib] Use of weak SHA1 hash for security. Consider usedforsecurity=False	High	High	CWE-327

Table 5. Findings using Bandit

developers with visualizations of code quality metrics, allowing them to track issues over time and effectively manage technical debt. This continuous inspection helps teams proactively address code quality and security issues, leading to more robust and maintainable software. SonarQube's reporting and dashboard features offer insights into the overall health of the codebase, facilitating informed decision-making. By integrating SonarQube into the development workflow, teams can establish a culture of code quality and improve their software development practices.

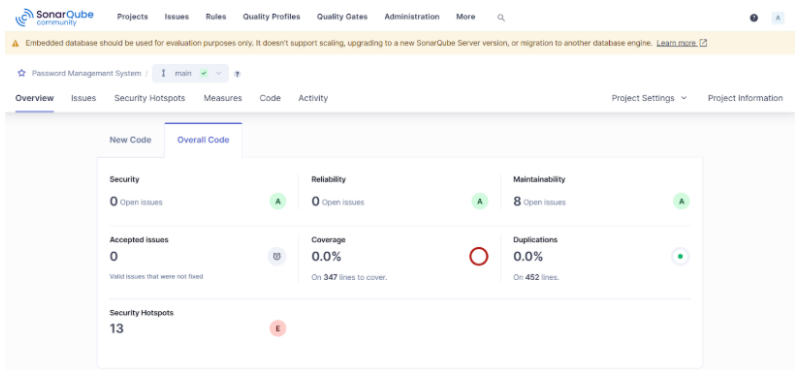

Fig. 16. Evidence for using SonarQube Tool

Table 6. SonarQube Findings

S. No.	Issue	Review Priority	CWE
1	Make sure disabling CSRF protection is safe here.	High	CWE-352
2	Make sure that using this pseudorandom number generator is safe here.	Medium	CWE-338, CWE-330, CWE-326, CWE-1241
3	Make sure that hashing data is safe here.	Low	CWE-1240

5.4 MAIN FINDINGS AND RECOMMENDATIONS

Static analysis tools, Bandit and SonarQube, revealed several potential security vulnerabilities in the Password Management System. These include the use of insecure cryptographic primitives, such as weak hashing algorithms and potentially insecure random number generation. The analysis also highlighted the potential for timeouts in network requests, which could lead to application instability and security risks. Furthermore, the tools identified possible Cross-Site Request Forgery (CSRF) vulnerabilities if CSRF protection is not implemented or is misconfigured.

To address these vulnerabilities, several key improvements are recommended. Security measures should be upgraded by replacing weak hashing algorithms with stronger alternatives like bcrypt or Argon2. Implementing timeouts for all network requests is crucial to prevent indefinite hangs. A careful review and implementation of CSRF protection mechanisms are also necessary. Code security should be improved by utilizing cryptographically secure random number generators for all security-critical operations. Finally, regular security audits and penetration testing should be conducted to proactively identify and address any further vulnerabilities.

6 CONCLUSION

In conclusion, the development of a secure and robust Password Management System (PMS) is paramount for protecting ACME Corporation's sensitive data and ensuring the integrity of its internal systems. This report has detailed the comprehensive SSDLC process undertaken, from initial requirements gathering and threat modeling to the prototypical Python implementation and rigorous testing. The emphasis on a "shift-left" security approach, incorporating security considerations at every stage, has resulted in a PMS designed to meet stringent FIPS standards and enforce dynamic password policies. The use of strong cryptographic hashing algorithms, secure storage mechanisms, and thorough input validation, coupled with regular security audits and penetration testing, further strengthens the system's defenses against potential threats. The implemented PMS offers a secure and reliable solution for managing passwords across ACME Corporation's applications, mitigating the risks associated with password compromise and data breaches. This proactive approach to security ensures the long-term protection of user credentials and contributes to a more secure operational environment.

A APPENDIX

A.1 IMPORTANT LINKS

(1) The **Google Docs link** provides a comprehensive analysis of the Password Management System (PMS), detailing all aspects of the project. While the document uses examples for illustrative purposes, the complete analysis, including user stories, requirements tables, threat modeling using both IRIUS Risk and the Microsoft Threat Modeling Tool, is contained within its pages.
(2) The **EduGit repository** hosts the complete source code for the Password Management System, including both the main application code and the associated unit tests.