Password Management System

Iqbal Khatoon, Iran Izadyariaghmirani

COSC 4010/5010 University of Wyoming Department of Computer Science

Secure Software Design April 02, 2023

Project Design Document

(i) Design Document

In this document, we provide a formalized design of a Python-based Password Management system. With the password manager, users no longer need to remember multiple passwords for different accounts. Instead, they can create complex passwords using the built-in password generator tool and save them securely in a local database. The database is encrypted with a master password, which only the user knows, providing an extra layer of security to protect their passwords. Below are the details of this system:

☐ A discussion of what the software does

The primary goal of the password manager is to make it easy for users to create strong, unique passwords for each of their accounts while providing a secure and convenient way to access those passwords. This is critical for protecting the confidentiality, integrity, and availability of user accounts and systems. When developing a password management utility, it is essential to consider security measures such as strong encryption to protect the passwords stored in the utility. Secure password generation algorithms create strong and unique passwords for users and provide protection against common attacks, such as brute force attacks.

The password manager requires the user to create a master password, which is used to encrypt and decrypt the stored passwords. Once the master password is set, the user can add passwords for different accounts to the password manager. The passwords are encrypted using the Advanced Encryption Standard (AES) before being stored in the MarioDB database, ensuring they are protected from unauthorized access. The password manager also includes a password generator that can generate strong, unique passwords with a combination of uppercase and lowercase letters, numbers, and symbols. The password manager requires users to enter a master password to access the stored passwords, ensuring only authorized users can view or modify the password data.

☐ Descriptions of individual components

The code was developed in Python programming language and linked to the MarioDB database. Python is a high-level programming language used to implement the password manager functionality. Python is known for its simplicity, readability, and ease of use, and it has a large and active developer community. We included various modules in the utility for the following tasks:

I. Master passwords: Master password: The master password is a password that is required to access the password manager and the stored password data. It is a critical security feature that ensures only authorized users can access the password data. Admin control is a feature that allows an administrator to manage the password manager, including adding and removing data (username, passwords, websites.
II. Password generator: The password generator is a key component of the password manager, as it is used to generate strong, unique passwords that are difficult to guess or crack. The password generator uses a combination of uppercase and lowercase letters, numbers, and symbols and allows users to specify the length and complexity of the generated password.
III. Password storage and encryption: securely store all user passwords and sensitive information using strong encryption algorithms. This ensures that the passwords cannot be read or accessed by unauthorized users. This module uses a Cryptography library to implement the Advanced Encryption Standard (AES) encryption algorithm before storing passwords in the database. AES is a widely-used encryption standard known for its security and efficiency.
IV. Password strength checker: analyzes the strength of passwords and provides feedback on how to improve them. This module is a loop, which asks the user to include at least a lowercase letter, an uppercase letter, a digit, and a special character in the passwords.
V. Data recall: The data recall feature allows users to retrieve their stored username and password data after entering the master password.

There are other modules that one may consider adding to this system to further improve the security. Examples of these modules are listed below:

VI. Two-factor authentication: adds an extra layer of security to the password manager by requiring users to provide a second factor of authentication, such as a code sent to their phone, in addition to their master password.
VII. Forgot password and recovery: This module is a mechanism for users to recover their account if they forget their master password or if they are locked out of their account for some other reason.

☐ Informal contracts each component must adhere to

I. Master passwords: The master password must be kept confidential and not be stored in plain text. The program must prompt the user to enter the master password before allowing access to the password manager. The program should enforce a minimum password complexity requirement for the master password.
II. Password generator: The password generator must generate strong and unique passwords that are difficult to guess or crack. Users should be able to specify the length and complexity of the generated passwords.
III. Password storage and encryption: User passwords and other sensitive information must be stored securely using strong encryption algorithms. The encryption key used to encrypt the password data must be derived from the master password and not be stored in plain text. The program should use a secure and industry-standard encryption algorithm, such as AES. The program should protect the encryption key and database file from unauthorized access.

IV. Password strength checker: The password strength checker should ensure that all user passwords meet minimum complexity requirements and are not easily guessable. The component should also provide feedback to users on how to improve their passwords and should never store or transmit passwords in plain text.
V. Data recall: The program should implement secure methods for data recall to prevent unauthorized access to user data. The program should not reveal any password information without proper authentication and authorization.

☐ Descriptions of how individual components integrate with each other

Here is a description of how the individual components of the password manager integrate with each other:

When the program starts, the program prompts the user to set a master password. The master password is the primary authentication mechanism for accessing the password manager and the stored password data. The master password component is tightly integrated with the password storage and encryption component, as the master password is used to encrypt and decrypt the stored passwords and data. The next time that user enters a command, the system prompts the user to enter the master password. If the entered master password matches the stored password hash, the user is granted access to the password manager. The user can then add entries, including usernames, website names, and passwords. The password generator (generate_password function in utils.py) can also be used to generate strong and unique passwords for different websites or applications. The password is first checked by the strength checker and then stored securely in the encrypted database using AES encryption. When users want to recall a stored password, they must enter their master password to decrypt the database and retrieve the password information. Overall, these components work together to ensure that user passwords and sensitive information are always kept secure and protected from unauthorized access.

☐ Discussion of potential security challenges (threat modeling)

There are several potential security challenges that the password management system described above may face and have been addressed in the code. Threat modeling involves identifying potential threats and vulnerabilities to the system and taking steps to mitigate them. Some potential threats to consider are listed as follows:

- Insufficient encryption: This threatens the confidentiality and integrity of sensitive data in a system and occurs when the encryption techniques employed in the system are weak or not properly implemented, thereby making it possible for attackers to access and manipulate data that should be secure. To prevent this attack vector, we used a strong encryption algorithm (Advanced Encryption Standard (AES) with complex encryption keys that are not easily guessable.
- Password guessing attacks (Brute force attack): Weak passwords are vulnerable to password guessing attacks, where an attacker systematically tries different combinations of usernames and passwords until they find a match. To mitigate this threat, we enforced the use of strong password policies, as explained before.

- Logging: The lack of logging can pose a threat of not being able to detect and respond to security incidents, as it provides no record of events and activities within the system. Without logging, it becomes difficult to trace and investigate malicious activities or to identify potential vulnerabilities that may have been exploited. Attackers can take advantage of this by performing malicious activities without being detected, such as stealing sensitive information or installing malware. In addition, it can be difficult to detect insider threats, as there is no record of their activities. To mitigate this threat, we have implemented a robust logging mechanism that records all activities within the system.
- Injection Attacks: Attackers could exploit SQL injection vulnerabilities to get unauthorized access to sensitive data or even complete control over the database. The password manager uses parameterized SQL queries to prevent SQL injection attacks. Parameterized SQL queries ensure that user input is properly escaped and quoted before being used in a database query, preventing malicious input from being executed as SQL code. Another attempt will be to include input validation and sanitization to ensure that the inputs are formatted correctly and free of any malicious characters or commands.
- Denial of Service (DoS) Attacks: Attackers could overwhelm the system with a flood of requests, causing it to become unresponsive or crash. DoS aims to disrupt the normal functioning of a system, service, or network by overwhelming it with traffic, requests, or data. The goal of a DoS attack is to make a service or network unavailable to its users by causing it to crash or become unresponsive. Mitigating the risk of DoS attacks involves a combination of preventative measures and proactive monitoring. The application currently does not include rate limiting to mitigate potential attacks.

(ii) A High-Level Proof Sketch using Type First Design

In this section, we discuss how to use Type First Design to provide a high-level proof sketch that demonstrates the software is secure. "Type first design" is a term that refers to a design approach that emphasizes the use of strong typing systems in programming languages to ensure the correctness and safety of software. In this approach, the software is designed and implemented in such a way that the types of data and objects are carefully specified and enforced by the programming language. In the context of password management software, Type First Design can be applied to ensure that all inputs and outputs are correctly typed, and that there are no type mismatches, null pointer dereferences, or other type-related errors that could lead to security vulnerabilities.

(ii)-a: Specification of Types and Interactions

Below is a formal specification of the software's behavior and properties using a type system. The following types of objects involved in the password manager system:

- **MasterPassword**: This type represents the master password used to access the password vault. It contains a single field, password, which is a string representing the actual password. When a user creates a new account, they must set a master password, which is then stored in an encrypted form using the Encryption type. The MasterPassword type interacts with the PasswordVault type through a “check master password” function, which takes a MasterPassword object as an argument and returns True if the password matches the stored master password.

- **Password:** This type represents a password, which consists of a randomly generated string of characters. It is associated with other information, including the website name, website URL, and username. The Password type interacts with the PasswordVault type to add passwords in the vault, and also with the PasswordGenerator type.

- **PasswordPolicy:** This type represents the password policy enforced by the password manager API. It typically contains fields such as minimum password length and password complexity requirements. The PasswordPolicy type interacts with the Password type to ensure that users create strong and secure passwords.

- **User:** This type represents the user of the password manager API. It contains fields such as username, email, and account creation date. The User type User type interacts with the PasswordVault and MasterPassword types to manage user authentication, access control, and user-specific settings.

- **PasswordVault**: This type contains all of the user's stored passwords and is implemented as a database, with each password stored in an encrypted form using the Encryption type. The PasswordVault type interacts with the following types.
  - Takes a MasterPassword object as an argument and returns True if the provided password matches the stored master password.
  - Takes a Password object as an argument and adds it to the passwords list.

    - Takes a website url string as an argument and returns the Password object with the matching website url from the passwords list in PasswordVault.

- **Encryption:** This type represents the encryption algorithm used to encrypt and decrypt sensitive information, such as the master password and individual passwords. The Encryption type contains two functions, encrypt() and decrypt(), which take a plaintext string and a secret key as arguments. The Encryption type is used by the PasswordVault and MasterPassword types to encrypt and decrypt passwords.

- **AuditLog:** This type represents a log of all actions taken within the password manager API, including user logins, password changes, and database updates. The AuditLog type interacts with the PasswordVault and User types to monitor user activity, detect suspicious behavior, and track changes made to the password vault.

- **PasswordGenerator:** This type is responsible for generating random passwords of a specified length and complexity. It uses the Password type to represent the generated passwords. The PasswordGenerator takes an integer argument specifying the desired length of the password and returns a randomly generated password string.

(ii)-b: Main Functions

The main functions that should be considered are as follows:

- A function to create a new user account, which would involve setting a master password, creating a new password vault, and storing user information in the database.
- A function to add a new password to the vault, which would involve generating a new password using the PasswordGenerator type, encrypting it using the Encryption type, and adding it to the vault using the PasswordVault type.
- A function to retrieve a password from the vault, which would involve decrypting the password using the Encryption type and returning it to the user.
- A function to enforce the password policy, which would involve checking the PasswordPolicy type for password complexity requirements and expiration rules, and rejecting passwords that do not meet these criteria.
- A function to log user activity, which would involve creating a new entry in the AuditLog type for each action taken by the user and storing this information in the database.

(ii)-c: Other considerations

Other steps to use the type first design to generate a type-safe implementation of the software are summarized below:

1) Code review: A team of developers can review the implementation code to check for security vulnerabilities such as buffer overflows, SQL injection attacks, and other common coding mistakes.
2) Use programming languages that provide built-in support for type checking and static analysis. These languages and tools provide a solid foundation for formal specification and type-driven development. Implementation the necessary practices to ensure that all inputs and outputs are correctly typed and that there are no type mismatches, null pointer dereferences, or other type-related errors that could lead to security vulnerabilities.
3) Static analysis tools: Static analysis tools such as FindBugs, PMD, and Checkstyle can help to identify potential security issues by analyzing the code for known patterns of vulnerabilities. Use static analysis and testing tools to verify that the implementation satisfies the formal specification and does not contain any security flaws. This can include techniques such as type checking, symbolic execution, and model checking.
4) Dynamic analysis tools: Dynamic analysis tools such as Burp Suite, Zed Attack Proxy, and OWASP Zap can simulate attacks on the software and identify security vulnerabilities such as SQL injection, cross-site scripting, and other types of web application vulnerabilities.
5) Fuzz testing: Fuzz testing involves sending random or invalid inputs to the software to see how it behaves under unexpected conditions. This can help to uncover security vulnerabilities such as buffer overflows and other input-related issues.

in addition to ensuring correct typing of inputs and outputs, a type-safe implementation should also consider the following security-related issues (in addition to authentication and encryption):

- ☐ Input validation: The implementation should validate all inputs to prevent buffer overflows, SQL injection attacks, cross-site scripting attacks, and other types of input-related vulnerabilities.
- ☐ Error handling: The implementation should handle errors and exceptions gracefully, and avoid exposing sensitive information or system details in error messages.
- ☐ Resource management: The implementation should manage resources such as memory, file handles, and network sockets carefully, and prevent resource leaks or denial of service attacks.

By using a combination of these techniques, we can verify that the implementation satisfies the formal specification and does not contain any security flaws. This can help to improve the overall security of the software and reduce the risk of successful attacks. Nonetheless, it is necessary to continuously monitor and update the software to address any new security threats or vulnerabilities that are discovered.