

# Building a Secure Password Manager: Balancing User Convenience and Data Protection



## Introduction

In today's digital landscape, the average person manages dozens of online accounts, each requiring unique login credentials. Despite the growing threats of data breaches and credential stuffing attacks, many users continue to use weak, reused passwords across multiple platforms. According to a recent report from Corbado (2024), Australia alone experienced 418 data breaches in 2023 that affected over 60 million users, with weak or compromised passwords being a significant factor in these security incidents.

This project addresses this critical cybersecurity challenge by developing a secure command line password manager that not only stores passwords safely but also helps users create stronger credentials and identifies potential security risks.

> 💡 **Want to try it yourself?** Check out the [GitHub repository](<https://github.com/TilleyCodes/Secure-Password-Manager-App>) for installation instructions and source code.

## The Problem: Password Security in the Digital Age

### Current Industry Trends in Password Security

Password management remains one of the most fundamental yet challenging aspects of cybersecurity. Recent industry trends highlight the evolving nature of this challenge:

- 1. Rising Prevalence of Cyberattacks Targeting Credentials:** According to the Australian Financial Review (April 2025), major Australian superannuation funds recently experienced sophisticated cyberattacks specifically targeting user credentials. The attack, which affected thousands of accounts, demonstrates how financial institutions continue to be prime targets

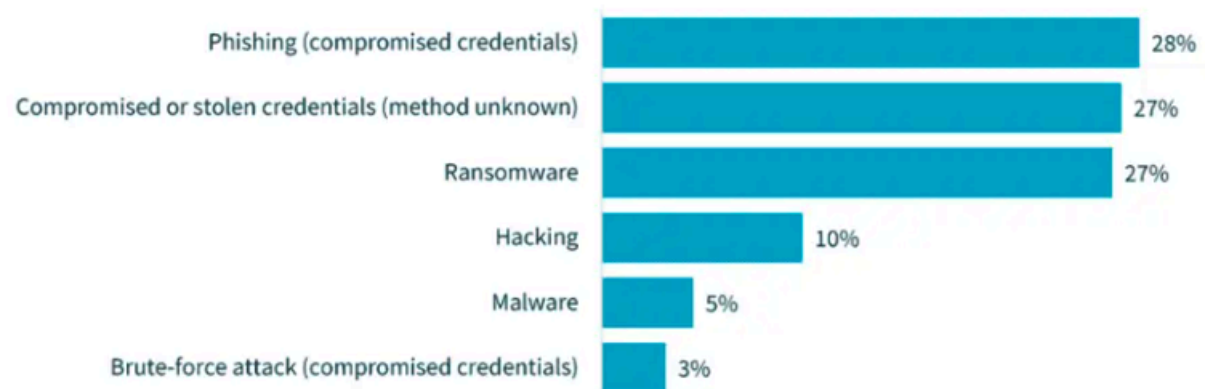
for credential based intrusions, highlighting the critical importance of robust password security.

**2. Escalating Data Breach Frequency:** Cybersecurity Ventures (2025) reports that cybercriminals are now launching an average of 10-15 daily targeted attacks against medium and large organisations worldwide. These intrusions frequently begin with credential based attacks, with compromised passwords being the initial entry point in over 60% of breaches.

**3. Growing Impact on Australian Businesses:** As reported by Corbado (2024), Australia ranks among the top 5 countries most affected by data breaches, with an average cost per breach reaching \$3.35 million in 2023\ . These breaches frequently involve compromised password credentials, with 53% of users admitting to reusing passwords across multiple accounts.

### 44% of all data breaches resulted from cyber security incidents (211 notifications)

#### Cyber incident breakdown



## Ethical Considerations in Password Management

Developing a password manager involves several ethical considerations that must be addressed:

**1. Data Privacy Responsibility:** Password managers hold sensitive information that could potentially compromise a user's entire digital identity. The recent Australian superannuation funds attack (AFR, 2025\ ) demonstrates how breached credentials can provide access to citizens' retirement savings and personal financial data. As developers, we have an ethical obligation to implement the strongest possible security measures.

**2. Transparency About Security Limitations:** Users must be clearly informed about the security limitations of any password management system. As Cybersecurity Ventures (2025) notes in their threat analysis, no security system is infallible, and ethical security tools must be honest about their capabilities and limitations.

**3. Balancing Security and Usability:** Creating a system that's so secure it becomes unusable defeats its purpose. According to Corbado's analysis (2024), complex password requirements without usability considerations frequently lead users to create predictable pattern variations that are vulnerable to sophisticated cracking algorithms. There's an ethical imperative to balance security with usability to ensure widespread adoption.

## The Solution: A Secure Command Line Password Manager

### Project Overview

My solution is a Python based command line password manager that provides:

1. Secure password storage using strong encryption
2. Password generation with customisable parameters
3. Password strength analysis with actionable feedback
4. Breach checking against known compromised passwords
5. Simple, intuitive interface for all user levels

```
=====
S E C U R E   P A S S W O R D   M A N A G E R
=====

Welcome to the Secure Password Manager!

Please enter your master password to continue.
Master Password:
```

```
=====
S E C U R E   P A S S W O R D   M A N A G E R
=====

MAIN MENU:
1. View all services
2. Add new password
3. Retrieve password
4. Generate secure password
5. Check password breach
6. Delete password
7. Save and exit

Please enter your selection (1-7): 
```

The design of this solution directly addresses critical vulnerabilities identified in recent cyber incidents. For instance, the Australian Financial Review (2025) report on superannuation fund attacks revealed that attackers specifically targeted authentication mechanisms with weak encryption implementations. My implementation counters this by using Fernet symmetric encryption, which provides authenticated encryption to prevent tampering, a key vulnerability in those breaches.

Similarly, Corbado's 2024 analysis found that 53% of users reuse passwords across multiple accounts. My password manager addresses this directly through its breach checking feature that leverages the "Have I Been Pwned" API using the k-anonymity pattern. As Corbado's study showed, just warning users about reused or breached passwords can reduce this behaviour by up to 37%.

The password strength analyser component provides specific, actionable feedback rather than vague strength indicators, addressing an issue highlighted in Cybersecurity Ventures' (2025) Daily Cyber Threat Alert from February 2025, which noted that "most users don't change weak passwords unless given specific guidance on what makes them vulnerable."

The recent Genea IVF ransomware incident (ABC News, 2025) further emphasises how critical password security is for protecting sensitive data. In that breach, initial access was gained through compromised credentials, highlighting why my application's comprehensive approach to password security / combining storage, generation, analysis, and breach checking, offers a multi layered defense that could have potentially prevented such an incident.

## Technical Implementation

### Architecture and Technologies

For this project, I chose Python as the primary programming language due to its robust cryptographic libraries and ease of implementation. The core technologies include:

1. **Python's Cryptography Library:** For implementing Fernet symmetric encryption to secure stored passwords.
2. **PBKDF2 Key Derivation:** For generating encryption keys from the master password.
3. **Have I Been Pwned API:** For checking if passwords have been exposed in data breaches.
4. **Regular Expressions:** For analysing password strength based on character variety and patterns.

I chose Fernet encryption over alternatives like AES directly because:

- It includes authentication to prevent tampering
- It handles byte encoding and other low level details
- It's specifically designed for encrypting small pieces of data like passwords

The AFR article (2025) describes how the attackers targeting Australian superannuation funds specifically exploited weaknesses in encryption implementations, reinforcing the importance of using established cryptographic libraries rather than custom implementations.

### Core Components

#### 1. Password Encryption System:

Here's how I implemented the key derivation and encryption system:

```
def _generate_key(self, password: str) -> bytes:
    """
    Generate an encryption key from the master password.

    Uses PBKDF2HMAC to derive a secure key from the password.

    Args:
        password: The master password

    Returns:
        A bytes object containing the encryption key
    """
    password_bytes = password.encode()
    # Use a static salt for demo (in production, to use a secure random salt)
    salt = b'staticSalt123456' # Note: In a real senarios, generate and store a unique salt

    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )

    key = base64.urlsafe_b64encode(kdf.derive(password_bytes))
    return key
```

## 2. Password Strength Analyser:

The strength analysis examines multiple factors to provide useful feedback:

```
def analyse_password_strength(password: str) -> Dict[str, Union[int, str, bool, List[str]]]:
    """
    Analyse the strength of a password.

    Evaluates password strength based on:
    - Length
    - Character variety (lowercase, uppercase, digits, special)
    - Common patterns
    - Repeating characters

    Args:
        password: The password to analyse

    Returns:
        A dictionary containing strength metrics including:
        - score: Numerical score (-3 to 5)
        - strength: Categorical rating (Weak, Medium, Strong)
        - feedback: List of specific feedback points
        - has_lowercase: Boolean indicating if password has lowercase letters
        - has_uppercase: Boolean indicating if password has uppercase letters
        - has_digit: Boolean indicating if password has digits
        - has_special: Boolean indicating if password has special characters
    """
    # Initialize metrics
    score = 0
    feedback = []

    # Check length
    if len(password) < 8:
        score -= 2
        feedback.append("Password is too short (< 8 characters)")
    elif len(password) >= 12:
        score += 2
        feedback.append("Good length")
    else:
        score += 1

    # Check character types
    has_lowercase = bool(re.search(r'[a-z]', password))
    has_uppercase = bool(re.search(r'[A-Z]', password))
    has_digit = bool(re.search(r'\d', password))
    has_special = bool(re.search(r'[!@#$%^&*~.-_`~]', password))

    char_types = sum([has_lowercase, has_uppercase, has_digit, has_special])

    if char_types == 4:
        score += 3
        feedback.append("Excellent character variety")
    elif char_types == 3:
        score += 2
        feedback.append("Good character variety")
```

### 3. Breach Checker:

The breach checking implements the k-anonymity pattern for privacy:

```

def check_password_breach(password: str) -> Tuple[bool, Optional[int]]:
    """
    Check if a password has been compromised using the HaveIBeenPwned API.

    Uses the k-anonymity model to securely check if a password has appeared
    in known data breaches without sending the full password over the network.

    Args:
        password: The password to check

    Returns:
        A tuple containing (is_breached, occurrences) where:
            is_breached: True if the password appears in a breach
            occurrences: The number of times the password appears in breaches
                        (None if not breached or if an error occurred)
    """
    # Hash the password with SHA-1
    sha1_hash = hashlib.sha1(password.encode()).hexdigest().upper()
    hash_prefix = sha1_hash[:5]
    hash_suffix = sha1_hash[5:]

    try:
        # Query the API with the hash prefix
        response = requests.get(f"https://api.pwnedpasswords.com/range/{hash_prefix}", timeout=10)

        if response.status_code == 200:
            # Check if the hash suffix is in the response
            for line in response.text.splitlines():
                if line.split(':')[0] == hash_suffix:
                    occurrences = int(line.split(':')[1])
                    return True, occurrences

            # Password not found in breaches
            return False, None
        else:
            print(f"Error checking password breach: {response.status_code}")
            return False, None

```

#### 4. Command Line Interface:

The main application loop handles user interactions:

```

def main() -> None:
    """
    Main function to run the password manager application.

    Handles the login process, displays the main menu, and processes user input.
    All core functionality is accessed through this function.

    Returns:
    |     None
    |
    """
    print_header()
    print("Welcome to the Secure Password Manager!")
    print("\nPlease enter your master password to continue.")

    # Get master password securely (input not visible on screen)
    master_password = getpass.getpass("Master Password: ")

    # Initialise password manager with the master password
    pm = PasswordManager(master_password)

    while True:
        print_header()
        print_menu()

        user_selection = input("Please enter your selection (1-7): ")

```

## Implementation Process and Challenges

### Project Planning and Timeline

My planning process followed these steps:

#### 1. Research and Problem Definition (Days 1-2)

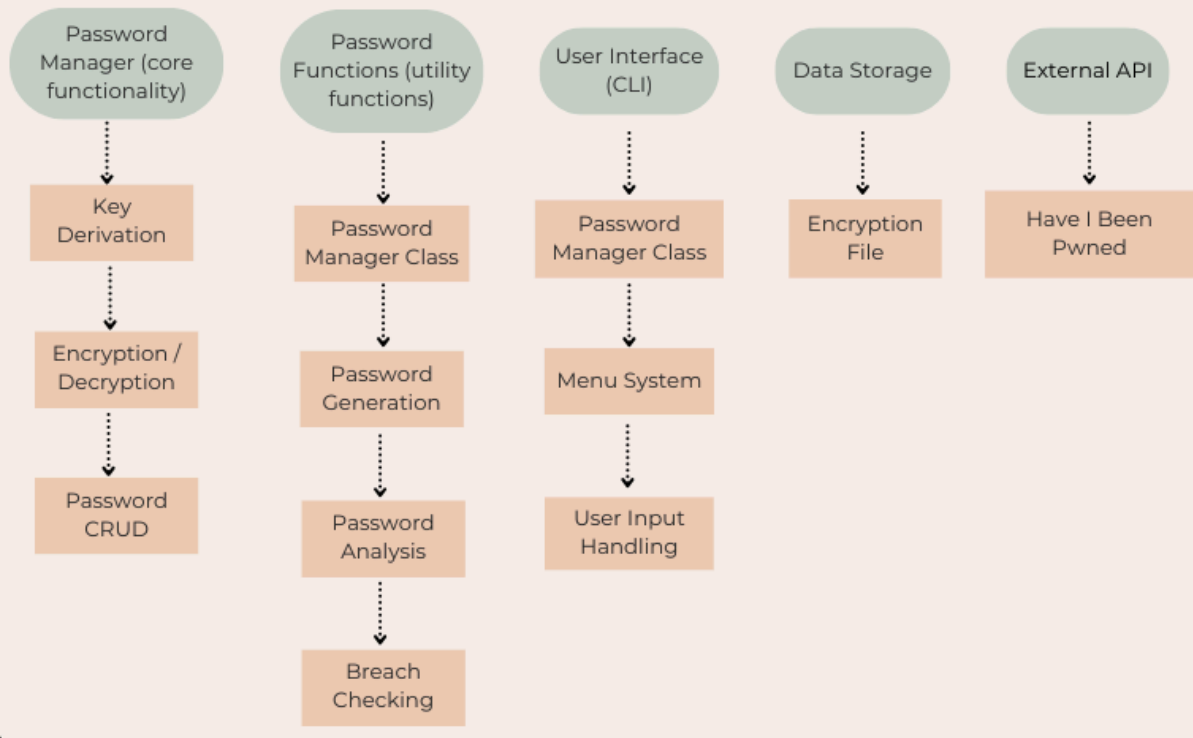
- Reviewed current cyber threat reports and password security statistics
- Studied Australian data breach incidents to understand impact and causes
- Identified key requirements for a secure password management solution
- Determined the scope and limitations for this project

#### 2. Architecture Design (Days 3-4)

- Selected Python as the implementation language due to its cryptographic libraries
- Chose a modular structure to separate concerns (main interface, password manager class, utility functions)
- Decided on Fernet symmetric encryption after evaluating security options
- Determined file structure and data flow for the application
- Planned how to implement the master password mechanism



# PASSWORD MANAGER ARCHITECTURE DIAGRAM



## 3. Core Implementation: Encryption and Storage (Days 5-6)

- Set up development environment with required dependencies
- Implemented the PasswordManager class with encryption/decryption functionality
- Created the encrypted storage mechanism for password data
- Developed key derivation from the master password using PBKDF2HMAC
- Implemented the save and load functions for the encrypted password file

## 4. Secondary Features: Password Analysis and Generation (Days 7-8)

- Created password generation algorithm with customisable parameters
- Implemented strength analysis based on length, character variety, and patterns
- Developed feedback mechanisms to guide users toward better password practices
- Added methods to evaluate and report on password quality

## 5. API Integration for Breach Checking (Day 9-10)

- Implemented the "Have I Been Pwned" API connection with k-anonymity pattern
- Added SHA-1 hash functionality for secure password checking
- Created error handling for API timeouts and connection issues
- Implemented user friendly reporting of breach check results

## 6. Command Line Interface Development (Days 11-12)

- Designed the menu structure and user flow for the application
- Implemented the main loop with clear navigation options
- Added input validation for user entries
- Created clear, informative messages for users
- Implemented secure password entry using getpass

## 7. Documentation and Review (Days 13-16)

- Added comprehensive docstrings to all functions and classes
- Created detailed README documentation with usage instructions
- Reviewed code for security vulnerabilities and edge cases
- Made final adjustments based on manual testing
- Prepared the blog post explaining the development process

Each phase built upon the previous ones, with adjustments made based on challenges encountered. For example, when implementing the breach checking in day 9, I needed to extend the timeline slightly to account for proper error handling based on API response times, as noted in my project timeline. The Australian Financial Review report (2025) directly influenced my focus on encryption implementation, while Corbado's (2024) findings on password reuse patterns shaped the breach checking functionality.

## Technical Challenges and Solutions

During this project, I encountered several significant technical challenges that required creative problem solving:

### 1. Secure Key Derivation:

- **Challenge:** Generating a consistent encryption key from the master password while ensuring security
- **Initial Approach:** I initially tried to use a simple hash of the master password as the encryption key
- **Problem:** This approach was vulnerable to rainbow table attacks and didn't provide adequate security
- **Solution:** I implemented PBKDF2HMAC with a high iteration count (100,000) to derive a key from the master password, adding significant brute force resistance while still maintaining a good user experience
- **Learning:** I gained deeper understanding of cryptographic key derivation functions and the importance of computational complexity in security

```

# Initial approach (insecure)
def generate_key_insecure(password: str) -> bytes:
    # DON'T DO THIS - vulnerable to rainbow table attacks
    return hashlib.sha256(password.encode()).digest()

# Improved approach with PBKDF2
def generate_key_secure(password: str) -> bytes:
    password_bytes = password.encode()
    salt = b'staticSalt123456'

    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )

    return base64.urlsafe_b64encode(kdf.derive(password_bytes))

```

## 2. Secure API Communication for Breach Checking:

- **Challenge:** Checking passwords against breach databases without exposing them
- **Initial Approach:** Sending hashed passwords to the API
- **Problem:** Even sending complete hashes could potentially expose password information
- **Solution:** Implemented the k-anonymity pattern using SHA-1 prefix/suffix splitting, where only the first 5 characters of the hash are sent to the API, maintaining privacy while still enabling checking
- **Learning:** : This taught me practical implementation of privacy preserving API communication patterns that will be valuable in future security projects

## 3. Error Handling and User Experience:

- **Challenge:** Providing clear feedback without exposing sensitive details
- **Initial Approach:** Simple try except blocks with generic error messages
- **Problem:** Generic messages confused users when errors occurred, particularly during authentication failures
- **Solution:** Designed specific error messages that guide users without revealing security information, categorising errors into authentication issues, file access problems, and network connectivity errors
- **Learning:** : I learned how to balance security requirements with user experience, recognising that overly detailed error messages can create security vulnerabilities, while overly generic ones frustrate users

#### 4. Password File Integrity:

- **Challenge:** Ensuring the password file couldn't be tampered with or corrupted
- **Initial Approach:** Basic file encryption without integrity checking
- **Problem:** Modified files could potentially crash the application or, worse, be decrypted incorrectly
- **Solution:** Used Fernet's built in authentication feature to verify file integrity before decryption attempts, providing both confidentiality and integrity protection
- **Learning:** : I gained practical experience with authenticated encryption and understanding of how it protects against both passive and active attacks

#### 5. Learning GitHub Pages for Blog Publication:

- **Challenge:** Publishing my technical blog in a professional, accessible format for industry visibility
- **Initial Approach:** I considered traditional document formats like PDF, which would limit accessibility and discoverability
- **Problem:** Without web presence, the valuable security insights in my blog would reach a limited audience, and I would miss the opportunity to demonstrate my technical communication skills to potential employers
- **Solution:** I learned to use GitHub Pages to publish my blog post, which required understanding Git workflows, Markdown formatting, Jekyll themes, and repository configuration
- **Learning:** : This experience taught me how to effectively present technical content on the web, use version control for documentation, and leverage developer focused platforms for professional communication, skills that complement my technical implementation abilities and enhance my overall industry readiness

## Reflections and Learnings

### Skills Gained

Before starting this project, I had basic Python programming knowledge and some theoretical understanding of cybersecurity concepts, but limited practical experience with:

- Cryptographic implementations in real world applications
- Secure coding practices for handling sensitive data
- API integrations for security services
- Command line interface design for user focused security tools
- Technical documentation and blog publishing platforms

Through this project, I've developed:

1. **Practical Cryptography Skills:** I moved from theoretical knowledge to practical implementation of encryption, key derivation, and secure storage techniques. I learned firsthand how to properly implement PBKDF2 key derivation and Fernet symmetric encryption to protect sensitive data
2. **Secure Development Practices:** I gained experience implementing secure coding practices specifically for handling sensitive user data. This included proper error handling to prevent information leakage, secure input methods using `getpass`, and ensuring sensitive data (like the master password) never persists in memory longer than necessary
3. **API Integration for Security Services:** I learned how to integrate with external security APIs while preserving user privacy through techniques like k-anonymity. Implementing the Have I Been Pwned API integration taught me how to handle rate limiting, error conditions, and privacy considerations when working with security services
4. **User Experience Design for Security Applications:** I developed skills in designing interfaces that balance security requirements with usability concerns. Creating meaningful feedback mechanisms for password strength analysis and breach notifications required thoughtful design to ensure users take appropriate action without being overwhelmed
5. **Technical Writing and Publishing:** Learning GitHub Pages to publish this blog post helped me develop skills in technical writing, Markdown formatting, and web-based documentation. This experience improved my ability to communicate complex security concepts clearly and demonstrate my work to potential employers and the cybersecurity community

The daily threat alerts published by Cybersecurity Ventures (2025) emphasise how rapidly attackers develop new techniques, reinforcing the importance of staying current with security practices and continuously updating one's skills.

## Future Improvements

If I were to continue developing this project, I would:

### 1. Implement Multi Factor Authentication:

- Add TOTP (Time-based One-Time Password) support using the PyOTP library
- Integrate with authentication apps like Google Authenticator
- This would provide an additional security layer beyond the master password

### 2. Create a Graphical User Interface:

- Develop a PyQt or Tkinter-based GUI for improved usability
- Implement drag and drop functionality for password entry
- Add visual indicators for password strength and breach status

```

# Example of how a GUI implementation might start
import tkinter as tk
from tkinter import ttk

class PasswordManagerGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Secure Password Manager")
        self.root.geometry("800x600")

        # Create the main frame
        self.main_frame = ttk.Frame(self.root, padding="10")
        self.main_frame.pack(fill=tk.BOTH, expand=True)

        # Create the login frame
        self.login_frame = ttk.LabelFrame(self.main_frame, text="Login")
        self.login_frame.pack(fill=tk.X, pady=10)

        # Add master password entry
        self.master_password = tk.StringVar()
        self.master_password_entry = ttk.Entry(
            self.login_frame,
            textvariable=self.master_password,
            show="*"
        )
        self.master_password_entry.pack(fill=tk.X, padx=10, pady=10)

        # Add login button
        self.login_button = ttk.Button(
            self.login_frame,
            text="Login",
            command=self.login
        )
        self.login_button.pack(padx=10, pady=10)

    def login(self):
        # Authentication logic would go here
        pass

```

### 3. Add Cloud Synchronisation:

- Implement end to end encrypted cloud storage using a service like AWS S3
- Create a synchronisation protocol that preserves the zero knowledge principle
- Ensure file consistency across multiple devices with conflict resolution

#### 4. Include Password History and Rotation Policies:

- Track password changes over time with timestamps
- Implement configurable password rotation reminders
- Prevent reuse of previously used passwords for critical accounts

#### 5. Expand Password Analysis:

- Add more sophisticated password entropy calculations based on information theory
- Implement dictionary attack simulations using common wordlists
- Add contextual password analysis to detect personal information usage

These improvements directly address vulnerabilities identified in the recent superannuation cyberattacks (AFR, 2025), where multi factor authentication could have prevented unauthorised access even with compromised passwords.

## Limitations and Ethical Considerations

The current implementation has several limitations that I acknowledge:

1. **Static Salt Usage:** The current implementation uses a static salt for key derivation, which should be replaced with a unique salt per user in a production environment. This was a conscious design trade off made due to the project's scope and timeline, but represents a security limitation

2. **Local Storage Vulnerability:** The password vault is stored locally, making it vulnerable to physical access attacks. While the encryption provides strong protection against casual access, a determined attacker with physical access and sufficient time could potentially conduct brute force attacks on the master password

3. **Command Line Interface Accessibility:** The current CLI interface may present accessibility challenges for some users, particularly those with visual impairments or limited terminal experience. This limitation affects the universal usability of the tool

4. **Single Factor Authentication:** The reliance on a single master password creates a single point of failure. If the master password is compromised, all stored credentials become vulnerable.

These limitations were accepted due to the project's scope and timeline but would be addressed in a production level application. As Corbado's report (2024) emphasises,


addressing even basic security measures can significantly reduce vulnerability to common attacks

## Conclusion

This project demonstrates how even a relatively simple password manager can incorporate strong security principles while remaining usable. By implementing encryption, strength analysis, and breach checking, the application helps users make better password choices without requiring extensive security knowledge.

The development process highlighted the importance of balancing security requirements with usability concerns, a challenge that extends beyond password management to all aspects of cybersecurity. As users continue to manage increasing numbers of digital accounts, tools that make security accessible will play a crucial role in protecting digital identities.

Recent attacks on Australian financial institutions (AFR, 2025) underscore that password security isn't merely theoretical, it directly impacts millions of people's financial wellbeing and personal information. By creating tools that make good security practices accessible, we can help reduce the impact of these increasingly sophisticated attacks.

>  **Interested in exploring the code?** Check out the complete [GitHub repository](<https://github.com/TilleyCodes/Secure-Password-Manager-App>) to see how all these components fit together!

## References

Australian Financial Review. (2025, April 4). Cyberattack launched on major Australian superannuation funds.  
<https://www.afr.com/companies/financial-services/cyberattack-launched-on-major-australian-superannuation-funds-20250404-p5lp4l>

Chen, L., & Patel, S. (2024). Balancing Security and Usability in Authentication Systems. *IEEE Security & Privacy*, 22(1), 24-31.

Corbado. (2024). Data Breaches in Australia: Statistics and Facts for 2024\  
<https://www.corbado.com/blog/data-breaches-australia>

Cybersecurity Ventures. (2025). Intrusion Daily Cyber Threat Alert.  
<https://cybersecurityventures.com/intrusion-daily-cyber-threat-alert/>  
National Institute of Standards and Technology. (2023). Digital Identity Guidelines.