JagzPass Design and Implementation

Motivation and Goals

When starting this project, I wanted to build a GUI-based portfolio piece and learn Python in the process. During COMP6841, I noticed how many scripts and tools, especially in wargames and personal projects , relied on Python. I've had minimal experience with python before this, so I chose to do my project in python as a way to learn it.

Rather than replicating full-featured password managers with cloud sync, autofill, or 2FA, I chose to keep JagzPass minimal. I wanted to focus on the core security logic and explore how password managers defend against real-world threats. This document outlines the reasoning behind key design decisions in JagzPass, the trade-offs I made, and what I learned along the way about cryptography and building secure software.

Project Structure and Files

Jagzpass initially had two files: a vault.py that controlled the core functionality of Jagzpass(vault encryption and credential storage) and main.py which had the CLI. As I added more features to my program, I did a major refactoring of the program to make it easily manageable (I also happened to be taking COMP2511 at the time). Refactoring after putting all the functions in two files was a bit of a hassle, especially because I didn't quite understand that I needed __init__.py to let the python interpreter know I was calling a custom python package. The functions of all the files are explained in the README.md of my project.

All cryptographic operations rely on cryptography and argon2-cffi, both of which I chose after researching secure defaults and audit history.

Key Derivation with Argon2id

Digging into password manager breaches and KDF benchmarks from Hashcat showed me how weak fast hashes like SHA-256 are against modern hardware. I settled on Argon2id for its balance of its memory-hardness and how tunable it was after extensively looking into other options (check my research document). JagzPass uses argon2-cffi's low-level hash_secret_raw to generate a 256-bit key.

Configuration:

- Salt: 16 bytes, generated using os.urandom(16), stored unencrypted with the vault.
- Time cost: 3 iterations
- Memory cost: 64 MiB
- Parallelism: 2 threads

This configuration keeps the login time reasonable while making brute-force and GPU cracking expensive. It is also better than the minimum OWASP recommendation of 19 MiB for the memory cost and 2 iterations. Choosing the right balance took trial-and-error because higher configurations took a little more time than I liked to hash and I think this config is safe enough while being quick to execute. If we look at it from a bits of work perspective, my configuration would take around 10 times more work than the minimum configuration; see my calculations below:

	Minimum OWASP configuration	My configuration
Memory cost (bytes)	19 MiB=19×220= **19,922,944 bytes**	64 MiB=64×220= **67,108,864 bytes**
Work = Memory Cost × Time Cost × Parallelism	19,922,944×2×1= **39,845,888 bytes**	67,108,864×3×2= **402,653,184 bytes**
My KDF takes around 10 times more effort to break!	402,653,184/39,845,888≈10.1	

Vault Encryption with AES-GCM

Unlike with the KDF, I didn't look to much into different kinds of encryption because almost all the password managers I looked at almost universally used AES; that's why I chose to use AES in Galois/Counter Mode (GCM) for my vaults. The encryption is handled by Python's cryptography library. I formatted each credential to be saved in the following format.

Vault File Format:

[salt][nonce][ciphertext + tag]

Details:

- Nonce (Number used once): 12-byte random value generated fresh for each encryption.
- Tag: 16 bytes appended by GCM for integrity/authentication.
- Encrypted data: JSON-encoded list of credentials, encrypted using the derived key.

AES-GCM was one of the things that just worked for me right off the bat. I was a bit worried about dealing with the authentication tag and having to isolate it and find out if any tampering had occurred but the cryptography library handles it.

Lockout and Rollback Protection

One of the subtler challenges I faced in JagzPass was designing a lockout system that couldn't be easily bypassed. I was inspired to protect against rollbacks particularly because of the Bitwarden case I analyzed in my research. Just preventing logins after too many wrong

attempts wasn't enough; someone could just edit or delete the .lock file or roll back the system clock to cheat the timer. So I made the current UTC time from an external API (worldtimeapi.org) the primary source of truth for lockout duration. This external reference makes it harder for an attacker to spoof time-based conditions just by adjusting their local system clock.

However, I did know that it would be entirely possible that the user was offline. I don't know why someone offline would want to look for passwords they need to use online but nonetheless, I tried to account for it. I used time.monotonic(), which is basically a system clock that always moves forward and isn't affected by the user changing system time. When a user is being locked out, I store both the default system clock and the montonic clock. When the user reattempts a login, I would check if the difference in the montonic time matches the system clock time to ensure that enough time has legitimately passed.

Password Generation and Reuse Detection

Studying the LastPass breach emphasized the real-world impact of password reuse. Attackers were able to move exploit reused passwords and steal $150 000 000 as far as we know. This made me realize that even if credentials are encrypted, reusing weak passwords makes the entire vault vulnerable. JagzPass addresses this by:

- Checking all user-provided passwords against the NCSC top 100k list.
- Preventing password reuse across saved entries.

This not only improves security but also enforces a culture of good password hygiene. Implementing this was a learning moment for me when it came to writing python, it taught me a lot about iterating over nested dictionaries in Python.

CLI and User Interaction

The command-line interface is simple and deliberately linear. While I wanted to build a GUI initially, I kept things CLI-only so I could focus on security and logic first. The CLI also gives real-time feedback on password strength using color-coded messages (red/yellow/green), which turned out to be one of the most fun features to implement.

Threats Addressed

Brute-Forcing
JagzPass uses an Argon2id configuration that takes a minimum of ten times longer to crack than the minimum OWASP requirements.

Password Reuse and Dictionary Attacks
The system checks passwords against the NCSC top 100,000 list and refuses to accept any matches as valid credentials. It also detects password reuse across entries during vault creation and modification.

Vault Tampering
AES-GCM provides authenticated encryption. Any modification to the encrypted vault causes decryption to fail due to tag mismatch. This protects the vault against tampering and ensures data integrity.

Rainbow Table Attacks
JagzPass never stores raw keys or passwords. All the vaults are salted and each time a credential is added to a vault, the entire vault is resalted with a different salt, rendering rainbow table attacks ineffective.

Replay or Restoration Attacks
Each vault encryption uses a fresh 12-byte nonce and a new salt. This ensures that even if the same vault contents are encrypted twice, the resulting ciphertext will be entirely different. Observing encrypted vault files over time does not reveal reuse or data patterns.

Credential Exposure on Disk
Credentials are only decrypted in-memory when actively in use. There is no plaintext stored on disk at any point. Even temporary files are avoided by design.

Vulnerabilities, Reflection, and Future Work

If I had more time or a bigger scope, I'd want to address several known limitations and polish edges:

Vault deletion
Right now, the .vault file can be deleted manually. While deletion isn't a vulnerability in itself, there's no tamper protection or recovery mechanism. I'd like to implement checksums, soft delete, or vault backups (encrypted) in future versions.

GUI
JagzPass is terminal-based, which is fine for a technical user. But a GUI would open it up to non-technical users and reduce misuse (e.g., copy-pasting passwords into the wrong place).

Backup & Portability
Currently, vaults are tied to a specific local directory. An encrypted import/export feature would allow users to migrate their data securely between devices, or back up to cloud storage they control.

Hardware-backed security (YubiKey, TPM)
Integrating hardware tokens for encryption/unlocking would significantly raise the bar for attackers, I'd like to try my hand at using YubiKey if possible.

Biometric unlock
Combining biometrics (fingerprint) with the master password would add to the user's convenience.

Account recovery / forgotten master password
Right now, there is no way to recover access if the master password is forgotten. But in practice, this is a really hard for users to deal with. I'd like to explore secure recovery mechanisms, possibly with secret sharing or a trusted contact model. I also don't have anything that deletes vault files if someone forgets their master password.

logs / vault change history
Right now, there's no visibility into when entries were added, edited, or deleted, and JagzPass doesn't detect if an old vault file is restored. A malicious user with access to the filesystem could roll back the vault to a previous state without any warning. I'd like to maybe implement a log or embed a last_modified timestamp inside the encrypted vault. This change is definitely doable but I've ran out of time.

Password aging / rotation reminders
There's no expiration or rotation mechanism for stored credentials. A future version could track password age and nudge users when something hasn't been updated in a while.

Per-credential encryption
Currently, a new salt is used per vault encryption, but all credentials are encrypted together. I'd like to try switching to per-credential salting and encryption, so that each password has its own key derivation and ciphertext.
On-demand decryption
At the moment, everything is decrypted at login. I'd prefer to explore a model where credentials are only decrypted when specifically requested, minimizing exposure if the session is hijacked or left idle.

Conclusion and Reflection

JagzPass gave me a much clearer understanding of what building secure software involves. Before this, I thought of security mostly as picking strong algorithms that can't be brute-forced. Now I see how much of it is about anticipating edge cases, bad configurations, and user mistakes.

Throughout this process I found myself asking rhetorical questions, hypothetical ways in which my program could be breached. These questions helped me make my decisions while working on the program, and they ultimately stemmed from my research into the algorithms I use and security incidents, along with my conversations with my tutor. While I did jump into the project hoping to gain python skills, my key take away from it has been how to "think" while creating a security related program.

There were frustrating moments throughout the project. I had bugs in my code and also had trouble finding benchmarks to use for my KDF research but I did overcome them. I would have loved to address the open vulnerabilities that I do recognise now, but I do think I've hit a wall with my lack of technical skills with python or systems. JagzPass is not perfect yet but the threats that I did choose to address are clear, and the implementation decisions match them.