Project Report: FileCrypt

1. Introduction

The goal of this project was to simulate a complete secure development lifecycle. This involved designing and building a secure file storage application from the ground up, identifying a critical vulnerability within it, demonstrating a practical exploit of that flaw, and finally, proposing a robust security patch to neutralize the threat. The resulting application, FileCrypt, serves as a practical case study in multi-layered security design.

2. Phase 1: Initial Design & Core Functionality

The initial version of FileCrypt was built as a local desktop application using Python and the `ttkbootstrap` library for the graphical user interface. Its security was founded on four core cryptographic primitives:

- **PBKDF2:** Used to securely hash user passwords. It combined the user's password with a unique, randomly generated salt, making it resistant to common password-cracking techniques.
- **RSA (2048-bit):** Used for asymmetric key management. Each user was issued a public/private key pair upon registration. The private key, protected by a user-created password, was saved locally by the user, while the public key was stored in the user database.
- **AES-256:** Used for symmetric file encryption. Its speed and security make it ideal for encrypting the contents of the files themselves.
- **SHA-256:** Used for a simple integrity check of the encrypted file content.

The application employed a **hybrid encryption scheme**: when a user uploaded a file, it was encrypted with a new, single-use AES key. That AES key was then encrypted with the user's public RSA key. This process ensured that only the owner of the corresponding private key could ever decrypt the file.

All application metadata—the user list, their salted password hashes, public keys, and the index of encrypted files—was stored in local JSON files (`users.db.json` and `files.db.json`).

3. Phase 2: The Flaw - A Critical Oversight

While the encryption of the file *content* was secure, the application had a critical architectural flaw: **it implicitly trusted its own metadata files.**

The `files.db.json` file, which stored information about which user owned which file, was a simple, unencrypted text file. The application had no mechanism to verify the integrity of this database file itself. Any program with standard user permissions could read, modify, or delete this file, and the FileCrypt application would have no way of knowing it had been tampered with. This single point of failure became the target for our attack.

4. Phase 3: The Exploit - Data Availability Attack

To demonstrate the vulnerability, a separate "attacker" application named `FileStealer` was used. This tool was designed to automate a **Data Availability Attack**, a form of **Denial of Service (DoS)**, by directly manipulating the database.

The Attack Steps:

1. **Setup:** A victim user account was created in FileCrypt. The user logged in and uploaded a sensitive file.
2. **Execution:** The `FileStealer` tool was run on the same computer, locating the application's `files.db.json` database.
3. **Manipulation:** The tool read the JSON data, identified all file entries belonging to the victim user, and **deleted them** from the database before saving the modified file.
4. **Impact:** When the victim user logged back into FileCrypt, their file list was empty. The application, reading from the tampered database, had no record of their files, making them permanently inaccessible through the application. The attacker successfully made the service (access to files) unavailable to the legitimate user.

5. Phase 4: The Proposed Fix - Digital Signatures

To make FileCrypt invulnerable to this type of metadata tampering, a robust cryptographic control must be implemented: **Digital Signatures**.

The proposed patch involves the following changes:

1. **Unique File IDs:** To prevent file overwriting, each uploaded file will be given a unique internal ID, for example, by appending a timestamp to the filename.

2. **Signing on Upload:** The user will be required to load their private key to upload a file. The application will then use this key to create a cryptographic signature of the file's essential metadata (its unique ID and owner). This signature will be stored alongside the file's entry in the database.
3. **Verification on Login:** When a user logs in, the application will use that user's public key to verify the signature of every file listed under their name. If a signature is invalid—meaning the metadata has been altered or deleted and re-inserted incorrectly—the file is considered tampered with and will not be displayed.

This digital signature system makes the database tamper-proof. An attacker can no longer delete an entry and have it go unnoticed, nor can they successfully re-assign ownership, as they cannot forge a valid signature without the victim's private key.

6. Conclusion

This project successfully demonstrated the complete secure development lifecycle. The initial version of FileCrypt was functional and followed good practices for encrypting data content, but a subtle flaw in its trust model left it vulnerable to a significant Denial of Service attack. By developing a tool to exploit this flaw, we were able to prove the severity of the risk. The final proposed patch, using digital signatures, highlights a critical security principle: one must not only protect the data itself but also the integrity and authenticity of the metadata surrounding it.

Tools used :

IDE - Visual Studio Code

AI support : Gemini 2.5 Pro, Grok 3, ChatGPT 3.5.