

## Students in the group LAB\_13\_CMAC2526

NAME	SURNAME	e-mail
Kevin	Bassett	100578348@alumnos.uc3m.es
Matthew	Chin	100578279@alumnos.uc3m.es
Jiho	Lee	100578296@alumnos.uc3m.es

## Code sharing option

Google Folder

## High-level description of the app

Our Encrypted File Storage System is a secure local file vault that lets users store sensitive files with end-to-end encryption. The main features are user registration/login, encrypted file uploads, and secure downloads with decryption. It's designed for individual users who want to keep personal files safe on their own machine.

The app has three main data flows. During registration, the user provides a username and password, and the system generates an RSA key pair, hashes the password with Argon2id, and encrypts the private key using PBKDF2 + AES-GCM before saving everything to JSON. During upload, the system generates a random AES-256 key, encrypts the file with AES-GCM, wraps the key with RSA-OAEP using the user's public key, and stores the ciphertext. During download, the user provides their password to decrypt their private key, which unwraps the file key, which then decrypts the file.

The protected data includes: password hashes (protected by Argon2id), RSA private keys (encrypted with PBKDF2 + AES-GCM), file contents (encrypted with AES-256-GCM), and wrapped file keys (protected by RSA-OAEP). Public keys and metadata don't need protection since they're either public or integrity-protected by the GCM tag.

## Technical description

### Modules

The codebase is split into three main parts: accounts/ for user authentication, storage/ for file encryption, and gui.py for the interface.

In the accounts package: hashing.py handles password hashing with Argon2id. models.py defines the User dataclass with fields for credentials and crypto material (public key, encrypted private key, nonce, salt). storage.py provides JSON persistence with atomic writes to prevent corruption. manager.py ties everything together with register(), authenticate(), and decrypt\_private\_key() methods.

In the storage package: models.py defines FileMetadata with fields for the file info plus encryption parameters (wrapped\_key, nonce, tag). file\_manager.py implements the crypto

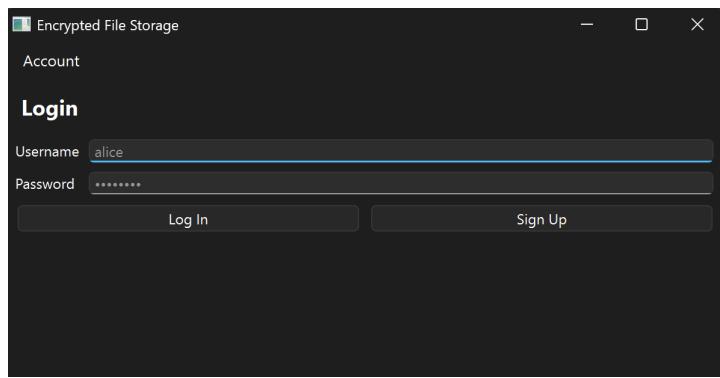
functions—`encrypt_bytes_aes_gcm()`, `wrap_key_rsa_oaep()`, and the main `upload_file()` and `download_file()` operations.

The `gui.py` module uses PySide6 to create a `LoginPage` for auth, a `StoragePage` for file management, and a `MainWindow` that switches between them.

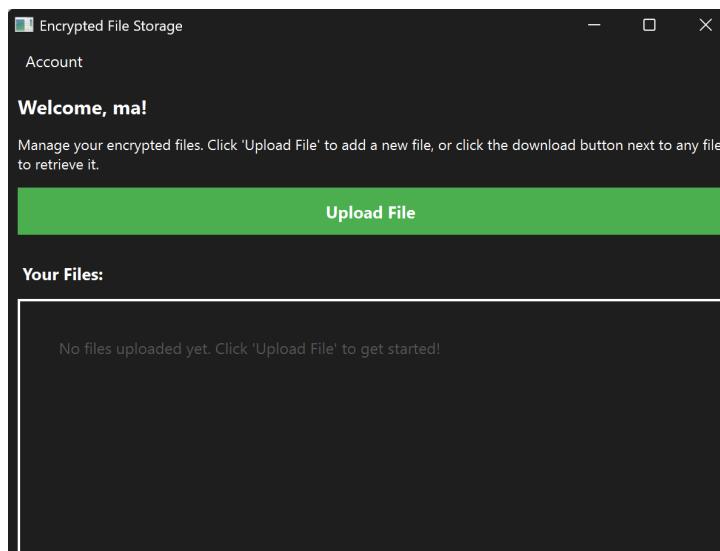
## Main functionalities

**Registration:** User enters username/password and clicks Sign Up. The system generates RSA-2048 keys, hashes passwords with Argon2id, encrypts the private key with PBKDF2 (200k iterations) + AES-GCM, and saves to `users.json`.

- Login: User enters credentials and clicks Log In. System verifies password against stored Argon2id hash. If valid, the user proceeds to the storage page.

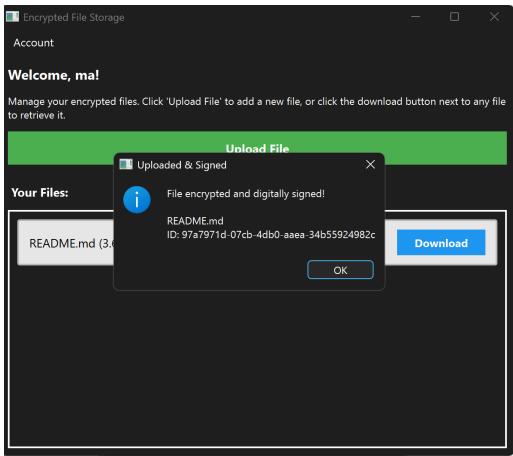


- Upload: User selects a file. System generates random AES-256 key, encrypts file with AES-GCM, wraps the key with RSA-OAEP, and stores the ciphertext blob plus metadata in the vault.



- Download: User picks a file and enters their password. System decrypts the private key, unwraps the file key with RSA-OAEP, decrypts the file with AES-GCM, and saves the plaintext to Downloads.

- Logout: Clears the session and returns to the login screen.



## Byte-like/text-like data encoding/decoding

Since JSON can't store raw bytes, we use Base64 encoding to convert all binary crypto data to text for storage.

**Encoding** (bytes → text): `base64.b64encode(data).decode("ascii")`

**Decoding** (text → bytes): `base64.b64decode(text)`

We apply this in four places:

1. Registration (`accounts/manager.py` lines 57-60): Encode the public key, encrypted private key, nonce, and salt before storing in the User object.
2. Private key decryption (`accounts/manager.py` lines 83-85): Decode the salt, nonce, and encrypted private key before running PBKDF2 and AES-GCM decryption.
3. File upload (`storage/file_manager.py` lines 159-162): Encode the wrapped key, nonce, and tag before storing in FileMetadata.
4. File download (`storage/file_manager.py` lines 216-222): Decode the wrapped key, nonce, and tag before RSA unwrapping and AES decryption.

We chose Base64 because JSON requires text, it's consistent across the whole codebase, and it's a standard format that's easy to debug.

## User authentication

Users register by entering a username and password in the GUI. When they click Sign Up, the system first checks if the username is taken, then hashes the password using Argon2id via the `argon2-cffi` library. We chose Argon2id because it's the winner of the Password Hashing Competition and is memory-hard, meaning it's resistant to GPU-based brute force attacks.

The password hash is stored in `users.json` alongside other user data. During login, the system

retrieves the stored hash and uses Argon2's verify function to check if the entered password matches. The verify function handles all the complexity of extracting the salt and parameters from the stored hash.

#### Password hashing function (accounts/hashing.py)

```
from argon2 import PasswordHasher

class SimpleHasher:
    def __init__(self):
        self._ph = PasswordHasher()

    def hash(self, password: str) -> str:
        return self._ph.hash(password)

    def verify(self, stored_hash: str, password: str) -> bool:
        try:
            return self._ph.verify(stored_hash, password)
        except VerifyMismatchError:
            return False
```

#### Authentication logic (accounts/manager.py):

```
def authenticate(self, username: str, password: str) -> Optional[User]:
    user = self.storage.get_user_by_username(username)
    if not user:
        return None
    if not self.hasher.verify(user.pwd_hash, password):
        return None
    return user
```

#### Example users.json content:

```
{
  "users": [
    {
      "user_id": "a1b2c3d4-...",
      "username": "alice",
      "pwd_hash": "$argon2id$v=19$m=65536,t=3,p=4$...",
      "created_at": "2025-01-15T10:30:00Z",
      "public_key": "LS0tLS1CRUdJTi...",
      "enc_private_key": "base64-encrypted-key...",
      "enc_private_key_nonce": "base64-nonce...",
      "enc_private_key_salt": "base64-salt..."
    }
  ]
}
```

The GUI flow: user fills in username/password → clicks "Sign Up" or "Log In" → system shows success/error message → on success, transitions to the storage page.

## Data encryption and authentication

Files are encrypted using AES-256-GCM, an authenticated encryption algorithm that provides both confidentiality and integrity in one operation. We chose AES-GCM because it's the industry standard for authenticated encryption, it's fast, and the authentication tag detects any tampering with the

ciphertext.

When uploading a file, the system generates a random 256-bit AES key, encrypts the file with AES-GCM using a random 12-byte nonce, and stores the ciphertext blob. The GCM mode produces a 16-byte authentication tag that's stored alongside the ciphertext. During download, decryption will fail if any bit of the ciphertext has been modified.

**AES-GCM encryption (storage/file\_manager.py lines 20-31):**

```
def encrypt_bytes_aes_gcm(plaintext: bytes, key: bytes) -> Tuple[bytes, bytes, bytes]:
    nonce = os.urandom(12)
    aesgcm = AESGCM(key)
    ct_with_tag = aesgcm.encrypt(nonce, plaintext, None)
    ciphertext, tag = ct_with_tag[:-16], ct_with_tag[-16:]
    return ciphertext, nonce, tag
```

**AES-GCM decryption (storage/file\_manager.py lines 34-41):**

```
def decrypt_bytes_aes_gcm(ciphertext: bytes, nonce: bytes, tag: bytes, key: bytes) -> bytes:
    aesgcm = AESGCM(key)
    return aesgcm.decrypt(nonce, ciphertext + tag, None)
```

**Example vault/alice/index.json:**

```
{
  "files": [
    {
      "file_id": "71904587-4116-...",  

      "owner": "alice",  

      "filename": "secret.txt",  

      "stored_name": "f312c1eed792....bin",  

      "size": 1024,  

      "created_at": "2025-01-15T12:00:00Z",  

      "wrapped_key": "rRzKe7Ll+xMbjC+7amkvhf...",  

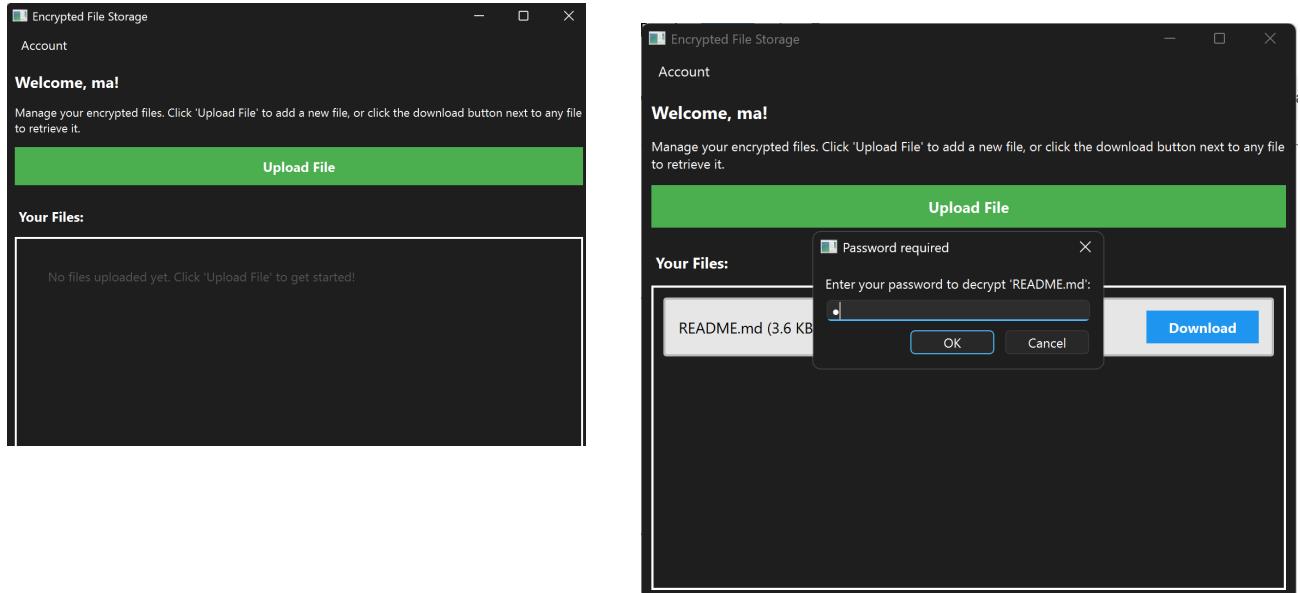
      "wrap_algo": "rsa-oaep-sha256",  

      "nonce": "8ACLgh0ueSsdjvm6",  

      "tag": "csKIqE/PRUtW3Tn/Is1Rcg=="
    }
  ]
}
```

The actual encrypted file is stored as a binary blob (e.g., vault/alice/f312c1eed792....bin).

GUI flow: User clicks "Upload" → selects file → system encrypts and stores it → success message with file ID. For download: user selects file → enters password → system decrypts → file saved to Downloads.



## Symmetric key management

Our app uses two types of symmetric keys:

Per-file AES-256 keys – Used to encrypt each file

Password-derived AES-256 keys – Used to encrypt the user's RSA private key

Per-file keys are generated using `os.urandom(32)` which provides 32 cryptographically random bytes (256 bits). A new key is generated for every file upload, so compromising one file's key doesn't affect other files. These keys are never stored directly—instead, they're wrapped with RSA-OAEP and stored in the file metadata.

**Key generation for files (storage/file\_manager.py line 149):**

```
file_key = os.urandom(32) # AES-256
```

**Key wrapping with RSA-OAEP (storage/file\_manager.py lines 44-56):**

```
def wrap_key_rsa_oaep(file_key: bytes, public_key_pem: bytes) -> bytes:
    public_key = serialization.load_pem_public_key(public_key_pem)
    return public_key.encrypt(
        file_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )
```

Password-derived keys are generated using PBKDF2-HMAC-SHA256 with 200,000 iterations and a random 16-byte salt. This key is used to encrypt the user's RSA private key at rest. We chose PBKDF2 because it's a standard key derivation function that makes brute-force attacks expensive.

### Key derivation for private key encryption (accounts/manager.py lines 41-48):

```
salt = os.urandom(16)
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=200_000,
)
aes_key = kdf.derive(password.encode("utf-8"))
```

Who creates/accesses keys:

- File keys: Created automatically during upload, accessed during download after password verification
- Derived keys: Created during registration, re-derived during any operation requiring the private key
- RSA keys: Created during registration, public key stored plaintext, private key stored encrypted

Key storage:

- File keys are stored wrapped (RSA-encrypted) in vault/<user>/index.json
- The salt and nonce for private key encryption are stored in users.json
- No plaintext symmetric keys are ever written to disk

Key rotation: Not currently implemented. Each file has its own unique key, so rotating would require re-encrypting all files.

## Asymmetric key management

Every registered user gets an RSA-2048 key pair. The keys are generated automatically during registration—users don't interact with key generation directly.

The keys have two purposes:

- Encryption (RSA-OAEP): The public key wraps per-file AES keys; the private key unwraps them during download
- Digital signatures (RSA-PSS): The private key signs files before encryption; the public key (via certificate) verifies signatures during download

### Key generation during registration (accounts/manager.py lines 28-38):

```
from cryptography.hazmat.primitives.asymmetric import rsa

# Generate RSA-2048 key pair
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()

# Serialize public key
public_pem = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo,
)

# Serialize private key (unencrypted, will be encrypted with AES-GCM)
private_bytes = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption(),
)
```

We chose RSA-2048 because it provides adequate security (equivalent to ~112 bits symmetric), is widely supported, and works for both encryption and signatures.

## Loading and serializing asymmetric keys/public key certificates

Keys are serialised in PEM format with PKCS8 for private keys and SubjectPublicKeyInfo for public keys. PEM is base64-encoded with headers, making it human-readable and standard across tools.

Serialisation format:

- Public key: PEM + SubjectPublicKeyInfo → Base64 encoded for JSON storage
- Private key: PEM + PKCS8 → Encrypted with AES-GCM → Base64 encoded for JSON storage

Loading the public key (accounts/manager.py lines 74-76):

```
def public_key_pem(self, user: User) -> bytes:  
    return base64.b64decode(user.public_key)
```

Decrypting and loading the private key (accounts/manager.py lines 78-95):

```
def decrypt_private_key(self, user: User, password: str) -> bytes:  
    salt = base64.b64decode(user.enc_private_key_salt)  
    nonce = base64.b64decode(user.enc_private_key_nonce)  
    enc_priv = base64.b64decode(user.enc_private_key)  
  
    # Re-derive the AES key from password  
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=200_000)  
    aes_key = kdf.derive(password.encode("utf-8"))  
  
    # Decrypt and return PEM bytes  
    aescm = AESGCM(aes_key)  
    return aescm.decrypt(nonce, enc_priv, None)
```

Example of stored public key (decoded from Base64):

```
-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBCgKCAQEA2K3S8...  
-----END PUBLIC KEY-----
```

Loading keys for cryptographic operations (storage/file\_manager.py):

```
# Load public key for wrapping  
public_key = serialization.load_pem_public_key(public_key_pem)  
  
# Load private key for unwrapping  
private_key = serialization.load_pem_private_key(private_key_pem, password=None)
```

## Digital signatures

Files are signed by the uploader using RSA-PSS with SHA-256 before encryption. This ensures the recipient can verify that the file hasn't been tampered with and was uploaded by the claimed user.

What gets signed: The original plaintext file content (before encryption)

When: During upload, after reading the file but before AES-GCM encryption

Storage: Signature is Base64-encoded and stored in the file's metadata along with the signer's certificate

### **[Signing code \(crypto/signatures.py\):](#)**

```
from cryptography.hazmat.primitives.asymmetric import padding

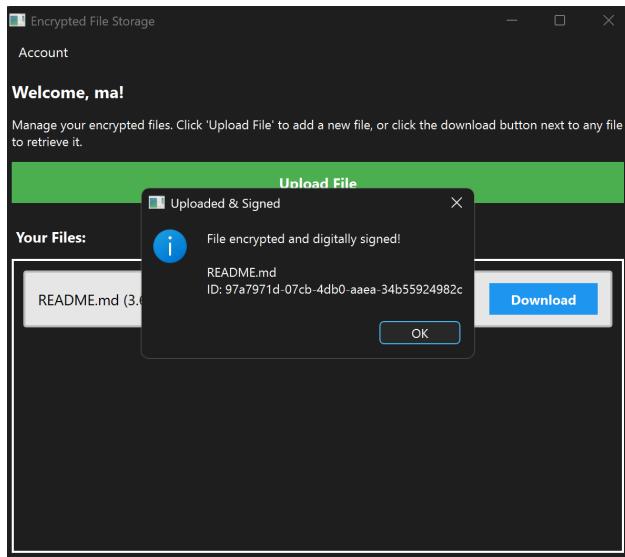
def sign_data(data: bytes, private_key_pem: bytes) -> bytes:
    private_key = serialization.load_pem_private_key(private_key_pem, password=None)
    signature = private_key.sign(
        data,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature
```

### **[Verification code \(crypto/signatures.py\):](#)**

```
def verify_signature(data: bytes, signature: bytes, public_key_pem: bytes) -> bool:
    try:
        public_key = serialization.load_pem_public_key(public_key_pem)
        public_key.verify(
            signature, data,
            padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH),
            hashes.SHA256()
        )
        return True
    except Exception:
        return False
```

We chose RSA-PSS because it's more secure than PKCS#1 v1.5 padding (provably secure under certain assumptions) and is recommended for new applications. SHA-256 provides collision resistance.

Verification flow: During download, the system decrypts the file, loads the signer's certificate, extracts the public key, and verifies the signature matches the decrypted plaintext. If verification fails, the user is warned that the file may have been tampered with.



## **Asymmetric encryption / hybrid encryption**

We use hybrid encryption: RSA-OAEP encrypts a random AES key, and AES-GCM encrypts the actual file. This combines the best of both worlds—asymmetric key management with symmetric performance.

Why hybrid? RSA can only encrypt small amounts of data (< 245 bytes for RSA-2048 with OAEP). Files can be gigabytes. So we generate a random AES key, encrypt the file with AES (fast), and encrypt just

the AES key with RSA (secure key exchange).

#### RSA-OAEP key wrapping (storage/file\_manager.py lines 44-56):

```
def wrap_key_rsa_oaep(file_key: bytes, public_key_pem: bytes) -> bytes:
    public_key = serialization.load_pem_public_key(public_key_pem)
    return public_key.encrypt(
        file_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )
```

#### RSA-OAEP key unwrapping (storage/file\_manager.py lines 59-71):

```
def unwrap_key_rsa_oaep(wrapped_key: bytes, private_key_pem: bytes) -> bytes:
    private_key = serialization.load_pem_private_key(private_key_pem, password=None)
    return private_key.decrypt(
        wrapped_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None,
        ),
    )
```

We chose RSA-OAEP because it's the recommended padding scheme for encryption (unlike PKCS#1 v1.5, which has known vulnerabilities). SHA-256 is used for both the hash and MGF1 mask generation.

Data flow: Upload: plaintext → AES-GCM(random key) → ciphertext; random key → RSA-OAEP(public key) → wrapped key. Download: wrapped key → RSA-OAEP(private key) → random key; ciphertext → AES-GCM(random key) → plaintext.

## Public key certificates and mini-PKI

We use self-signed X.509 certificates generated automatically during user registration. Each user's certificate binds their public key to their username.

#### Certificate generation (crypto/pki.py):

```
from cryptography import x509
from cryptography.x509.oid import NameOID

def generate_self_signed_certificate(private_key_pem: bytes, common_name: str, validity_days: int = 365) -> bytes:
    private_key = serialization.load_pem_private_key(private_key_pem, password=None)

    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "ES"),
        x509.NameAttribute(NameOID.COMMON_NAME, common_name),
    ])

    cert = x509.CertificateBuilder().subject_name(subject).issuer_name(issuer).public_key(
        private_key.public_key()
    ).serial_number(x509.random_serial_number()).not_valid_before(
        datetime.datetime.utcnow()
    ).not_valid_after(
        datetime.datetime.utcnow() + datetime.timedelta(days=validity_days)
    ).sign(private_key, hashes.SHA256())

    return cert.public_bytes(serialization.Encoding.PEM)
```

Mini-PKI tools: We also implemented scripts for a full PKI workflow:

- `pki_tools/setup_ca.py` – Creates a Root CA with a 4096-bit RSA key
- `pki_tools/generate_user_csr.py` – Generates Certificate Signing Requests for users
- `pki_tools/sign_csr.py` – CA signs CSRs to issue user certificates

#### Certificate verification ([crypto/pki.py](#)):

```
def verify_certificate_signature(cert_to_verify, issuer_public_key_pem: bytes) -> bool:
    issuer_public_key = serialization.load_pem_public_key(issuer_public_key_pem)
    issuer_public_key.verify(
        cert_to_verify.signature,
        cert_to_verify.tbs_certificate_bytes,
        padding.PKCS1v15(),
        cert_to_verify.signature_hash_algorithm
    )
    return True
```

## Other aspects

Atomic file writes: Both `users.json` and `vault` indexes use atomic writes—data is written to a temp file first, then renamed. This prevents corruption if the app crashes mid-write.

Username canonicalization: Usernames are lowercased and trimmed to prevent "Alice" and "alice" from being different accounts.

Error handling: The GUI catches crypto exceptions (wrong password, tampered files) and shows user-friendly error messages instead of crashing.

File organisation: Each user has their own `vault` subdirectory, providing logical separation even though crypto already isolates users.

## Conclusions

This project taught us how real-world encryption systems work in practice. The biggest challenge was understanding how all the pieces fit together—password hashing, key derivation, symmetric encryption, asymmetric encryption, and digital signatures each solve different problems, but combining them correctly is tricky.

We learned why hybrid encryption exists (RSA is too slow for large files), why authenticated encryption matters (AES-GCM vs plain AES), and why key derivation functions like PBKDF2 are necessary (passwords aren't random enough to use directly as keys).

The most enjoyable part was seeing the system actually work—uploading a file, seeing it stored as encrypted garbage, then downloading and getting the original back. The most frustrating part was debugging encoding issues (bytes vs strings, Base64 everywhere) and understanding the cryptography library's API.

Overall, building this app definitely helped us understand the course material better than just reading about it. Implementing encryption yourself makes you appreciate both how powerful these tools are and how easy it is to make mistakes if you don't understand what each component does.