МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ
ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ
ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Звіт
з виконання завдань практичної роботи №3 на тему
«Реалізація основних асиметричних криптосистем»
з кредитного модуля «Методи реалізації криптографічних механізмів»

Виконали:
студенти гр. ФБ-31мн
Журибіда Ю.Б.
Швець М.К.
Шостак А.А.

Київ 2024

Зміст

Вступ

Мета роботи: дослідження можливостей побудови загальних та спеціальних криптографічних протоколів за допомогою асиметричних криптосистем.

Ми будемо реалізовувати завдання для підгрупи 3А — IT-систему Web-сервісу електронного цифрового підпису.

Теоретичні відомості

Електронний цифровий підпис (ЕЦП) — це криптографічний механізм, який використовується для забезпечення автентичності та цілісності електронних документів. ЕЦП гарантує, що документ не був змінений після підписання і підтверджує особу підписанта.

Основою для створення і перевірки ЕЦП є асиметричне шифрування, яке використовує пару ключів: приватний і публічний:

- приватний ключ використовується для створення підпису. Зберігається в таємниці і доступний лише власнику;
- публічний ключ, в свою чергу, використовується для перевірки підпису. Він може бути доступний будь-кому.

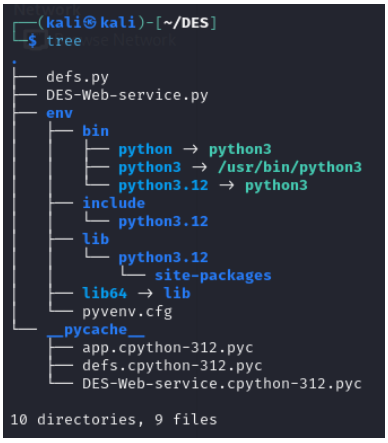
Пройдемося по окремим параметрам та методам, які використаємо для реалізації завдання:

- RSA (Rivest-Shamir-Adleman) — це один з найпоширеніших алгоритмів асиметричного шифрування, який використовується для створення і перевірки ЕЦП. Він базується на математичній складності факторизації великих чисел;
- PEM — це текстовий формат, що використовується для зберігання криптографічних ключів та сертифікатів. Він містить дані у вигляді тексту, закодованого з використанням Base64, та заголовки, які ідентифікують тип даних;
- PSS — це схема доповнення, яка використовується для підпису даних в RSA. Вона забезпечує високий рівень безпеки завдяки використанню випадкових значень (сіль) та хешування;
- SHA-256 (Secure Hash Algorithm 256-bit) — це криптографічний алгоритм хешування, який створює унікальний фіксований розмір хеш для даних. Він використовується для створення хешу документа перед підписанням, що забезпечує цілісність даних.

Для спрощення реалізації веб-сервісу ми вирішили використовувати FastAPI. FastAPI — це сучасний веб-фреймворк для Python, який дозволяє швидко створювати високопродуктивні веб-додатки. Він підтримує автоматичну генерацію [якої/не якої :)] документації та перевірку типів даних та надає можливість легкого тестування сервісу.

Практична реалізація

Структура Web-сервісу виглядає наступним чином:



Файл «defs.py» містить у собі підключення допоміжних бібліотек та визначення функцій, що будуть використовуватися в процесі роботи сервісу. Бібліотеки, що використовуються:

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization
```

Перша визначена функція — функція генерації ключей:

```
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )
    public_key = private_key.public_key()

    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    return private_pem, public_pem
```

Що робить ця функція:

1) генерує ключ довжиною 2048 бітів з використання публічної експоненти рівною 65537;
2) за допомогою методу private_key.public_key() отримую публічний ключ з приватного;
3) за допомогою методу private_key.private_bytes отримую байтовий рядок приватного ключа, де:
    a. параметр encoding=serialization.Encoding.PEM встановлює кодування для серіалізації у вигляді PEM-у (текстовий формат для зберігання криптографічних об'єктів);
    b. параметр format=serialization.PrivateFormat.PKCS8 встановлює формат для серіалізації у вигляді PKCS8 (стандарт для зберігання приватних ключів);
    c. параметр encryption_algorithm=serialization.NoEncryption() визначає алгоритм шифрування. В даному випадку (оскільки це все просто демонстрація роботи алгоритмів) шифрування не використовується.
4) за допомогою методу public_key.public_bytes() отримую байт стрічку публічного ключа, де:
    a. параметр encoding=serialization.Encoding.PEM встановлює кодування для серіалізації у вигляді PEM-у;
    b. параметр format=serialization.PublicFormat.SubjectPublicKeyInfo визначає формат для серіалізації публічного ключа;
5) повертає серіалізовані приватний (private_pem) і публічний (public_pem) ключі у форматі PEM.

Простіше кажучи, функція generate_keys() генерує пару ключів RSA (приватний та публічний), серіалізує їх у форматі PEM і повертає у вигляді байт-строк. Цей процес дозволяє легко зберігати і використовувати ключі для криптографічних операцій, таких як шифрування, розшифрування, підписання та верифікація.

Друга визначена функція — функція підписання документів:

```
def sign_document(document, private_key_pem):
    private_key = serialization.load_pem_private_key(
        private_key_pem,
        password=None,
    )
    signature = private_key.sign(
        document,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )
    return signature
```

Що робить ця функція:

1) приймає на вхід два параметри:
   a. document — документ, який необхідно підписати;
   b. private_key_pem – приватний ключ у форматі PEM, за допомогою якого буде створено підпис;
2) за допомогою методу serialization.load_pem_private_key завантажує приватний ключ з байтової стрічки у форматі PEM, де:
   a. аргумент private_key_pem безпосередньо містить приватний ключ у форматі PEM;
   b. аргумент password=None вказує, що приватний ключ не захищено паролем.
3) за допомогою методу private_key.sign підписує документ, де:
   a. document — документ, який необхідно підписати;
   b. метод padding.PSS() вказує схему доповнення PSS для підпису, де:
      i. mgf=padding.MGF1(hashes.SHA256()) — функція генерації маски, що використовує алгоритм хешування SHA-256;
      ii. salt_length=padding.PSS.MAX_LENGTH — максимальна довжина випадкової послідовності (сіль), що додається до хешу;
   c. параметр hashes.SHA256() встановлює алгоритм хешування SHA-256, що використовується для створення хешу документа перед підписом.
4) повертає створений підпис у вигляді байт-строки.

Простіше кажучи, функція sign_document() використовується для підписання документа за допомогою приватного ключа. Вона завантажує приватний ключ у форматі PEM, створює підпис для заданого документа за допомогою схеми доповнення PSS та алгоритму хешування SHA-256, а потім повертає створений підпис.

Третя визначена функція — функція перевірки підписів на документах:

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

Що робить ця функція:

1) приймає на вхід три параметри:
   a. document — документ, який необхідно підписати;
   b. signature — підпис документу, який потрібно перевірити;
   c. public_key_pem – публічний ключ у форматі PEM, за допомогою якого буде перевірено підпис;
2) метод serialization.load_pem_public_key завантажує публічний ключ з байт-рядку у форматі PEM;
3) метод public_key.verify() перевіряє підпис, де:
   a. signature – підпис, який потрібно перевірити;
   b. document — документ, для якого перевіряється підпис;
   c. метод padding.PSS() вказує схему доповнення PSS для перевірки підпису, де:
      i. mgf=padding.MGF1(hashes.SHA256()) — функція генерації маски, що використовує алгоритм хешування SHA-256;
      ii. salt_length=padding.PSS.MAX_LENGTH — максимальна довжина випадкової послідовності (сіль), що додається до хешу;
   d. параметр hashes.SHA256() встановлює алгоритм хешування SHA-256, що використовується для перевірку хешу документа;
4) блок try-except використовується для обробки можливих винятків під час перевірки підпису.

Простіше кажучи, функція verify_signature() перевіряє підпис документа за допомогою публічного ключа у форматі PEM. Вона завантажує публічний ключ, перевіряє підпис, використовуючи схему доповнення PSS та алгоритм хешування SHA-256, і повертає True, якщо підпис дійсний, або False, якщо перевірка неуспішна.

Файл «DES-Web-service.py» містить у собі безпосередню програмну реалізацію веб-сервісу ЕЦП. Нас зустрічає підключення додаткових бібліотек та файлу з визначеними функціями:

```
from fastapi import FastAPI, File, UploadFile
from defs import generate_keys, sign_document, verify_signature
```

Далі ми створюємо об’єкт FastAPI, який використовується для визначення маршрутів та запуску сервісу:

```
app = FastAPI()
```

Далі ми визначаємо маршрут для POST запиту за адресою «/generate_keys»:

```
@app.post("/generate_keys")
def generate_keys_endpoint():
    private_key, public_key = generate_keys()
    return {"private_key": private_key.decode(), "public_key": public_key.decode()}
```

Тут generate_keys_endpoint() — це функція-обробник маршруту, яка:

1) викликає функцію generate_keys() для генерації приватного і публічного ключів у форматі PEM;
2) повертає згенеровані ключі у JSON-форматі.

Далі ми визначаємо маршрут для POST запиту за адресою «/sign_document»:

Тут sign_document_endpoint() — це функція-обробник для цього маршруту, яка:

1) приймає файл для підпису (file: UploadFile) та приватний ключ у форматі PEM (private_key: str);

2) заміщує символи \\n на \n у приватному ключі для правильної інтерпретації форматування (ака нормалізація ключа на мінімалках);
3) зчитує вміст файлу і підписує його за допомогою функції sign_document();
4) повертає згенерований підпис у форматі hex.

Далі ми визначаємо маршрут для POST запиту за адресою «/verify_signature»:

Функція-обробник цього маршруту — verify_signature_endpoint(), яка:

1) приймає файл для перевірки (file: UploadFile), підпис (signature: str) та публічний ключ у форматі PEM (public_key: str);
2) заміщує символи \\n на \n у публічному ключі для правильної інтерпретації форматування (ака нормалізація ключа на мінімалках);
3) зчитує вміст файлу і перевіряє підпис за допомогою функції verify_signature();
4) повертає результат перевірки (True або False) у форматі JSON.

Отже, код з цього файлу визначає веб-сервіс для генерації RSA ключів, підпису документів та перевірки підписів за допомогою FastAPI. Він забезпечує три основні маршрути:

1. /generate_keys для генерації ключів.
2. /sign_document для підпису документа.
3. /verify_signature для перевірки підпису документа.

Всі інші директорії та файли, відображені у виводі команди tree, були створені автоматично як додаткові файли середовища Python для цього проєкту.

Програмний код сервісу

defs.py

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )
    public_key = private_key.public_key()
    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    return private_pem, public_pem
```

```
def sign_document(document, private_key_pem):
    private_key = serialization.load_pem_private_key(
        private_key_pem,
        password=None,
    )
    signature = private_key.sign(
        document,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH,
        ),
        hashes.SHA256(),
    )
    return signature
def verify_signature(document, signature, public_key_pem):
    public_key = serialization.load_pem_public_key(public_key_pem)
    try:
        public_key.verify(
            signature,
            document,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH,
            ),
            hashes.SHA256(),
        )
        return True
    except Exception:
        return False
```

DES-Web-service.py

```
from fastapi import FastAPI, File, UploadFile
from defs import generate_keys, sign_document, verify_signature
app = FastAPI()
@app.post("/generate_keys")
def generate_keys_endpoint():
    private_key, public_key = generate_keys()
    return {"private_key": private_key.decode(), "public_key": public_key.decode()}
```

```
@app.post("/sign_document")
def sign_document_endpoint(file: UploadFile, private_key: str):
    private_key = private_key.replace("\\n", "\n")
    document = file.file.read()
    signature = sign_document(document, private_key.encode())
    return {"signature": signature.hex()}
@app.post("/verify_signature")
def verify_signature_endpoint(file: UploadFile, signature: str, public_key: str):
    public_key = public_key.replace("\\n", "\n")
    document = file.file.read()
    is_valid = verify_signature(document, bytes.fromhex(signature), public_key.encode())
    return {"is_valid": is_valid}
```

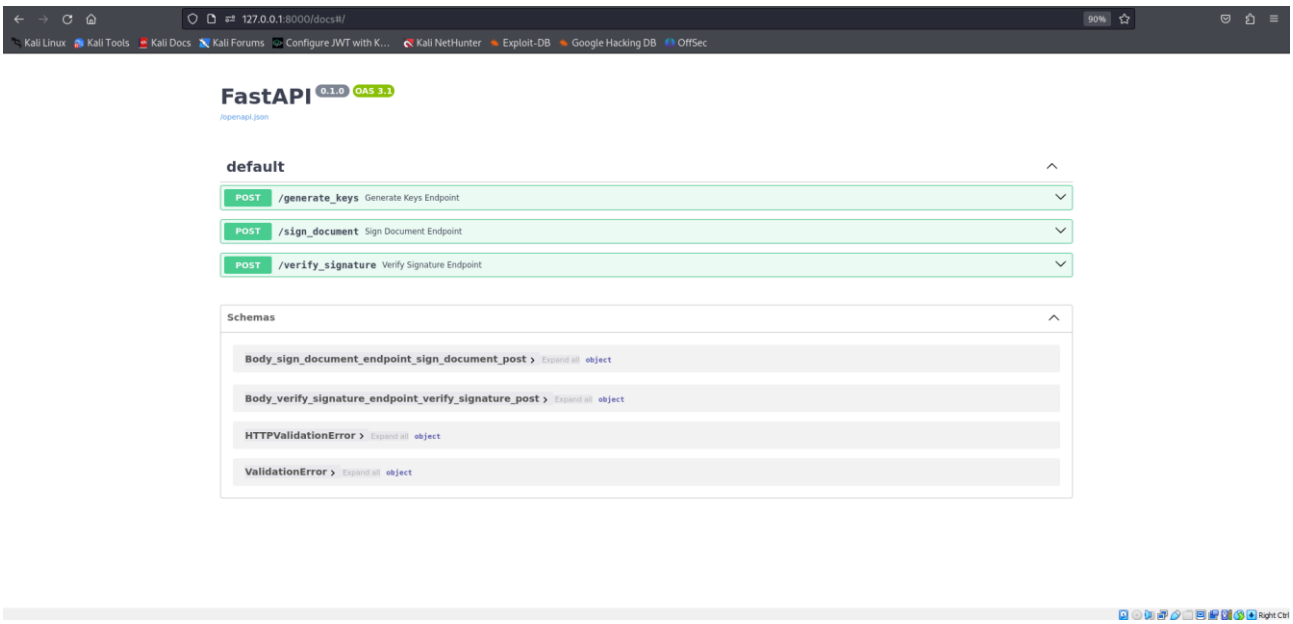
Демонстрація

Для запуску проєкту ми будемо використовувати сервер ASGI uvicorn. Він використовується для запуску веб-додатків, зокрема, написаних за допомогою FastAPI.

Запуск сервісу:

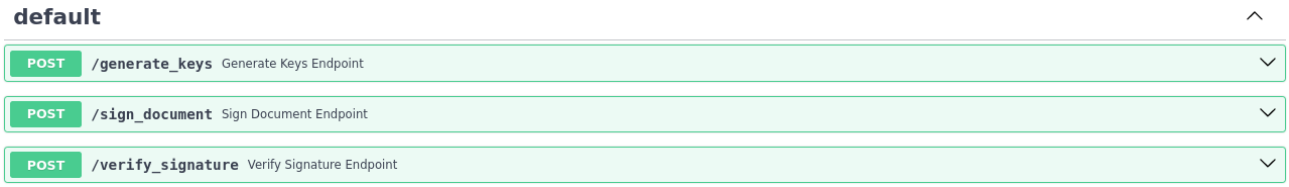
```
┌──(kali㉿kali)-[~/DES]
└─$ uvicorn DES-Web-service:app --host 0.0.0.0 --port 8000 --reload
INFO:     Will watch for changes in these directories: ['/home/kali/DES']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [38112] using StatReload
INFO:     Started server process [38114]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Головна сторінка веб-сервісу виглядає наступним чином:



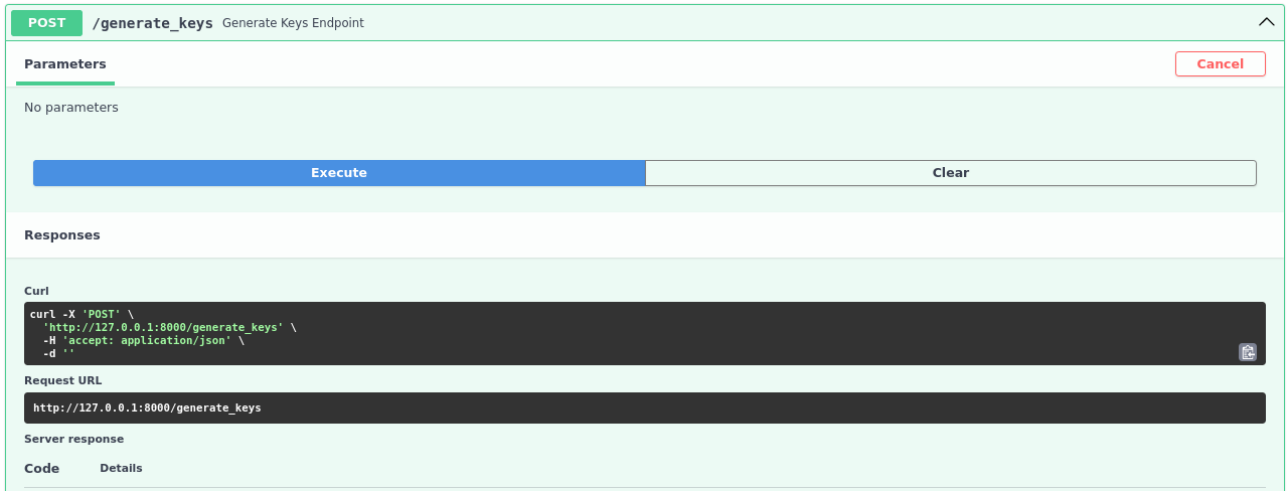
Тут ми маємо автоматично створену документацію у вигляді схем:

А також самі реалізовані функції для POST-запитів:

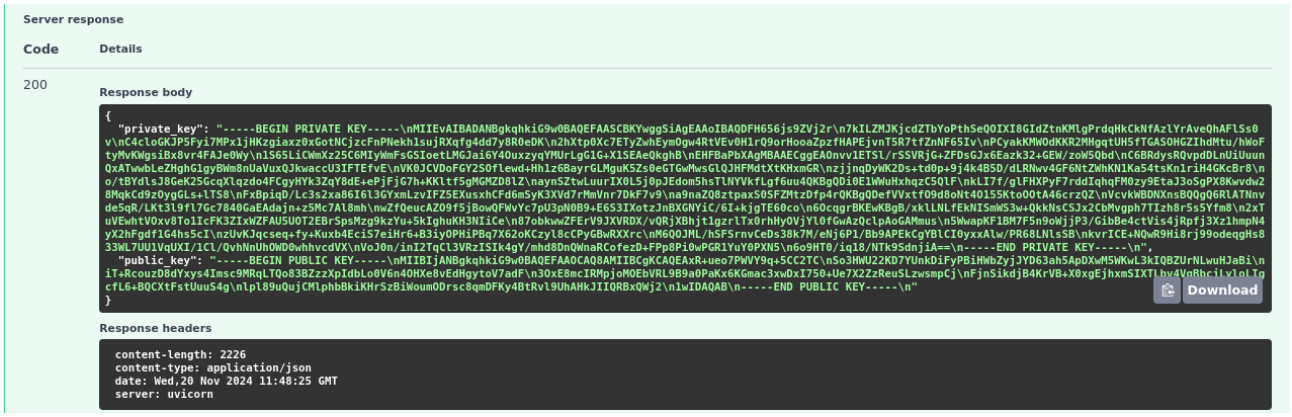


Демонстрація роботи веб-сервісу через WEB UI

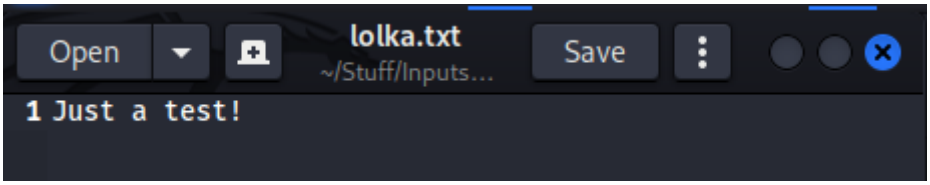
Через інтерактивну панель ініціалізуємо виконання створення ключів:



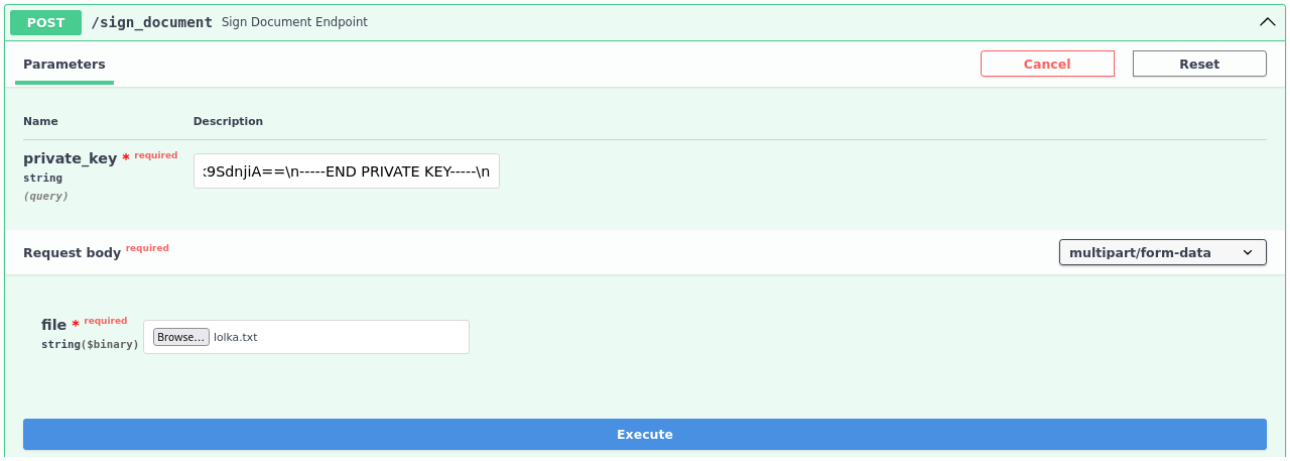
Отримуємо наступну відповідь:



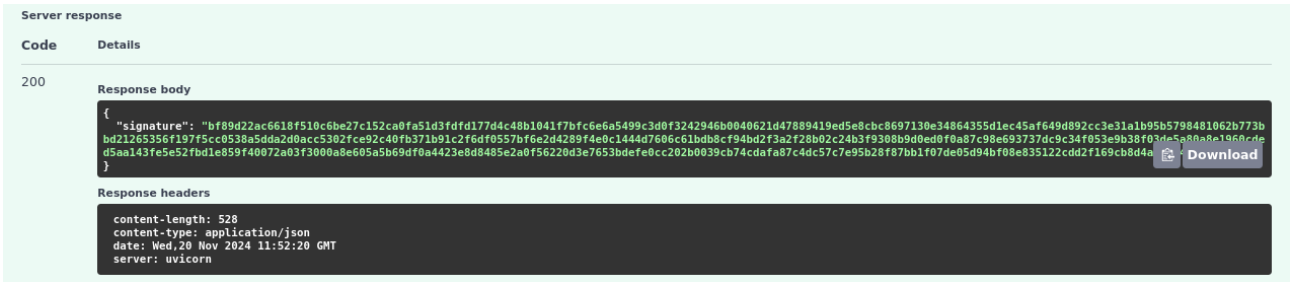
Створили файл, який хочемо підписати щоб в майбутньому підтверджувати його автентичність:



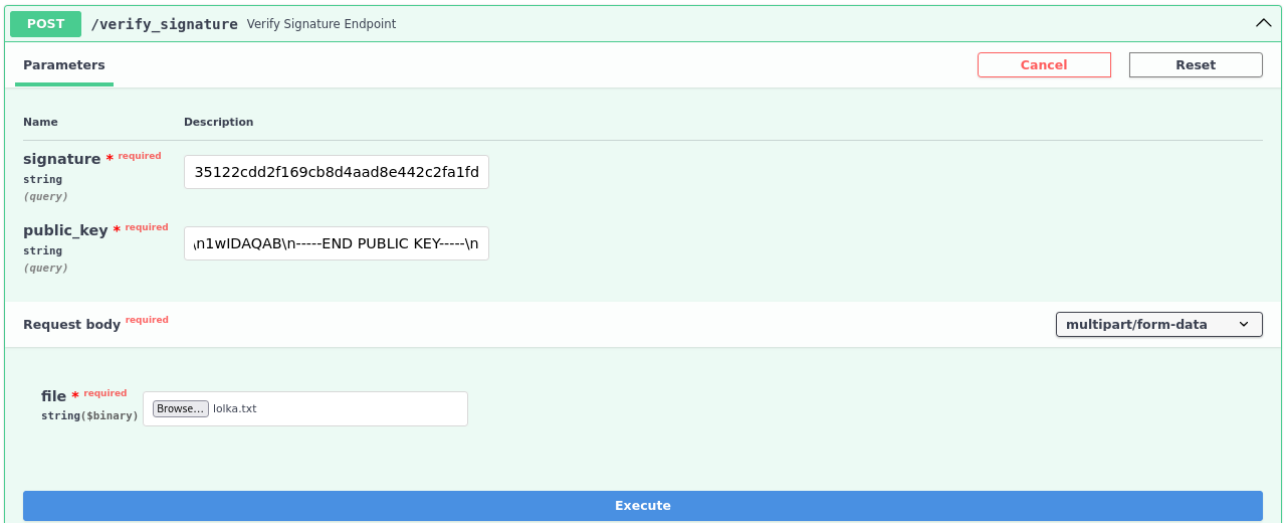
Через інтерактивну панель ініціалізуємо виконання підпису обраного файлу з нашим приватним ключем:



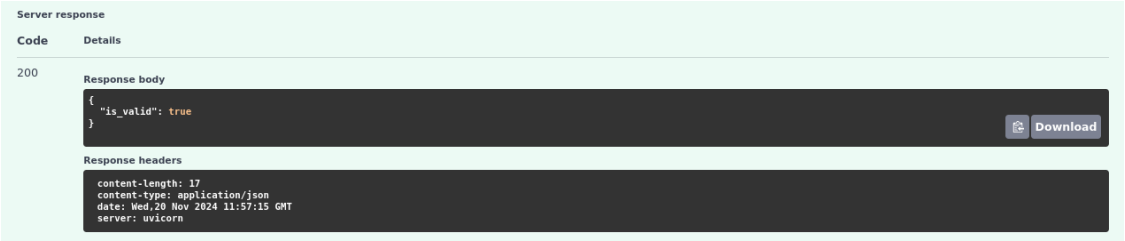
Отримуємо наступну відповідь з нашим цифровим підписом:



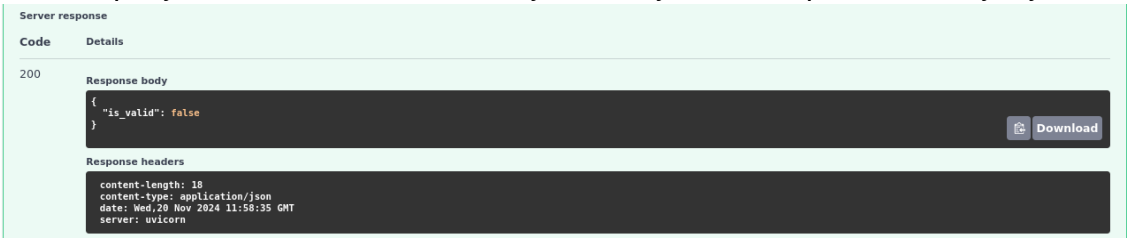
Через інтерактивну панель ініціалізуємо виконання перевірку підпису обраного файлу з публічним ключем:



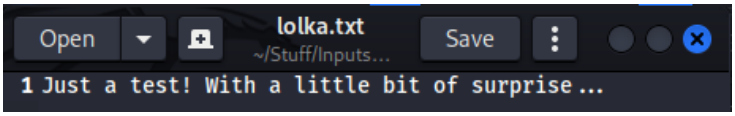
Отримали наступну відповідь, що свідчить про валідність підпису:



Спробуємо змінити 1 символ в публічному ключі. Отримали наступну відповідь:



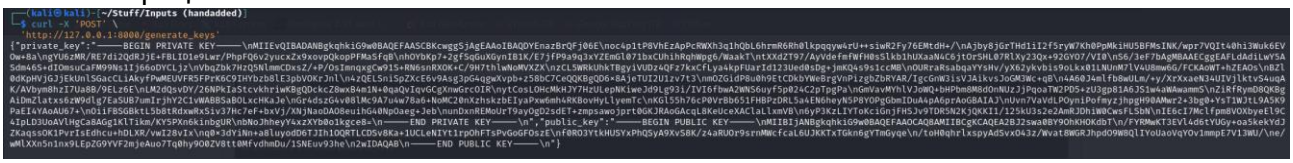
Внесемо зміни у файли:



Спробуємо з валідними підписом та публічним ключем провести перевірку над цим файлом. Отримали наступну відповідь, що свідчить не невідповідність підпису до цього файлу:

Демонстрація роботи веб-сервісу через cmd (curl)

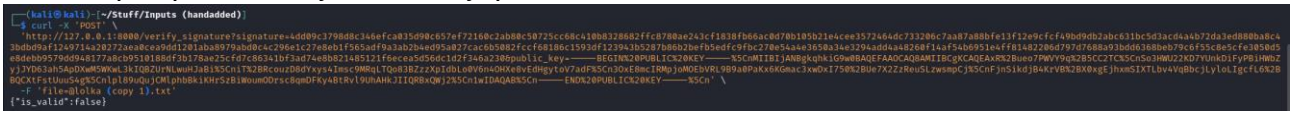
Генерація ключів:



Підписання документу:

Перевірка підпису:

Перевірка підпису на іншому файлі з тим самим підписом:



Висновки

Реалізація веб-сервісу для роботи з електронним цифровим підписом вимагає розуміння криптографічних принципів, таких як асиметричне шифрування та алгоритми хешування. Важливими теоретичними аспектами є використання приватного та публічного ключів, формату PEM, схеми доповнення PSS та алгоритму хешування SHA-256.

У програмній реалізації ми розглянули ключові функції для генерації RSA ключів, підписання та перевірки підписів, а також створення веб-сервісу за допомогою FastAPI. Використання FastAPI забезпечує швидку та ефективну реалізацію сервісу для генерації ключів, підписання документів та перевірки підписів. Завдяки Uvicorn, ми можемо легко запустити сервіс і зробити його доступним для користувачів.