**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»**

**КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ**

МЕТОДИЧНІ ВКАЗІВКИ І ЗАВДАННЯ

ДО ЛАБОРАТОРНИХ ЗАНЯТЬ З ДИСЦИПЛІНИ

**«ТЕХНОЛОГІЇ СТВОРЕННЯ КРИПТОГРАФІЧНИХ ДОДАТКІВ»**

для студентів ОС «Бакалавр» галузі знань F Інформаційні технології

спеціальності F5 Кібербезпека та захист інформації усіх форм навчання



Дрогобич 2025

**УДК 004.456.5(72)**

**М 54**

Методичні вказівки і завдання до лабораторних занять з дисципліни «Технології створення криптографічних додатків» для студентів ОС «Бакалавр» галузі знань F Інформаційні технології спеціальності F5 Кібербезпека та захист інформації усіх форм навчання / уклад. М.О. Александров, Т.В. Скрипник. – Дрогобич : ДонНТУ, 2025 - 106 с.

Наведені завдання і приклади виконання лабораторних занять з дисципліни «Технології створення криптографічних додатків» присвячені знайомству та поетапному опануванню студентами ключових інструментів та методів криптографії. В рамках лабораторних занять розглядається базове використання бібліотек Python, що містять потужний інструментарій для цього. Надані методичні вказівки включають теоретичний та практичний матеріал у вигляді фрагментів програмного коду щодо виконання лабораторних занять. Завдання орієнтовані на самостійну підготовку до лабораторних занять студентів всіх форм навчання спеціальності F5 «Кібербезпека та захист інформації», а також студентів всіх спеціальностей галузі знань F Інформаційні технології.

Укладачі: Микита АЛЕКСАНДРОВ, к.т.н., доц. каф. ПМІ

Тетяна СКРИПНИК, ст. викладач каф. ПМІ

Рецензент: Сергій КОВАЛЬОВ, к.т.н., доц., зав. каф. електронної техніки і комп’ютерної інженерії

Відповідальний за випуск: Ярослав ДОРОГИЙ, д.т.н., проф., в.о. зав. кафедри Прикладної математики та інформатики

Затверджено на засіданні навчально-методичного відділу ДонНТУ,

протокол № 7 від 27.05.2025 р.

Ухвалено на засіданні кафедри прикладної математики і інформатики,

протокол. № 5 від 30.04.2025 р.

© Донецький національний

технічний університет, 2025

**ЗМІСТ**

[ВСТУП 4](#_heading=h.ik4hieo5xz3g)

[Критерії оцінювання робіт 5](#_heading=h.slpkngiqci4x)

[1. ЛАБОРАТОРНЕ ЗАНЯТТЯ №1 Основи роботи з Python для криптографії 6](#_heading=h.pqhlccbezjw)

[Варіанти завдань до лабораторного заняття №1 7](#_heading=h.yztivyzfpbq9)

[Теоретичні відомості до виконання лабораторного заняття № 1. 7](#_heading=h.xx2lk3y32t43)

[2. ЛАБОРАТОРНЕ ЗАНЯТТЯ №2 Реалізація симетричного шифрування 14](#_heading=h.ejp5qob5nmvg)

[Варіанти завдань до лабораторного заняття №2 14](#_heading=h.q85nsbdf94av)

[Теоретичні відомості до виконання лабораторного заняття № 2. 15](#_heading=h.nzn4iemxbh4o)

[3. ЛАБОРАТОРНЕ ЗАНЯТТЯ №3. Реалізація RSA та цифрових підписів 24](#_heading=h.wrwfi7l0bvfw)

[Варіанти завдань до лабораторного заняття №3. 25](#_heading=h.jb7nlugwmpm2)

[Теоретичні відомості до виконання лабораторного заняття № 3. 25](#_heading=h.21g23i3y9vsb)

[4. ЛАБОРАТОРНЕ ЗАНЯТТЯ №4. Хешування та автентифікація паролів 31](#_heading=h.jmdq89dk3dso)

[Теоретичні відомості до виконання лабораторного заняття № 4. 32](#_heading=h.3k333gtppydr)

[5. ЛАБОРАТОРНЕ ЗАНЯТТЯ №5. Реалізація шифрування файлів 37](#_heading=h.i0fttiu2qs8g)

[Теоретичні відомості до виконання лабораторного заняття № 5. 39](#_heading=h.pbsiv8igbz1y)

[6. ЛАБОРАТОРНЕ ЗАНЯТТЯ №6. Реалізація шифрування файлів 46](#_heading=h.cfm7v2lj5sgo)

[Теоретичні відомості до виконання лабораторного заняття № 6. 48](#_heading=h.i3sasz1chsku)

[7. ЛАБОРАТОРНЕ ЗАНЯТТЯ №7. Реалізація цифрового сертифіката та PKI 62](#_heading=h.lon9a2la55b2)

[Теоретичні відомості до виконання лабораторного заняття № 7. 64](#_heading=h.r6oitpx4q0dm)

[8. ЛАБОРАТОРНЕ ЗАНЯТТЯ №8. Написання простого блокчейну 88](#_heading=h.qgkslgy23j86)

[Теоретичні відомості до виконання лабораторного заняття № 8. 89](#_heading=h.74wj7eeewjh9)

[СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 105](#_heading=h.8o1v8kb1lsc3)

[ІНФОРМАЦІЙНІ РЕСУРСИ 105](#_heading=h.8g5c5f71cvwx)

# ВСТУП

Дисципліна «Технології створення криптографічних додатків» входить до списку категорії дисциплін, що є обов’язковими згідно освітньої програми для означеної групи студентів. Основною метою даного курсу є формування практичних навичок розробки криптографічних додатків та розуміння принципів, на яких базується сучасна криптографія.

Курс передбачає поетапне опанування студентами ключових інструментів та методів криптографії. Основні технології створення криптографічних додатків поділяються за такими категоріями, як:

* типи криптографічних алгоритмів (симетричне та асиметричне шифрування, хешування та цифрові підписи)
* програмні рішення
* функціональне призначення (захист даних, автентифікація, цілісність даних, розподіл ключів)
* рівень інтеграції (базові криптографічні додатки та комплексні системи)
* тип середовища (локальні та мережеві додатки).

В рамках лабораторних занять розглядається базове використання бібліотек Python, принципи симетричного та асиметричного шифрування, механізми створення цифрових підписів та хешування. Завершальним етапом є реалізація студентами простого блокчейну — інтегрованого криптографічного рішення, що демонструє практичне застосування отриманих знань.

Обрання мови Python для опанування даної дисципліни є доцільним завдяки її простоті та читабельності синтаксису, що робить її ідеальною навіть для новачків. Крім того, Python має широкий набір бібліотек, таких як *cryptography*, *PyCrypto* та *hashlib*, які значно спрощують реалізацію складних криптографічних алгоритмів. Гнучкість мови дозволяє легко інтегрувати криптографічні рішення в різні додатки, що демонструє практичний аспект роботи з технологіями безпеки. Розробка криптографічних додатків за допомогою мови програмування Python завдяки своїм потужним бібліотекам є особливо зручним способом для реалізації криптографічних алгоритмів. Крім Python, популярними є також Java (з використанням BouncyCastle) та C++ (з OpenSSL), які в даних методичних вказівках не обрано, але їх можна опанувати самостійно при необхідності.

Методичні вказівки спрямовані на забезпечення структурованого та доступного навчального процесу. Вони містять чіткі інструкції, приклади кодів та пояснення, що полегшують опанування складних тем і сприяють формуванню практичних навичок програмування і проєктування криптографічних систем.

# Критерії оцінювання робіт

Оцінювання знань студентів з дисциплін «Технології створення криптографічних додатків» формуються наступним чином:

| **Вид роботи** | **Робота в семестрі (кільк. балів)** | | | | | | | | **Сума балів** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лабораторні роботи/ мінімум | **№1** | **№2** | **№3** | **№4** | **№5** | **№6** | **№7** | **№8** | **40** |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | **24** |
| **Підсумок за роботи протягом семестру** | | | | | | | | | **40/24** |
| **Підсумок за екзаменаційну роботу** | | | | | | | | | **60** |
| **Всього за семестр** | | | | | | | | | **100** |

***Критерії оцінювання виконання лабораторного заняття:***

| **Кількість балів** | **Критерій** |
| --- | --- |
| 5 | Завдання виконано повністю, без помилок; результати виконання представлено своєчасно (наприкінці заняття або на початку наступного). |
| 4 | Завдання виконано повністю, з незначними помилками; результати виконання представлено своєчасно (наприкінці заняття або на початку наступного). |
| 3 | Завдання виконано не менше ніж на 60%, присутні незначні помилки; результати виконання представлено з запізненням, але не пізніше ніж за 2 тижні до закінчення семестру. |

# 

# 1. ЛАБОРАТОРНЕ ЗАНЯТТЯ №1 Основи роботи з Python для криптографії

**Мета -** опанування основних інструментів Python для криптографічних задач. Отримання навичок генерації випадкових чисел, роботи з двійковими даними, шифрування та розшифрування повідомлень та перетворення зашифрованих байтів в початковий текст.

**Завдання.** Необхідно реалізувати програму, яка виконує наступні операції:

* Генерація випадкового числа або набору байтів для криптографічних цілей.
* Перетворення текстового повідомлення в байти та його обробка.
* Створення криптографічно стійкого ключа для шифрування та збереження його у файл.
* Зашифрування повідомлення за допомогою згенерованого ключа.
* Розшифрування зашифрованого повідомлення.
* Переведення зашифрованих байтів назад у звичайний текст.

**Вимоги до завдання:**

* Для генерації випадкових чисел використовувати бібліотеки *secrets* та *os*.*urandom*.
* Повідомлення для обробки повинні бути різними в залежності від варіанту.
* Ключ для шифрування генерується випадковим чином і зберігається у файл.
* Виконати шифрування та розшифрування повідомлення за допомогою бібліотеки *cryptography*.
* Перетворити байти, отримані після шифрування, назад у текст.

**Вимоги до звіту:**

1. Мета, формулювання завдання та варіант.
2. Опис процесу розробки додатку (кожен етап має бути описаний).
3. Опис процесу тестування роботи розробленого додатку (кожен скріншот повинен містити опис та підпис).
4. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

## Варіанти завдань до лабораторного заняття №1

| **№** | **Текст повідомлення** | **Випадкове число (Сіль)** |
| --- | --- | --- |
| **1** | "Криптографія на Python: перші кроки." | від 0 до 100 |
| **2** | "Python як інструмент для криптографії." | 32 байт |
| **3** | "Шифрування та захист даних." | від 0 до 1000 |
| **4** | "Використання секретного ключа для шифрування." | 16 байтів |
| **5** | "Основи роботи з криптографією в Python." | від 0 до 500 |
| **6** | "Ключова криптографія на Python." | 64 байт |
| **7** | "Обробка даних і генерація ключів." | від 0 до 255 |
| **8** | "Захист даних та створення ключів у Python." | від 0 до 9999 |
| **9** | "Генерація випадкових чисел для криптографії." | 128 байт |
| **10** | "Створення шифруючого ключа з Python." | від 0 до 5000 |

## Теоретичні відомості до виконання лабораторного заняття № 1.

1. **Генерація випадкових байтів та чисел.**

Застосовується бібліотека *secrets* для генерації криптографічно стійких випадкових чисел, що підходять для використання в криптографії. Бібліотека *secrets* є більш безпечною для створення випадкових чисел, ніж звичайні методи, такі як *random*, оскільки вона забезпечує вищий рівень випадковості.

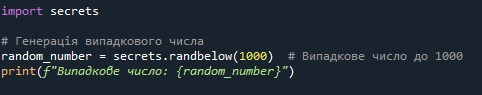


Рисунок 1.1. Приклад коду для генерації випадкового числа

**secrets.randbelow(1000)** генерує випадкове ціле число в межах від 0 до 999. Це число буде криптографічно стійким, тому його можна використовувати для створення випадкових ключів або інших криптографічних операцій. Метод *randbelow* є безпечнішим, ніж традиційний *random*.*randint*, і тому рекомендований для застосуванні в криптографії.

1. **Генерація випадкових байтів.**

Для генерації випадкових байтів використовується бібліотека *os*, а саме функція *os.urandom*, що генерує випадкові байти для криптографічних цілей. Випадкові байти можна обирати для створення ключів шифрування, ініціалізаційних векторів (англ. initialization vector, IV) або для будь-яких інших криптографічних задач.

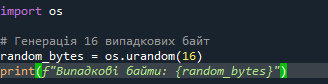


Рисунок 1.2. Приклад коду для генерації випадкових байт

**os.urandom(16)** генерує 16 випадкових байтів (128 біт). Байти отримуються з криптографічно стійкого джерела, що робить їх придатними для використання в різних криптографічних операціях, таких як генерація секретних ключів або ініціалізаційних векторів для шифрування. Метод *urandom* є більш надійним для криптографічних застосувань, ніж методи, що засновані на псевдовипадкових числах, оскільки *os.urandom* отримує випадковість з операційної системи, яка використовує надійніші джерела рандомізації.

1. **Обробка тексту.**

Текстові повідомлення часто потрібно перетворювати у байти для подальшого шифрування або іншої обробки. У цьому етапі потрібно перетворити текст у байти за допомогою методу *encode*().



Рисунок 1.3. Приклад коду для обробки тексту

**message.encode('utf-8')** перетворює рядок тексту в байти з використанням кодування UTF-8. Оскільки UTF-8 є стандартом для кодування символів, що підтримує всі мовні символи та є універсальним для обробки тексту в багатьох мовах програмування.

Після кодування текст перетворюється в байтову послідовність, яку можна використовувати для подальшого шифрування чи збереження.

1. **Створення секретного ключа для шифрування.**

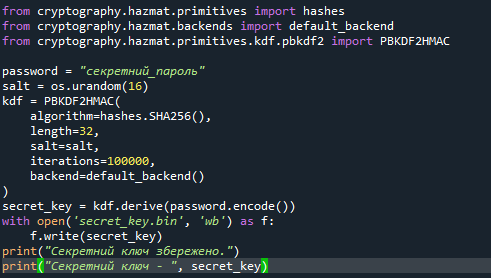


Рисунок 1.4. Приклад коду для створення секретного ключа

Пароль буде використовуватись як вхідні дані для генерації ключа.

**Сіль** (*salt*) — це випадкові дані, які додаються до пароля для забезпечення додаткової криптографічної стійкості.

**Пароль.** Тут використаний фіксований пароль "секретний\_пароль".

**УВАГА!!** В лабораторній роботі слід використовувати ПІБ студента як пароль.

**Сіль.** Тут використано випадкову сіль розміром 16 байтів за допомогою os.urandom(16).

**УВАГА!!** В лабораторній роботі слід генерувати випадкове число за варіантом завдання.

**KDF** (Key Derivation Function). На наступному кроці ініціалізується PBKDF2HMAC. Це функція похідних ключів, яка використовує пароль і сіль для генерації ключа.

Алгоритм хешування - SHA-256. Він є стійким і надає достатньо високий рівень безпеки для більшості випадків.

Довжина ключа - 32 байти, що є стандартним для AES (симетричне шифрування).

Кількість ітерацій - 100000 ітерацій для підвищення стійкості до атак методом перебору.

Після налаштування PBKDF2HMAC, генерується секретний ключ, який буде використовуватися для шифрування та розшифрування повідомлення. Ключ генерується через метод *derive*(), де передається пароль, з якого буде створено криптографічно стійкий ключ.

1. **Шифрування повідомлення.**

Для шифрування потрібно застосувати бібліотеку *cryptography* та симетричне шифрування.

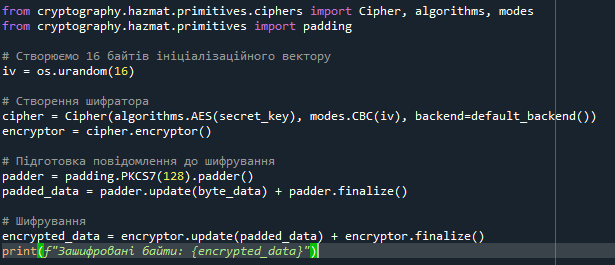


Рисунок 1.5. Приклад коду для шифрування повідомлення

**Ініціалізаційний вектор (IV)** є важливою складовою для деяких режимів шифрування, таких як CBC. Він додає додаткову випадковість до процесу шифрування. В данному випадку IV генерується випадково за допомогою функції *os*.*urandom(16)*, щоб отримати 16 байтів.

**Створення шифратора.** Об'єкт *Cipher* ініціалізується з використанням алгоритму AES, вибраного секретного ключа (який був згенерований раніше) та ініціалізаційного вектору (IV). Вибір режиму CBC є поширеним для шифрування блоків даних, оскільки він покращує стійкість до деяких атак, шифруючи блоки даних залежно від попереднього блоку.

**Паддінг (Padding).** Блоки даних, які передаються для шифрування, повинні бути кратними 16 байтам (розмір блоку для AES). Якщо розмір повідомлення не кратний 16 байтам, то додається "паддінг" - додаткові байти, які доповнюють повідомлення до потрібної довжини. Для цього використовуємо стандартний алгоритм паддінгу PKCS7. Після паддінгу дані стають готовими для шифрування, і надалі передаються у шифратор.

**Шифрування.** Шифрування здійснюється через метод *encryptor.update(padded\_data).* Після цього дані, що пройшли через шифратор, стають зашифрованими. Оскільки шифрування може бути виконане частинами, після обробки всіх частин настає виклик *finalize*(), щоб завершити процес. Зашифровані байти (які містять саму зашифровану інформацію) виводяться на екран. Ці байти будуть виглядати як випадкові дані, які не можна зрозуміти без ключа та IV.

1. **Розшифрування повідомлення**.

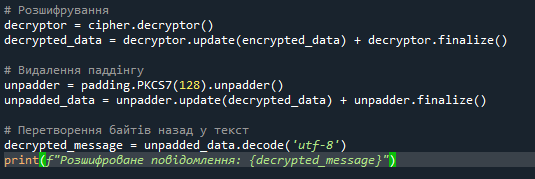


Рисунок 1.6. Приклад коду для розшифрування повідомлення

**Дешифрування зашифрованих даних.** Спочатку слід створити дешифратор, використовуючи той же самий ключ та ініціалізаційний вектор, що були використані при шифруванні. Для дешифрування використовується метод *update*(), який поступово дешифрує байти, а також метод *finalize*(), щоб завершити дешифрування.

**Видалення паддінгу.** Оскільки під час шифрування було додано паддінг для вирівнювання даних до необхідної довжини (в наведеному випадку 128 біт), розшифровані дані також містять цей паддінг. Для видалення паддінгу використовується клас PKCS7, а метод *unpadder*.*update()* працює на зняття цього паддінгу, в той час як *finalize*() завершує цей процес.

**Перетворення байтів у текст.** Після того як відбулося видалення зайвих байтів (паддінг), отримані дані можна перетворити назад у звичайний текст за допомогою методу *decode*(). Цей етап забезпечує повернення зашифрованого тексту у його початкову форму.

**Лістинг**

import secrets

import os

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.primitives import padding

# Генерація випадкового числа

random\_number = secrets.randbelow(1000) # Випадкове число до 1000

print(f"Випадкове число: {random\_number}")

# Генерація 16 випадкових байт

random\_bytes = os.urandom(16)

print(f"Випадкові байти: {random\_bytes}")

message = "Криптографія на Python"

byte\_data = message.encode('utf-8')

print(f"Текст у байтах: {byte\_data}")

password = "секретний\_пароль"

salt = random\_bytes

kdf = PBKDF2HMAC(

algorithm=hashes.SHA256(),

length=32,

salt=salt,

iterations=100000,

backend=default\_backend()

)

secret\_key = kdf.derive(password.encode())

with open('secret\_key.bin', 'wb') as f:

f.write(secret\_key)

print("Секретний ключ збережено.")

print("Секретний ключ - ", secret\_key)

# Створюємо 16 байтів ініціалізаційного вектору

iv = os.urandom(16)

# Створення шифратора

cipher = Cipher(algorithms.AES(secret\_key), modes.CBC(iv), backend=default\_backend())

encryptor = cipher.encryptor()

# Підготовка повідомлення до шифрування

padder = padding.PKCS7(128).padder()

padded\_data = padder.update(byte\_data) + padder.finalize()

# Шифрування

encrypted\_data = encryptor.update(padded\_data) + encryptor.finalize()

print(f"Зашифровані байти: {encrypted\_data}")

# Розшифрування

decryptor = cipher.decryptor()

decrypted\_data = decryptor.update(encrypted\_data) + decryptor.finalize()

# Видалення паддінгу

unpadder = padding.PKCS7(128).unpadder()

unpadded\_data = unpadder.update(decrypted\_data) + unpadder.finalize()

# Перетворення байтів назад у текст

decrypted\_message = unpadded\_data.decode('utf-8')

print(f"Розшифроване повідомлення: {decrypted\_message}")

**Питання для самоконтролю**

1. Що таке криптографічно стійке випадкове число і чим воно відрізняється від псевдовипадкового числа?
2. Яка функція в бібліотеці Python використовується для генерації криптографічно стійких випадкових чисел?
3. Як використати функцію secrets.randbelow() для генерації випадкового числа в діапазоні від 0 до 1000?
4. Чому для криптографічних задач не рекомендується використовувати звичайну бібліотеку random?
5. Що таке функція os.urandom() і для чого вона використовується в криптографії?
6. Як згенерувати 32 байти випадкових даних з використанням os.urandom()?
7. Яка різниця між генерацією випадкових байтів за допомогою secrets і os.urandom?
8. Як у Python перетворити рядок тексту в байти для подальшого шифрування?
9. Що таке кодування UTF-8 і чому його часто використовують при роботі з текстовими даними в Python?
10. Як можна використати згенеровані випадкові байти для створення криптографічного ключа?
11. Чому важливо використовувати криптографічно стійкі випадкові числа під час генерації ключів для шифрування?
12. Що відбудеться, якщо для генерації ключа шифрування використовувати не криптографічно стійке джерело випадковості?
13. Як в Python можна перетворити текст у байти для зберігання у файлі або базі даних?

**Література: [1-8]**

# 2. ЛАБОРАТОРНЕ ЗАНЯТТЯ №2 Реалізація симетричного шифрування

**Мета -** ознайомлення з принципами роботи симетричних алгоритмів шифрування AES та ChaCha20. Реалізація шифрування та розшифрування текстових повідомлень за допомогою AES та ChaCha20 у Python. Закріплення навичок роботи з бібліотеками *pycryptodome* та *cryptography*.

**Завдання.** Необхідно реалізувати симетричний алгоритм згідно з варіантом завдання, використовуючи відповідний режим роботи, секретний ключ та ініціалізаційний вектор (IV) за необхідності.

**Вимоги до завдання:**

* Приведений короткий опис заданого за варіантом симетричного алгоритму шифрування з відповідним режимом роботи (принципи роботи, особливості використання, розповсюдженість, відмінності, тощо).
* Використання бібліотек бібліотеками pycryptodome та/або cryptography.
* Програма повинна містити функціонал для шифрування та розшифрування текстових файлів, зі змогою збереження результатів в окремому файлі (початковий файл, зашифрований, розшифрований).
* Ключ для шифрування генерується випадковим чином, але має довжину відповідно до варіанту.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом симетричного алгоритму шифрування з відповідним режимом роботи.
3. Опис процесу розробки додатку (кожен етап повинен бути описаний).
4. Опис процесу тестування роботи розробленого додатку (кожен скріншот повинен містити опис та підпис).
5. Опис вмісту вхідних/результуючих файлів.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

## Варіанти завдань до лабораторного заняття №2

| **№** | **Завдання** |
| --- | --- |
| **1** | Реалізувати алгоритм AES у режимі CBC з довжиною ключа 128 біт для шифрування та розшифрування текстового файлу. |
| **2** | Реалізувати алгоритм AES у режимі CFB з довжиною ключа 256 біт для шифрування та розшифрування текстового файлу. |
| **3** | Реалізувати алгоритм ChaCha20 з довжиною ключа 256 біт для шифрування та розшифрування текстового файлу. |
| **4** | Реалізувати алгоритм AES у режимі GCM з довжиною ключа 192 біт для шифрування та розшифрування текстового файлу. |
| **5** | Реалізувати алгоритм ChaCha20 з довжиною ключа 128 біт для шифрування та розшифрування текстового файлу. |
| **6** | Реалізувати алгоритм AES у режимі CTR з довжиною ключа 128 біт для шифрування та розшифрування текстового файлу. |
| **7** | Реалізувати алгоритм ChaCha20 з довжиною ключа 192 біт для шифрування та розшифрування текстового файлу. |
| **8** | Реалізувати алгоритм AES у режимі OFB з довжиною ключа 256 біт для шифрування та розшифрування текстового файлу. |
| **9** | Реалізувати алгоритм AES у режимі CBC з довжиною ключа 256 біт для шифрування та розшифрування текстового файлу. |
| **10** | Реалізувати алгоритм AES у режимі OFB з довжиною ключа 128 біт для шифрування та розшифрування текстового файлу. |

## Теоретичні відомості до виконання лабораторного заняття № 2.

1. **Симетричне шифрування -** це метод криптографічного перетворення даних, у якому один і той самий ключ використовується як для шифрування, так і для розшифрування повідомлення. Основними перевагами цього методу є висока швидкість обробки даних та простота реалізації, проте він вимагає безпечної передачі ключа між відправником і отримувачем.

У цій практичній роботі розглядаються два сучасні симетричні алгоритми:

* AES (Advanced Encryption Standard) – один із найпоширеніших алгоритмів блочного шифрування.
* ChaCha20 – потоковий алгоритм шифрування, що використовується для швидкої та безпечної передачі даних.

Метою роботи є практична реалізація цих алгоритмів на мові Python з використанням відповідних криптографічних бібліотек.

1. **Блочне та потокове шифрування.** Симетричне шифрування поділяється на два основних види:

* ***Блочне шифрування.*** Дані розбиваються на блоки фіксованого розміру (наприклад, 128 біт для AES). Для кожного блоку застосовується однакова операція шифрування. Приклади алгоритмів - AES, DES, 3DES.
* ***Потокове шифрування.*** Дані шифруються по одному біту або байту в потоці. Зазвичай використовується генератор псевдовипадкових чисел для створення шифропотоку. Приклади алгоритмів - ChaCha20, RC4, Salsa20.

1. **Огляд алгоритму AES.** AES (Advanced Encryption Standard) – це стандарт блочного шифрування, який прийшов на зміну застарілому DES. Він був розроблений Вінсентом Райменом та Йоаном Даменом і став офіційним стандартом у 2001 році.

Характеристики AES:

* Довжина блоку 128 біт
* Довжина ключа 128, 192 або 256 біт
* Висока стійкість до криптоаналітичних атак
* Використовується у протоколах TLS, WPA2, IPsec

***Основні режими роботи AES:***

| Режим | Тип шифрування | Опис |
| --- | --- | --- |
| *ECB (Electronic Codebook)* | Блочне | Кожен блок шифрується окремо (вразливий до атак) |
| *CBC (Cipher Block Chaining)* | Блочне | Використовує ініціалізаційний вектор (IV), кожен блок залежить від попереднього |
| *CFB (Cipher Feedback Mode)* | Потокове | Шифрує менші порції даних, підходить для потокового шифрування |
| *OFB (Output Feedback Mode)* | Потокове | Подібний до CFB, але не залежить від шифрованого тексту |
| *CTR (Counter Mode)* | Потокове | Використовує лічильник для генерації псевдовипадкової послідовності |
| *GCM (Galois/Counter Mode)* | Потокове | Додатково забезпечує цілісність і автентифікацію даних |

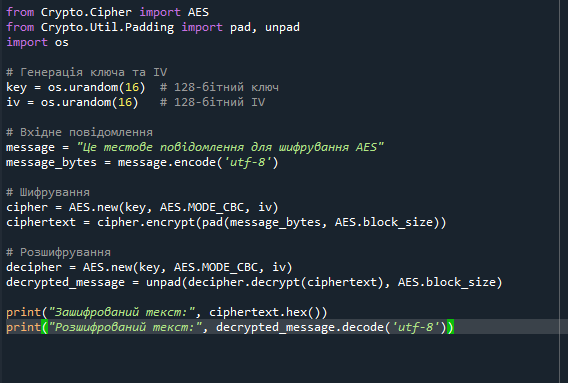


Рисунок 2.1. Приклад реалізації AES у режимі CBC:

1. **Огляд алгоритму ChaCha20.** ChaCha20 – це потоковий алгоритм шифрування, розроблений Даніелем Бернштайном у 2008 році як покращена версія Salsa20. Його основні переваги:

* Висока продуктивність навіть на слабких пристроях.
* Менша чутливість до атак, ніж AES.
* Відсутність необхідності в складних механізмах режимів роботи.

***Структура ChaCha20.*** ChaCha20 працює на 512-бітному внутрішньому стані, що складається з:

* 256-бітного ключа.
* 96-бітного nonce.
* Лічильника, що змінюється під час шифрування.

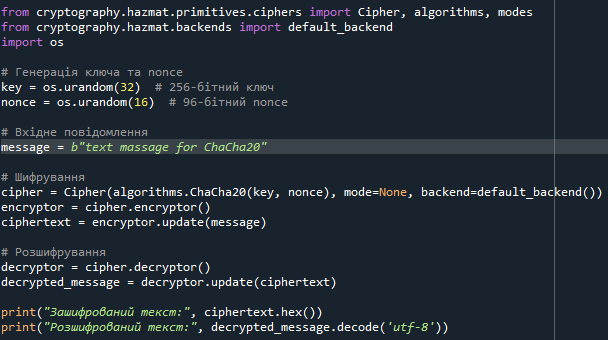


Рисунок 2.2. Приклад реалізації ChaCha20.

1. **Порівняння AES і ChaCha20.** Обидва алгоритми широко застосовуються в сучасній криптографії, але мають свої особливості та сфери використання.

| **Параметр** | **AES (GCM, CBC)** | **ChaCha20** |
| --- | --- | --- |
| **Тип алгоритму** | Блочний | Потоковий |
| **Розмір блоку** | 128 біт | Потокове шифрування |
| **Розмір ключа** | 128, 192, 256 біт | 256 біт |
| **Швидкість** | Висока на апаратному рівні (апаратне прискорення AES-NI) | Висока навіть без апаратного прискорення |
| **Безпека** | Захищений при правильному використанні | Вважається більш стійким до деяких атак |
| **Режими роботи** | CBC, CFB, OFB, CTR, GCM | Не потребує окремих режимів |
| **Використання** | TLS, IPsec, WPA3, VPN | TLS 1.3, WireGuard, Google QUIC |

ChaCha20 часто використовується в мобільних додатках та безпечних з'єднаннях (наприклад, у **Google Chrome, Signal, WireGuard**), оскільки він швидше працює на пристроях без апаратного прискорення AES.

1. **Автентифіковане шифрування (ChaCha20-Poly1305, AES-GCM).** Звичайне симетричне шифрування забезпечує конфіденційність, але не гарантує цілісність і автентифікацію даних. Для захисту від підміни використовують автентифіковане шифрування.

AES у режимі GCM (Galois/Counter Mode) додає механізм перевірки цілісності даних за допомогою тегу автентифікації.

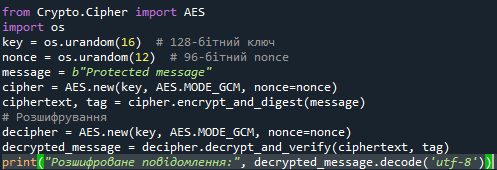
****

Рисунок 2.3. Приклад реалізації AES у режимі GCM

ChaCha20 можна використовувати разом із MAC-алгоритмом Poly1305 для перевірки цілісності повідомлення.

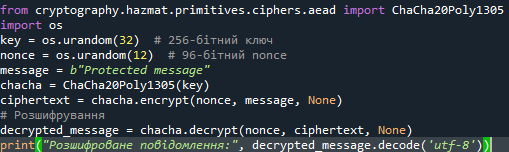


Рисунок 2.4. Приклад реалізації ChaCha20

Обидва підходи (AES-GCM та ChaCha20-Poly1305) застосовуються у TLS 1.3, VPN, месенджерах для безпечного передавання даних.

1. **Приклад реалізації завдання**. Тестове завдання – «Реалізувати AES у режимі GCM з довжиною ключа 128 біт та ChaCha20-Poly1305 з довжиною ключа 256 біт та порівняти отримані шифротексти»

**Алгоритм виконання:**

1. Згенерувати ключі:
   1. AES: 128-бітний ключ + 96-бітний nonce
   2. ChaCha20-Poly1305: 256-бітний ключ + 96-бітний nonce
2. Виконати шифрування обома алгоритмами.
3. Вивести зашифровані дані у вигляді hex-рядка.
4. Виконати розшифрування та перевірити, чи збігаються вихідні дані.

***Як працює програма?***

* Зчитує вміст вхідного файлу (input.txt).
* Виконує шифрування за допомогою AES-GCM та ChaCha20-Poly1305.
* Записує зашифровані дані у aes\_encrypted.txt та chacha\_encrypted.txt.
* Виводить шифротексти та ключі на консоль.
* Розшифровує дані та зберігає їх у файл aes\_decrypted.txt та chacha\_decrypted.txt.
* Виводить розшифровані дані на консоль для перевірки правильності роботи алгоритму.

***Очікуваний результат.***

Після виконання програмного коду (рис. 11) буде отримано наступне:

* Файл aes\_encrypted.txt – містить HEX-представлення зашифрованого тексту AES-GCM.
* Файл chacha\_encrypted.txt – містить HEX-представлення зашифрованого тексту ChaCha20-Poly1305.
* Файл aes\_decrypted.txt – містить вихідний текст, розшифрований AES-GCM.
* Файл chacha\_decrypted.txt – містить вихідний текст, розшифрований ChaCha20-Poly1305.
* У консолі відобразиться зашифрований та розшифрований текст разом із ключами.

**Реалізація на Python.**

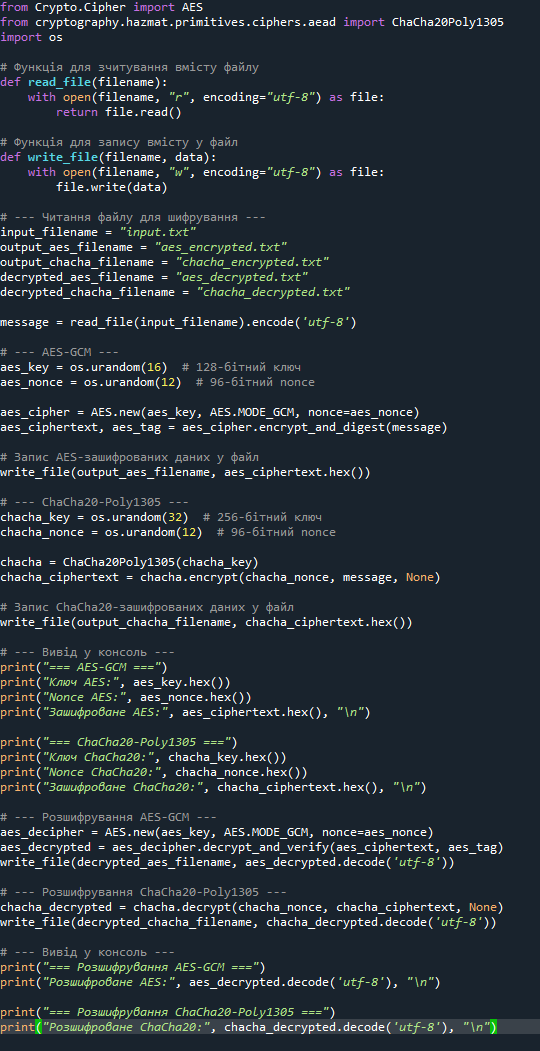
****

Рисунок 2.5. Приклад реалізації програмного коду

**Лістинг**

from Crypto.Cipher import AES

from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305

import os

# Функція для зчитування вмісту файлу

def read\_file(filename):

with open(filename, "r", encoding="utf-8") as file:

return file.read()

# Функція для запису вмісту у файл

def write\_file(filename, data):

with open(filename, "w", encoding="utf-8") as file:

file.write(data)

# --- Читання файлу для шифрування ---

input\_filename = "input.txt"

output\_aes\_filename = "aes\_encrypted.txt"

output\_chacha\_filename = "chacha\_encrypted.txt"

decrypted\_aes\_filename = "aes\_decrypted.txt"

decrypted\_chacha\_filename = "chacha\_decrypted.txt"

message = read\_file(input\_filename).encode('utf-8')

# --- AES-GCM ---

aes\_key = os.urandom(16) # 128-бітний ключ

aes\_nonce = os.urandom(12) # 96-бітний nonce

aes\_cipher = AES.new(aes\_key, AES.MODE\_GCM, nonce=aes\_nonce)

aes\_ciphertext, aes\_tag = aes\_cipher.encrypt\_and\_digest(message)

# Запис AES-зашифрованих даних у файл

write\_file(output\_aes\_filename, aes\_ciphertext.hex())

# --- ChaCha20-Poly1305 ---

chacha\_key = os.urandom(32) # 256-бітний ключ

chacha\_nonce = os.urandom(12) # 96-бітний nonce

chacha = ChaCha20Poly1305(chacha\_key)

chacha\_ciphertext = chacha.encrypt(chacha\_nonce, message, None)

# Запис ChaCha20-зашифрованих даних у файл

write\_file(output\_chacha\_filename, chacha\_ciphertext.hex())

# --- Вивід у консоль ---

print("=== AES-GCM ===")

print("Ключ AES:", aes\_key.hex())

print("Nonce AES:", aes\_nonce.hex())

print("Зашифроване AES:", aes\_ciphertext.hex(), "\n")

print("=== ChaCha20-Poly1305 ===")

print("Ключ ChaCha20:", chacha\_key.hex())

print("Nonce ChaCha20:", chacha\_nonce.hex())

print("Зашифроване ChaCha20:", chacha\_ciphertext.hex(), "\n")

# --- Розшифрування AES-GCM ---

aes\_decipher = AES.new(aes\_key, AES.MODE\_GCM, nonce=aes\_nonce)

aes\_decrypted = aes\_decipher.decrypt\_and\_verify(aes\_ciphertext, aes\_tag)

write\_file(decrypted\_aes\_filename, aes\_decrypted.decode('utf-8'))

# --- Розшифрування ChaCha20-Poly1305 ---

chacha\_decrypted = chacha.decrypt(chacha\_nonce, chacha\_ciphertext, None)

write\_file(decrypted\_chacha\_filename, chacha\_decrypted.decode('utf-8'))

# --- Вивід у консоль ---

print("=== Розшифрування AES-GCM ===")

print("Розшифроване AES:", aes\_decrypted.decode('utf-8'), "\n")

print("=== Розшифрування ChaCha20-Poly1305 ===")

print("Розшифроване ChaCha20:", chacha\_decrypted.decode('utf-8'), "\n")

**Питання для самоконтролю**

1. Що таке симетричне шифрування? У чому його основна відмінність від асиметричного шифрування?
2. Які основні переваги та недоліки симетричного шифрування?
3. Що таке блочне та потокове шифрування? У чому їхні відмінності?
4. Які алгоритми симетричного шифрування є найбільш поширеними?
5. Що таке AES? Які довжини ключа він підтримує?
6. Який розмір блоку використовує AES?
7. Які основні режими роботи AES ви знаєте?
8. У чому особливості режимів CBC, CFB, OFB, CTR, GCM?
9. Які переваги має режим GCM у порівнянні з іншими режимами AES?
10. Чому режим ECB вважається небезпечним?
11. Яку роль відіграє ініціалізаційний вектор (IV) у режимах AES, які його потребують?
12. Що таке PBKDF2, і як його можна використовувати для генерації ключа AES?
13. Що таке ChaCha20? Чим він відрізняється від AES?
14. Чому ChaCha20 є швидшим за AES на деяких пристроях?
15. Що таке ChaCha20-Poly1305 і чому він використовується?
16. Які криптографічні протоколи використовують ChaCha20?

**Література: [1-8]**

**3. ЛАБОРАТОРНЕ ЗАНЯТТЯ №3.   
Реалізація RSA та цифрових підписів Мета -** ознайомлення з алгоритмом RSA та його використанням для шифрування і цифрового підпису. Реалізація генерації ключової пари RSA, створення цифрового підпису повідомлення та перевірки цифрового підпису у Python. Закріплення навичок роботи з криптографічними бібліотеками.

**Завдання.** Необхідно реалізувати криптографічний алгоритм згідно з варіантом завдання.

**Вимоги до завдання:**

* Реалізувати генерацію відкритого та закритого ключів RSA заданої довжини (1024, 2048 або 4096 біт).
* Використовуючи згенеровані ключі, реалізувати функції:
  1. Шифрування та розшифрування повідомлень RSA
  2. Створення цифрового підпису за допомогою заданого алгоритму хешування.
  3. Перевірка цифрового підпису
* Використовувати бібліотеки Python, такі як *cryptography*, *pycryptodome* або *rsa*.
* Використовувати файли для збереження/завантаження вхідних шифрованих/підписаних повідомлень та ключів шифрування.
* Виконати тестування реалізованих функцій на кількох повідомленнях.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом алгоритму шифрування.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. (Кожен скріншот повинен містити опис та підпис.)
5. Опис вмісту вхідних/результуючих файлів.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

## Варіанти завдань до лабораторного заняття №3.

| № | Довжина ключа (біт) | Алгоритм хешування |
| --- | --- | --- |
| 1 | 1024 | SHA-256 |
| 2 | 2048 | SHA-256 |
| 3 | 1024 | SHA-512 |
| 4 | 2048 | SHA-512 |
| 5 | 4096 | SHA-256 |
| 6 | 4096 | SHA-512 |
| 7 | 2048 | SHA3-256 |
| 8 | 1024 | SHA3-512 |
| 9 | 4096 | SHA3-256 |
| 10 | 2048 | SHA3-512 |

## Теоретичні відомості до виконання лабораторного заняття № 3.

1. **RSA (Rivest–Shamir–Adleman) –** це один із найвідоміших алгоритмів асиметричної криптографії, який використовується для:

* Шифрування та розшифрування даних.
* Створення цифрового підпису та його перевірки.

**Принцип роботи RSA.**

* 1. Генерація ключів:
* Обираються два великі прості числа і .
* Обчислюється їх добуток (називається модулем).
* Обчислюється функція Ейлера
* Вибирається відкрита експонента , що є взаємно простим числом із φ(n).
* Обчислюється секретна експонента як обернене число до за модулем
  1. Формування ключів:
* Відкритий ключ -
* Закритий ключ -
  1. Шифрування:
* Повідомлення перетворюється у числову форму.
* Обчислюється зашифроване повідомлення
* Передається отримане .
  1. Розшифрування:
* Отримане розшифровується за допомогою закритого ключа

1. **Цифровий підпис RSA**. Цифровий підпис дозволяє гарантувати цілісність та автентичність повідомлення.

Процес підписання повідомлення:

* 1. Обчислюється хеш повідомлення (SHA-256, SHA-512, SHA3-256 тощо).
  2. Хеш підписується закритим ключем :
  3. Отриманий підпис передається разом із повідомленням.

Перевірка підпису:

* 1. Отримане повідомлення хешується, отримуємо
  2. Виконується перевірка підпису за допомогою відкритого ключа

Головна ідея, що лише власник закритого ключа може створити підпис, а будь-хто може його перевірити за допомогою відкритого ключа.

1. **Хеш-функції в цифрових підписах.**

Оскільки RSA працює з числами великої довжини, підписується не саме повідомлення, а його хеш.

Популярні хеш-функції:

* SHA-256 (256 біт)
* SHA-512 (512 біт)
* SHA3-256 (256 біт)

Хеш-функція повинна бути незворотною (неможливо відновити вихідне повідомлення) та стійкою до колізій (різні повідомлення мають різні хеші).

1. **Шифрування та підписання в RSA.** Різниця між шифруванням та підписом:

| Дія | Ключ | Призначення |
| --- | --- | --- |
| Шифрування | Відкритий ключ | Гарантує конфіденційність |
| Розшифрування | Закритий ключ | Отримувач може розшифрувати |
| Підписання | Закритий ключ | Гарантує автентичність |
| Перевірка підпису | Відкритий ключ | Будь-хто може перевірити |

**Часто підпис та шифрування використовуються разом:**

1. Відправник ***створює підпис*** свого повідомлення.
2. Додає підпис до повідомлення та ***шифрує*** його відкритим ключем отримувача.
3. Отримувач ***розшифровує*** повідомлення та перевіряє підпис.
4. **Приклад реалізації завдання на Python.**

***Крок 1. Генерація ключів RSA***



Рисунок 3.1. Приклад генерації ключів.

* Імпортуємо клас RSA з бібліотеки ***Crypto.PublicKey***, який містить методи для роботи з ключами.
* Генеруємо пару ключів RSA довжиною 2048 біт.
* Це стандартний розмір ключа, що забезпечує достатній рівень безпеки.
* Експортуємо закритий ключ та отримуємо відповідний відкритий ключ.
* Закритий ключ використовується для підписання та розшифрування.
* Відкритий ключ використовується для перевірки підпису та шифрування.
* Зберігаємо ключі у файли "***private.pem***" (закритий ключ) та "***public.pem***" (відкритий ключ).
* Використовуємо режим ***"wb"*** (запис у двійковому режимі), оскільки ключі зберігаються у спеціальному PEM-форматі.

***Крок 2. Шифрування повідомлення та його збереження.***

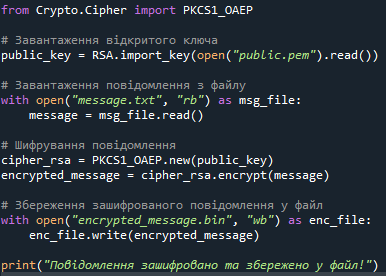


Рисунок 3.2. Приклад шифрування повідомлення та його збереження.

* Завантажуємо відкритий ключ.
* Читаємо вихідне повідомлення з message.txt.
* Шифруємо повідомлення за допомогою RSA та схеми OAEP.
* Записуємо зашифровані дані у файл encrypted\_message.bin.

***Крок 3. Розшифрування повідомлення.***

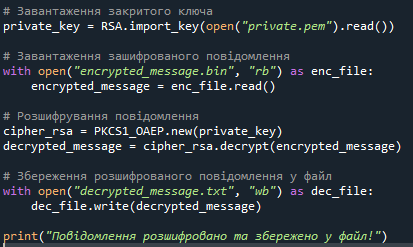


Рисунок 3.3. Приклад розшифрування повідомлення.

* Завантажуємо закритий ключ.
* Читаємо зашифровані дані з файлу ***encrypted\_message.bin.***
* Розшифровуємо повідомлення за допомогою RSA та OAEP.
* Зберігаємо розшифроване повідомлення у файл ***decrypted\_message.txt.***

***Крок 4. Створення цифрового підпису та його збереження.***

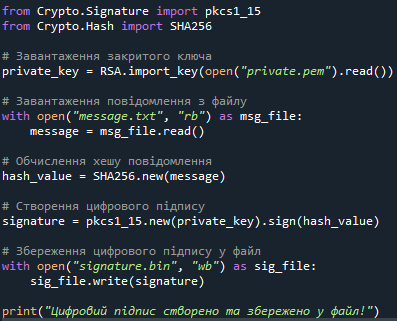


Рисунок 3.4. Приклад створення цифрового підпису та його збереження

* Завантажуємо закритий ключ.
* Читаємо вихідне повідомлення з файлу ***message.txt.***
* Обчислюємо хеш повідомлення ***(SHA-256).***
* Створюємо цифровий підпис.
* Зберігаємо підпис у файл ***signature.bin.***

***Крок 5. Перевірка цифрового підпису.***

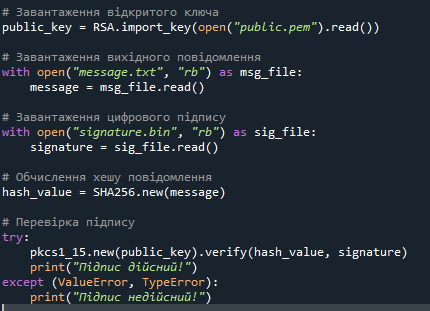
******

Рисунок 3.5. Приклад перевірки цифрового підпису

* Завантажується відкритий ключ.
* Читається вихідне повідомлення та підпис із файлів.
* Обчислюється хеш повідомлення.
* Виконується перевірка підпису.
* Якщо підпис правильний → "*Підпис дійсний!*".
* Якщо підпис змінено або невірний → "*Підпис недійсний!*".

Файли, що створюються в процесі:

* private.pem – закритий ключ
* public.pem – відкритий ключ
* message.txt – вихідне повідомлення
* encrypted\_message.bin – зашифроване повідомлення
* decrypted\_message.txt – розшифроване повідомлення
* signature.bin – цифровий підпис

**Питання для самоконтролю**

1. Що таке RSA і для чого він використовується?
2. У чому полягає різниця між симетричним і асиметричним шифруванням?
3. Які основні етапи генерації ключів у RSA?
4. Як відбувається процес шифрування даних у RSA?
5. Як здійснюється розшифрування повідомлення у RSA?
6. Що таке цифровий підпис, і яку функцію він виконує?
7. Чому для цифрового підпису використовується хеш-функція?
8. Як відбувається перевірка цифрового підпису?
9. Які переваги та недоліки має RSA?
10. Чому важливо використовувати довгі ключі (2048 біт і більше) у RSA?
11. Як можна зберігати та завантажувати відкритий і закритий ключі у файли?
12. Яка роль бібліотеки *pycryptodome* у реалізації RSA в Python?
13. Як правильно зашифрувати та зберегти повідомлення у файл?
14. Як реалізувати цифровий підпис у Python та зберегти його у файл?

**Література: [1-8]**

# 4. ЛАБОРАТОРНЕ ЗАНЯТТЯ №4. Хешування та автентифікація паролів

**Мета -** ознайомлення із сучасними алгоритмами хешування паролів та методами їхнього захисту. Реалізація криптографічних хеш-функцій (SHA-256, SHA-3, BLAKE2) та алгоритмів безпечного хешування паролів (PBKDF2, bcrypt, Argon2) у Python. Проведення аналізу ефективності різних алгоритмів.

**Завдання:**

* Ознайомитися з криптографічними алгоритмами хешування (SHA-256, SHA-3, BLAKE2).
* Вивчити методи захисту паролів за допомогою алгоритмів PBKDF2, bcrypt, Argon2.
* Реалізувати, протестувати та порівняти час виконання алгоритмів.
* Виконати багаторазове хешування (100-1000 разів) для оцінки максимального/середнього/мінімального часу виконання.

**Вимоги до завдання:**

* Написати програму, яка хешує введений користувачем пароль за допомогою двох алгоритмів хешування (SHA-256, SHA-3, BLAKE2) відповідно до індивідуального варіанта. Додати випадкову сіль та повторити хешування. Виконати хешування 100 разів і визначити середній час виконання для кожного алгоритму.
* Захешувати той самий пароль із використанням одного з алгоритмів: PBKDF2, bcrypt або Argon2 (відповідно до варіанта). Здійснити хешування 100-1000 разів і визначити середній час виконання. Реалізувати перевірку введеного пароля на відповідність збереженому хешу.
* Провести аналіз часу виконання для різних алгоритмів. Порівняти безпеку методів, враховуючи стійкість до атак brute-force та rainbow table. Зробити висновки щодо доцільності використання кожного методу.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданих за варіантом алгоритмів хешування.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. (Кожен скріншот повинен містити опис та підпис.)
5. Порівняльна таблиця часових вимірювань алгоритмів хешування .
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

Варіанти завдань до лабораторного заняття 4.

| № | Алгоритми хешування | Алгоритм захисту паролів |
| --- | --- | --- |
| 1 | SHA-256, BLAKE2 | bcrypt |
| 2 | SHA-3, BLAKE2 | Argon2 |
| 3 | SHA-256, BLAKE2 | PBKDF2 |
| 4 | SHA-3, SHA-256 | bcrypt |
| 5 | BLAKE2, SHA-256 | Argon2 |
| 6 | BLAKE2, SHA-3 | PBKDF2 |
| 7 | SHA-256, SHA-3 | bcrypt |
| 8 | SHA-3, BLAKE2 | Argon2 |
| 9 | SHA-256, BLAKE2 | PBKDF2 |
| 10 | SHA-256, SHA-3 | bcrypt |

## Теоретичні відомості до виконання лабораторного заняття № 4.

Хешування є фундаментальним механізмом у криптографії, що широко використовується для збереження паролів. Проте просте хешування паролів має вразливості, такі як атаки методом перебору (brute-force) та атаки за допомогою райдужних таблиць (rainbow table). Для захисту паролів використовують спеціальні алгоритми, що додають випадкову сіль (salt) та виконують хешування багаторазово (key stretching).

**4.1 Алгоритми хешування**

Криптографічні хеш-функції – це односторонні математичні функції, які приймають на вхід повідомлення будь-якої довжини та повертають хеш-фіксованого розміру. Головні вимоги до хеш-функцій:

* Односторонність – неможливість обернення хешу назад у вихідні дані.
* Детермінованість – однакові вхідні дані завжди дають однаковий хеш.
* Стійкість до колізій – складність у знаходженні двох різних вхідних значень із однаковим хешем.
* Ефективність – швидке обчислення хешу.

**SHA-256 (Secure Hash Algorithm 256-bit).** SHA-256 є частиною сімейства SHA-2 і генерує хеш розміром 256 біт (32 байти). Використовується у блокчейнах (наприклад, Bitcoin), цифрових підписах та сертифікатах безпеки. Відносно швидкий, але вразливий до атак brute-force без додаткового захисту.

**SHA-3** - це новіший стандарт хеш-функцій, заснований на алгоритмі Keccak. Має кращу стійкість до деяких атак, ніж SHA-256. Відрізняється внутрішньою структурою та принципом роботи (губчаста функція).

**BLAKE2** швидший за SHA-256 і забезпечує високу криптографічну стійкість. Оптимізований для ефективної роботи на різних платформах. Використовується у криптографічних додатках та цифрових підписах.

**4.2 Алгоритми захисту паролів**

Прості хеш-функції (SHA-256, SHA-3) не забезпечують достатнього захисту паролів, тому використовуються спеціальні алгоритми, що роблять хешування паролів складнішим для атакуючих.

**PBKDF2 (Password-Based Key Derivation Function 2).** PBKDF2 виконує багаторазове (ітеративне) хешування пароля для ускладнення атак методом перебору. Використовує сіль для унікальності хешу. Вимагає значних обчислювальних ресурсів для зламу. Використовується в Wi-Fi (WPA2), BitLocker, Apple iOS Keychain.

**Bcrypt.** bcrypt використовує адаптивний алгоритм хешування, що включає сіль і змінюваний параметр складності (cost factor). Вбудований механізм автоматичного уповільнення виконання при збільшенні потужності апаратного забезпечення. Використовується у системах Linux (passwd), веб-додатках.

**Argon2 -** сучасний алгоритм, переможець конкурсу Password Hashing Competition (2015). Використовує значний обсяг пам’яті для ускладнення атак. Висока стійкість до атак через спеціалізовані пристрої (GPU, FPGA, ASIC).Використовується у менеджерах паролів (1Password, Bitwarden).

**4.3 Часові характеристики алгоритмів**

Важливим критерієм вибору алгоритму є баланс між швидкістю виконання та стійкістю до атак.

| Алгоритм | Середній час хешування (ms) | Захищеність від brute-force |
| --- | --- | --- |
| SHA-256 | ~0.3 | слабка |
| SHA-3 | ~0.5 | слабка |
| BLAKE2 | ~0.2 | слабка |
| PBKDF2 | ~30 | сильна |
| bcrypt | ~200 | сильна |
| Argon2 | ~250 | дуже сильна |

SHA-256, SHA-3 та BLAKE2 підходять для загального хешування, але не для безпечного збереження паролів.

PBKDF2 забезпечує додатковий захист паролів, але потребує значного часу на виконання.

bcrypt добре підходить для автентифікації, але може бути повільним на старих пристроях.

Argon2 є найкращим варіантом для зберігання паролів завдяки ефективному використанню пам’яті та адаптивності до атак.

**4.4 Приклад виконання (SHA-256, SHA-3, bcrypt)**

***Хешування пароля та вимірювання часу виконання:***

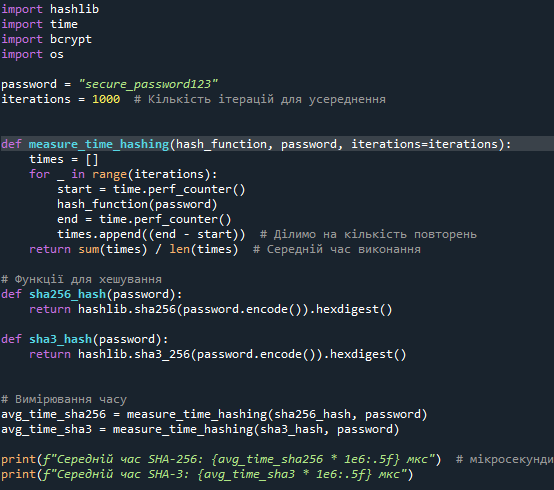
****

Рисунок 4.1. Приклад хешування пароля та вимірювання часу виконання.



Рисунок 4.2. Приклад результатів вимірювання часу виконання.

**Додавання солі та повторне вимірювання:**

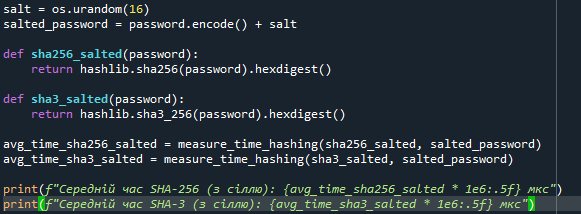
****

Рисунок 4.3. Додавання солі та повторне вимірювання.

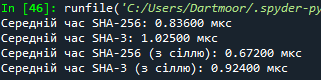


Рисунок 4.4. Приклад результатів вимірювання часу виконання з сіллю.

**Захист пароля за допомогою bcrypt та вимірювання часу.**

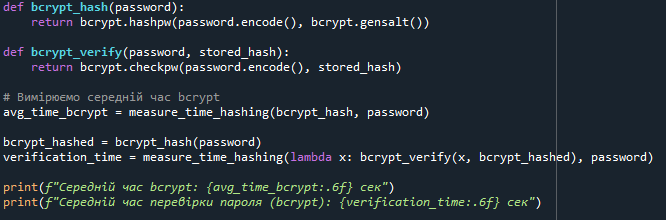
****

Рисунок 4.5. Захист пароля за допомогою bcrypt та повторне вимірювання.

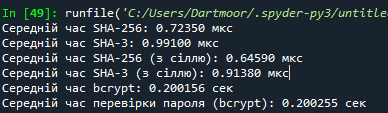


Рисунок 4.6. Приклад результатів вимірювання часу виконання з bcrypt.

**Питання для самоконтролю**

1. Які основні вимоги до криптографічних хеш-функцій?
2. У чому різниця між SHA-256, SHA-3 та BLAKE2?
3. Чому просте хешування паролів є недостатнім для безпечного їх збереження?
4. Що таке сіль (salt) і яку роль вона відіграє у хешуванні паролів?
5. Як атаки методом перебору (brute-force) та райдужні таблиці (rainbow table) загрожують збереженим паролям?
6. Чому хешування паролів багаторазовим виконанням (ітерації) підвищує безпеку?
7. Які переваги та недоліки PBKDF2 у порівнянні з bcrypt та Argon2?
8. Чому bcrypt вважається адаптивним алгоритмом захисту паролів?
9. Чим відрізняється Argon2 від інших алгоритмів хешування паролів?
10. Як можна змінювати параметри bcrypt та Argon2 для підвищення їхньої стійкості до атак?
11. Як вимірюється продуктивність алгоритмів хешування?
12. Чому швидкість хешування важлива для криптографічних алгоритмів?
13. Чи існує оптимальний алгоритм хешування паролів, який підходить для всіх випадків? Чому?
14. Як вибір алгоритму хешування залежить від сценарію його використання?
15. Які практичні рекомендації щодо збереження паролів у сучасних інформаційних системах?

**Література: [1-8]**

# 5. ЛАБОРАТОРНЕ ЗАНЯТТЯ №5. Реалізація шифрування файлів

**Мета -** Ознайомлення з методами симетричного та асиметричного шифрування файлів. Реалізація алгоритмів шифрування та дешифрування файлів, забезпечуючи їхню цілісність. Вивчення механізмів перевірки достовірності даних та цифрового підпису. Опанування бібліотек *cryptography*, *PyCryptodome*, *pyNaCl* та інших інструментів для роботи з криптографією. Отримання навичок створення безпечних рішень для захисту файлів від несанкціонованого доступу та модифікації.

**Завдання:**

Розробити програму, яка реалізує:

* Шифрування файлів із використанням сучасних криптографічних алгоритмів.
* Дешифрування зашифрованих файлів із відновленням вихідних даних.
* Перевірку цілісності даних після дешифрування за допомогою MAC (Message Authentication Code) або цифрового підпису.
* Можливість використання паролів або ключів для шифрування.
* Роботу з текстовими або бінарними файлами.

**Вимоги до завдання:**

**Шифрування -** використовувати один із сучасних алгоритмів: AES (CBC, GCM), ChaCha20-Poly1305, RSA (у гібридній схемі). Уникати слабких режимів (наприклад, AES-ECB). Використовувати випадкові або детерміновані IV (ініціалізаційний вектор) для безпечного шифрування.

**Дешифрування -** реалізувати функцію розшифрування із перевіркою коректності даних. Забезпечити механізм аутентифікації (MAC або цифровий підпис).

**Перевірка цілісності -** використовувати HMAC, AEAD (Authenticated Encryption with Associated Data) або RSA-підпис для перевірки цілісності.

**Формат вихідного файлу -** включати метаінформацію (алгоритм, IV, підпис) у зашифрований файл. Підтримувати можливість відновлення вихідного файлу без втрати даних.

**Додатково -** передбачити збереження ключів у захищеному вигляді або використання пароля для їх генерації.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом алгоритму та режимів роботи.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. Тестування повинно налічувати як позитивні так і негативні сценарії. (Кожен скріншот повинен містити опис та підпис.)
5. Приклади вмісту вихідних, шифрованих та дешифрованих файлів, а також файлів паролів.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

Варіанти завдань до лабораторного заняття №5.

| № | Завдання |
| --- | --- |
| 1 | **AES-256 у режимі CBC + HMAC-SHA256.** Реалізуйте симетричне шифрування файлу з використанням AES-256 у режимі CBC. Використовуйте HMAC-SHA256 для перевірки цілісності файлу. При дешифруванні перевіряйте HMAC перед розшифруванням. |
| 2 | **AES-256 у режимі GCM (автентифіковане шифрування).** Реалізуйте шифрування файлу за допомогою AES-256 у режимі GCM. Додайте IV та автентифікаційний тег до вихідного файлу.  При дешифруванні перевіряйте цілісність за допомогою вбудованого механізму AEAD. |
| 3 | **ChaCha20-Poly1305.** Реалізуйте шифрування файлу за допомогою ChaCha20-Poly1305. Використовуйте 256-бітний ключ та унікальний nonce. Додайте механізм перевірки достовірності даних (Poly1305 MAC). |
| 4 | **RSA-2048 + цифровий підпис (SHA-256).** Реалізуйте асиметричне шифрування файлу з використанням RSA-2048. Підпишіть файл за допомогою приватного ключа (SHA-256 + RSA). Реалізуйте перевірку підпису під час дешифрування. |
| 5 | **Використання пароля для шифрування (PBKDF2 + AES-256-CBC).** Реалізуйте шифрування файлу за допомогою AES-256-CBC. Ключ має генеруватися з пароля за допомогою PBKDF2.  Використовуйте HMAC для перевірки цілісності. |
| 6 | **Симетричне шифрування з XChaCha20.** Реалізуйте XChaCha20 для шифрування файлу. Використовуйте 24-бітний nonce для підвищеної безпеки. Реалізуйте механізм перевірки цілісності даних. |
| 7 | **Гібридне шифрування. AES-256 + ECDH (Elliptic Curve Diffie-Hellman).** Реалізуйте шифрування файлу за допомогою AES-256-GCM. Генеруйте симетричний ключ за допомогою ECDH. Використовуйте аутентифікаційний тег GCM для перевірки цілісності файлу. |
| 8 | **AES-256 у режимі OFB + HMAC-SHA256.** Використовуйте AES-256 у режимі OFB для шифрування файлу. Додайте HMAC-SHA256 для перевірки цілісності. Реалізуйте механізм перевірки HMAC перед дешифруванням. |
| 9 | **Гібридне шифрування. ChaCha20 + RSA-2048.** Використовуйте ChaCha20 для шифрування файлу. Генеруйте випадковий симетричний ключ та шифруйте його RSA-2048. Реалізуйте перевірку цілісності за допомогою Poly1305 MAC. |
| 10 | **AES-256 у режимі CTR + цифровий підпис ECDSA.** Реалізуйте шифрування файлу за допомогою AES-256-CTR. Підпишіть файл за допомогою ECDSA (Elliptic Curve Digital Signature Algorithm). Перевіряйте цифровий підпис перед дешифруванням. |

## Теоретичні відомості до виконання лабораторного заняття № 5.

Шифрування файлів забезпечує їх конфіденційність під час зберігання на локальних носіях, у хмарних сервісах або в базах даних, а також під час передачі через мережу, включаючи електронну пошту та хмарні сервіси.

**5.1 Вимоги до шифрування файлів**

При шифруванні файлів необхідно враховувати:

* Стійкість алгоритму (AES-256, ChaCha20, RSA-2048 тощо).
* Захист ключа (не зберігати відкрито, використовувати пароль або асиметричне шифрування).
* Наявність випадкового IV (nonce) (щоб уникнути атаки на однакові дані).
* Механізм перевірки цілісності (щоб виявляти зміни в файлах).

Шифрування файлів має відмінності від шифрування тексту через їхній розмір та структуру. Файли можуть бути великими, тому режими потокового шифрування (CTR, OFB, CFB) дозволяють обробляти дані частинами. Деякі файли мають заголовки та метадані, які можуть видавати інформацію навіть без дешифрування. Деякі алгоритми (AES-GCM, ChaCha20-Poly1305) одночасно шифрують і перевіряють цілісність.

**5.2 Симетричне шифрування файлів**

Найпоширеніший підхід, коли один і той самий ключ використовується для шифрування та дешифрування.

Основні режими роботи AES для файлів:

| **Режим** | **Особливості** | **Чи забезпечує цілісність?** |
| --- | --- | --- |
| *CBC (Cipher Block Chaining)* | Підходить для великих файлів, вимагає паддінг (PKCS7). | Ні |
| *CTR (Counter Mode)* | Потокове шифрування, не потребує паддінгу. | Ні |
| *GCM (Galois Counter Mode)* | Шифрування + автентифікація (вбудований MAC). | Так |
| *OCB (Offset Codebook Mode)* | Оптимізований для AEAD. | Так |

**Приклад. AES-256-GCM**

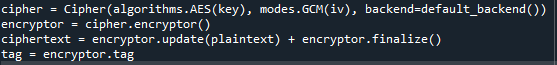
****

Рисунок 5.1. Приклад AES-256-GCM.

Використовує IV (Initialization Vector) для кожного шифрування. Генерує автентифікаційний тег (Tag) для перевірки достовірності. Якщо тег не співпадає, дешифрування неможливе.

**5.3 Гібридне шифрування файлів**

Змішує симетричне та асиметричне шифрування. Файл шифрується симетричним алгоритмом (AES, ChaCha20). Ключ шифрується асиметричним алгоритмом (RSA, ECDH). Це економить ресурси та дозволяє передавати зашифровані файли без розкриття ключа.

**Приклад. AES + RSA**

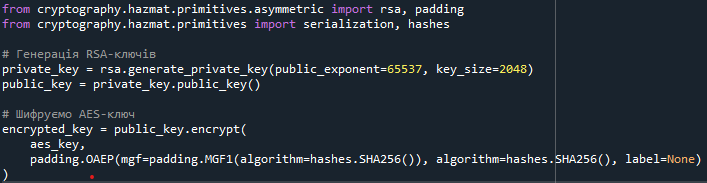


Рисунок 5.2. Приклад AES + RSA.

1. Генеруємо випадковий AES-ключ.
2. Шифруємо файл AES-256.
3. Шифруємо AES-ключ RSA-2048.
4. Передаємо обидва файли.
   1. **Перевірка цілісності файлів.**

Цілісність гарантує, що дані не були змінені або пошкоджені. Це особливо важливо для зашифрованих файлів, адже навіть одна змінена літера у файлі може зробити його повністю нерозшифрованим.

Методи перевірки цілісності.

| **Метод** | **Як працює** | **Додатковий ключ?** |
| --- | --- | --- |
| *HMAC (Hash-based MAC)* | Генерує MAC за допомогою секретного ключа. | Так |
| *AES-GCM / ChaCha20-Poly1305* | Вбудована перевірка під час дешифрування. | Так |
| *Цифровий підпис (RSA/ECDSA)* | Підписує хеш файлу, перевіряється відкритим ключем. | Так |

**Приклад. HMAC для перевірки файлу.**

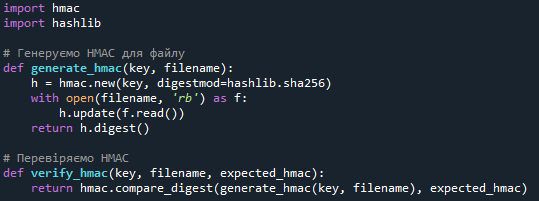


Рисунок 5.3. Приклад HMAC для перевірки файлу.

**Приклад. Цифровий підпис для файлів.**

Підписуємо SHA-256 хеш файлу закритим ключем RSA/ECDSA. Перевіряємо підпис відкритим ключем перед розшифруванням (рис.26):

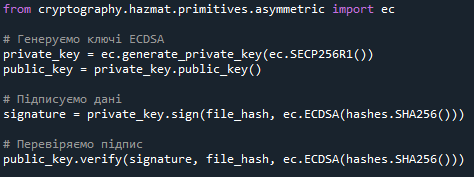
****

Рисунок 5.4. Приклад цифрового підпису для файлів.

**5.5 Приклад виконання**

У прикладі буде реалізовано симетричне шифрування файлу за допомогою AES-256 у режимі GCM (який забезпечує автентифікацію та перевірку цілісності).

**Генерація ключа.**

****

Рисунок 5.5. Приклад генерації ключа.

AES-256 вимагає ключа довжиною 32 байти (256 біт), тому ми використовуємо os.urandom(32) для створення випадкового ключа.

**Шифрування файлу.**

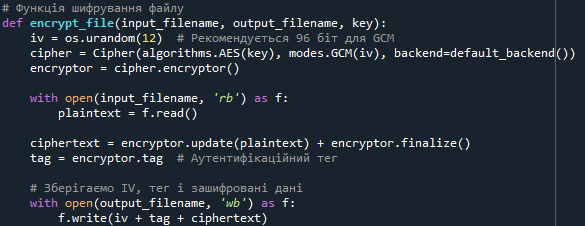


Рисунок 5.6. Приклад функції шифрування файлу.

***IV*** - унікальний випадковий вектор для кожного шифрування. ***GCM*** автоматично обчислює тег MAC для перевірки цілісності. ***update()*** шифрує дані, ***finalize()*** завершує процес. ***encryptor.tag*** містить тег ***MAC***, що використовується для перевірки достовірності. Записуємо у файл: ***IV*** (12 байтів), тег (16 байтів) та зашифровані дані.

**Дешифрування файлу:**

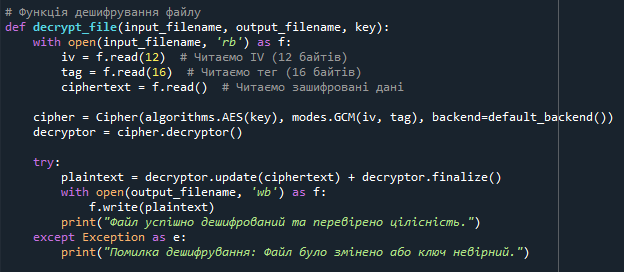
****

Рисунок 5.7. Приклад функції дешифрування файлу.

Зчитуємо *IV*, тег *MAC* та зашифровані дані. Створюємо об'єкт дешифрування з отриманими параметрами. Якщо тег не збігається, дешифрування не вдасться. Це означає що файл був змінений або невірний ключ.

**Лістинг**

import os

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default\_backend

# Генеруємо випадковий 256-бітний ключ

def generate\_key():

return os.urandom(32)

# Функція шифрування файлу

def encrypt\_file(input\_filename, output\_filename, key):

iv = os.urandom(12) # Рекомендується 96 біт для GCM

cipher = Cipher(algorithms.AES(key), modes.GCM(iv), backend=default\_backend())

encryptor = cipher.encryptor()

with open(input\_filename, 'rb') as f:

plaintext = f.read()

ciphertext = encryptor.update(plaintext) + encryptor.finalize()

tag = encryptor.tag # Аутентифікаційний тег

# Зберігаємо IV, тег і зашифровані дані

with open(output\_filename, 'wb') as f:

f.write(iv + tag + ciphertext)

# Функція дешифрування файлу

def decrypt\_file(input\_filename, output\_filename, key):

with open(input\_filename, 'rb') as f:

iv = f.read(12) # Читаємо IV (12 байтів)

tag = f.read(16) # Читаємо тег (16 байтів)

ciphertext = f.read() # Читаємо зашифровані дані

cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag), backend=default\_backend())

decryptor = cipher.decryptor()

try:

plaintext = decryptor.update(ciphertext) + decryptor.finalize()

with open(output\_filename, 'wb') as f:

f.write(plaintext)

print("Файл успішно дешифрований та перевірено цілісність.")

except Exception as e:

print("Помилка дешифрування: Файл було змінено або ключ невірний.")

# Використання

key = generate\_key()

encrypt\_file('example.txt', 'example\_encrypted.bin', key)

decrypt\_file('example\_encrypted.bin', 'example\_decrypted.txt', key)

**Питання для самоконтролю**

1. Чому шифрування файлів є важливим? У яких випадках воно застосовується?
2. Які особливості шифрування файлів у порівнянні з шифруванням тексту?
3. Які симетричні алгоритми шифрування використовуються для захисту файлів?
4. Які переваги має AES у режимі GCM у порівнянні з CBC або CTR?
5. Що таке Initialization Vector (IV) і чому його потрібно змінювати для кожного шифрування?
6. Як працює гібридне шифрування? Чому комбінація AES та RSA є ефективною?
7. У чому перевага ChaCha20-Poly1305 перед AES?
8. Які методи перевірки цілісності файлів використовуються в криптографії?
9. Як працює HMAC, і чому його не можна використовувати без секретного ключа?
10. Чим цифровий підпис відрізняється від HMAC у перевірці цілісності файлу?
11. Як працює AEAD (Authenticated Encryption with Associated Data) і які алгоритми його підтримують?
12. Яким чином можна перевірити, що файл після дешифрування не був змінений?
13. Чому RSA-2048 не використовується для прямого шифрування файлів?
14. Як можна безпечно передавати зашифрований файл через незахищений канал зв’язку?
15. Як захистити ключі шифрування при зберіганні та передачі файлів?

**Література: [1-8]**

# 6. ЛАБОРАТОРНЕ ЗАНЯТТЯ №6. Реалізація шифрування файлів

**Мета –** отриматипрактичні навички створення базових систем автентифікації з двофакторним захистом (2FA) на мові Python. Ознайомитися з принципами комбінування першого фактора (паролю) з додатковим фактором - тимчасовим кодом. Розвинути вміння проектувати графічні інтерфейси для безпечної взаємодії з користувачем у середовищі тестового застосунку.

**Завдання:**

1. Реалізувати **тестову систему автентифікації користувача** з графічним інтерфейсом на Python з використанням бібліотеки *tkinter*.
2. Створити два вікна:
   * Вікно ***реєстрації користувача***, в якому задається логін і пароль (що зберігається локально у захищеному вигляді).
   * Вікно ***входу* *(логіну)****,* в якому спочатку перевіряється логін/пароль, а потім виконується перевірка другого фактору.
3. Додати **другий фактор автентифікації** у вигляді тимчасового коду (TOTP) або симуляції коду (наприклад, випадкове значення, яке користувач отримує в окремому вікні або копіює).
4. Зберігати дані користувачів у **локальному файлі (наприклад, JSON)**, логін, хешований пароль, секрет для генерації коду 2FA (TOTP або іншого типу).
5. Використовувати **хешування паролів** з додаванням солі (*bcrypt* або *hashlib* + *os*.*urandom*).
6. Забезпечити **перевірку введених даних** та обробку типових помилок: неправильний пароль, неправильний код 2FA, порожні поля, повторна реєстрація існуючого користувача тощо.
7. Передбачити **гнучкість реалізації другого фактору**: час життя коду, кількість спроб, довжина, спосіб генерації тощо.

**Вимоги до завдання:**

* ***tkinter* для GUI** - мінімум два вікна (реєстрація та логін), поля для введення, кнопки, повідомлення.
* **Хешування паролів** з використанням *bcrypt*, або *hashlib* + сіль.
* **Зберігання користувачів** - у структурованому текстовому файлі (users.json), де кожен користувач має логін, хеш паролю, та параметри для другого фактору.
* **Другий фактор** реалізований у вигляді генератора коду (можна на основі *pyotp*, *secrets*, або власної функції).
* **Обробка помилок** - повідомлення про помилку прямо в GUI без краху програми.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом додатку.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. Тестування повинно налічувати як позитивні так і негативні сценарії. (Кожен скріншот повинен містити опис та підпис.)
5. Приклади вмісту вихідних, шифрованих та дешифрованих файлів, а також файлів паролів.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

Варіанти завдань до лабораторного заняття №6.

| № | Завдання |
| --- | --- |
| 1 | Використайте одноразовий 6-значний код як другий фактор автентифікації. Код дійсний протягом 30 секунд. Пароль має містити щонайменше 8 символів, включаючи цифри та великі літери. |
| 2 | Реалізуйте 2FA у вигляді секретної фрази (визначається користувачем під час реєстрації). При вході користувач повинен ввести фразу точно. Пароль має містити не менше 10 символів. |
| 3 | 2FA-код - 8-значний числовий код, що відображається у новому вікні. Код діє 45 секунд. Пароль має бути довжиною не менше 12 символів і містити спецсимвол. |
| 4 | Імітуйте надсилання коду через "електронну пошту": код не показується одразу, а відображається лише після натискання кнопки "отримати код". Код дійсний 30 секунд, 6 цифр. |
| 5 | При вході користувачу надається 3 одноразові коди. Один із них - правильний. Код показується в окремому вікні. Пароль має містити принаймні одну цифру і одну велику літеру. |
| 6 | Після введення паролю, код другого фактору надсилається (імітаційно) через текстове поле (типу "отримано на телефоні"). Код: 5 цифр, 30 секунд. |
| 7 | Реалізуйте 2FA через подвійний ввід паролю + підтвердження по 4-значному коду. Код показується лише після другого введення паролю. Обмеження на код: 60 секунд. |
| 8 | Замість випадкового коду використовуйте генерацію псевдо-TOTP (одноразовий код на основі часу, без бібліотек). Код змінюється кожні 60 секунд. Пароль: мін. 8 символів. |
| 9 | Після правильного паролю, користувач повинен вибрати правильну відповідь на секретне питання (встановлюється під час реєстрації). Пароль - мінімум 10 символів, без обмежень на склад. |
| 10 | Використайте 4-значний PIN-код як другий фактор. Код дійсний протягом 60 секунд. Пароль має бути не коротший за 6 символів і містити хоча б одну цифру. |

## Теоретичні відомості до виконання лабораторного заняття № 6.

Автентифікація (authentication) - це процес перевірки достовірності особи, яка намагається отримати доступ до системи. У найпростішому випадку автентифікація відбувається за допомогою логіна та пароля.

**6.1 Однофакторна vs. двофакторна автентифікація.**

Однофакторна автентифікація (1FA) - лише один фактор (зазвичай пароль).

Двофакторна автентифікація (2FA) - два незалежних фактори для більш надійного захисту.

Три категорії факторів:

| **Категорія** | **Приклад** |
| --- | --- |
| *Знання (Що ви знаєте)* | Пароль, PIN, секретне слово |
| *Володіння (Що ви маєте)* | Телефон, смарт-карта, токен |
| *Біометрія (Хто ви є)* | Відбиток пальця, обличчя |

2FA зазвичай поєднує пароль (знання) + код із пристрою або email (володіння).

**6.2 Алгоритм 2FA у простій системі та хешування паролів.**

1. Користувач вводить логін та пароль.
2. Якщо пароль правильний:

* Генерується одноразовий код (наприклад, 6-значне число).
* Користувач вводить цей код (з SMS, email, тощо).

1. Якщо код правильний - автентифікація успішна.

З міркувань безпеки ніколи не зберігаються паролі у відкритому вигляді. Замість цього пароль хешується за допомогою алгоритму (наприклад, *bcrypt*), разом із паролем використовується сіль (*salt*) - випадкове значення, яке унеможливлює атаки типу "rainbow table".

***OTP (One-Time Password)*** - код, що дійсний лише один раз і лише певний час.Типи:

* HOTP (HMAC-based OTP) - базується на лічильнику;
* TOTP (Time-based OTP) - базується на поточному часі.

У цій роботі реалізовується спрощений варіант, випадковий числовий код з обмеженим часом дії (30–60 сек).

**6.3 Створення GUI з tkinter.**

**tkinter** - стандартна бібліотека Python для створення графічного інтерфейсу. Базові елементи:

* ***Label*** - текст.
* ***Entry*** - поле вводу.
* ***Button*** - кнопка.
* ***messagebox*** - спливаючі повідомлення (помилки, успіх).
* ***pack() / grid() / place()*** - розміщення елементів.

Приклад створення вікна:

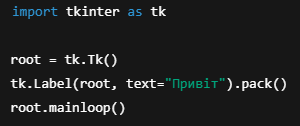


Рисунок 6.1. Приклад коду створення вікна.

**6.4 Збереження даних у JSON.**

Дані користувачів можуть зберігаютися у файлі (наприклад, users.json)

Приклад, users.json файлу:

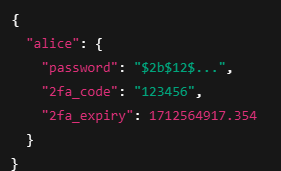


Рисунок 6.2. Приклад HMAC для перевірки файлу.

Робота з JSON:

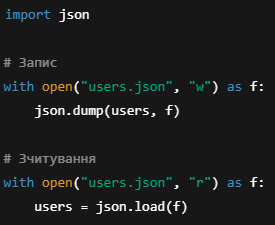
****

Рисунок 6.3. Приклад коду роботи з JSON.

Поради щодо безпеки:

* Завжди хешуйте паролі.
* Не зберігайте 2FA-коди довше, ніж потрібно.
* Не дозволяйте необмежену кількість спроб введення коду.
* Очищуйте поля вводу після кожної спроби.

**6.5 Приклад виконання**

Цей приклад реалізує тестову 2FA-систему з графічним інтерфейсом на базі *tkinter*, зберігає облікові дані в *users*.*json*, використовує *bcrypt* для хешування паролів і симульований другий фактор - одноразовий код, який генерується при логіні та відображається в окремому вікні (імітація SMS або додатку-генератора).

Структура проєкту.

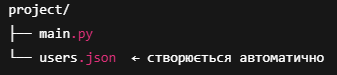
****

Рисунок 6.4. Структура проєкту.

Імпорт бібліотек і константи.

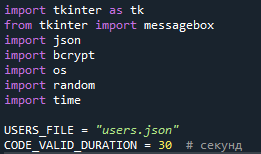


Рисунок 6.5. Приклад імпорту бібліотек і встановлення констант.

* ***tkinter*** - бібліотека для створення GUI.
* ***messagebox*** - вікна з повідомленнями (помилки, підтвердження).
* ***json*** - для зберігання даних користувачів у файлі.
* ***bcrypt*** - для безпечного хешування паролів.
* ***os*** - перевірка існування файлу.
* ***random*** - генерація 2FA-коду.
* ***time*** - перевірка часу дії коду.
* ***CODE\_VALID\_DURATION -*** час, протягом якого код 2FA є дійсним.

Функції для роботи з файлом користувачів:

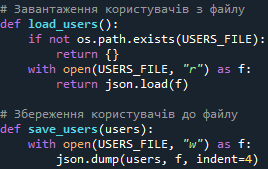
****

Рисунок 6.6. Приклад функції для роботи з файлом користувачів.

* ***load\_users()*** - читає файл users.json і повертає словник з даними користувачів.
* ***save\_users(users***) - записує словник користувачів у файл.

Якщо файл ще не створено - повертається порожній словник.

**Клас User, логіка користувача:**

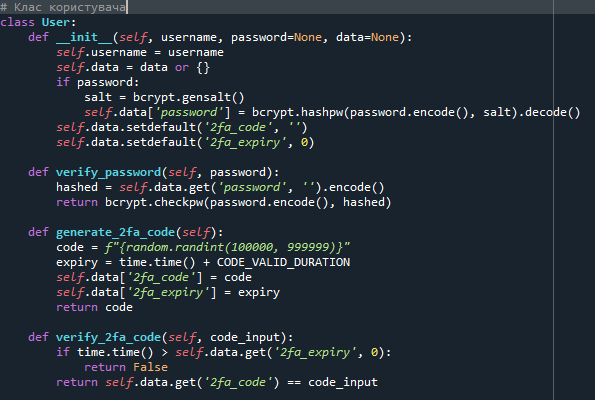
****

Рисунок 6.7. Приклад класу User.

***\_init\_()*** - ініціалізує об'єкт користувача. Якщо передано password, створюється новий хеш з сіллю і записується в поле password. Якщо data вже є (наприклад, при вході) - використовуються наявні дані. ***2fa\_code і 2fa\_expiry*** ініціалізуються, якщо не задані (для генерації коду з обмеженим терміном дії).

***verify\_password(self, password) -*** перевіряє, чи введений пароль відповідає збереженому хешу. Функція ***bcrypt.checkpw()*** автоматично обробляє сіль і хеш.

***generate\_2fa\_code(self) -*** Генерує 6-значний випадковий код. Зберігає його в полі 2fa\_code. Також зберігає час закінчення дії (2fa\_expiry). Повертає згенерований код, щоб показати його в GUI.

***verify\_2fa\_code(self, code\_input) -*** Перевіряє, чи введений код збігається з тим, що був згенерований. Якщо поточний час більше *2fa*\_*expiry* - код уже недійсний.

**Клас AuthApp, логіка інтерфейсу:**

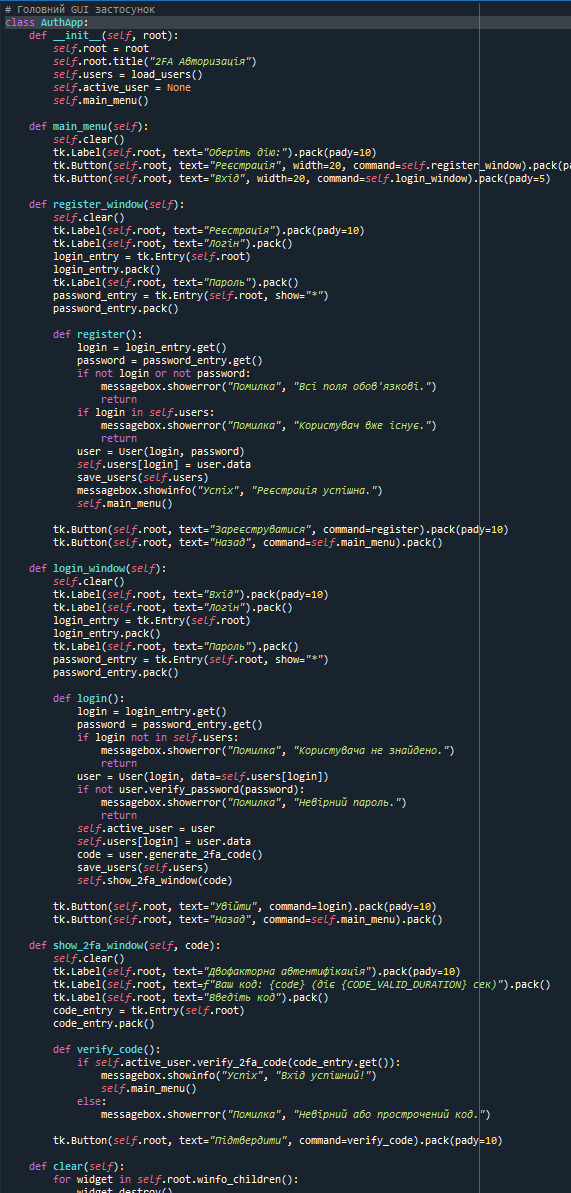
****

Рисунок 6.8. Приклад класу AuthApp.

***\_\_init\_\_(self, root). root*** - вікно tkinter. Завантажуються користувачі з файлу. ***active\_user*** - поточний користувач, який намагається увійти. Показується головне меню (реєстрація або вхід).

***main\_menu(self)*** - вікно з кнопками для переходу до вікон реєстрації та входу.

***register\_window(self) -*** вікно для реєстрації нового користувача. Перевіряє, чи: заповнені всі поля, користувач вже існує. Якщо все добре - створюється об'єкт ***User***, додається до словника, зберігається у файл.

***login\_window(self) -*** вікно для входу користувача. Перевіряє: чи існує логін, чи правильний пароль. Якщо так - створюється об'єкт користувача, генерується 2FA-код, запускається нове вікно підтвердження.

***show\_2fa\_window(self, code) -*** відображає код другого фактору (як ніби прийшов по SMS або email). Пропонує користувачу ввести цей код. Якщо введено правильно - виводиться повідомлення про успішний вхід.

***clear(self) -*** очищає вікно перед завантаженням нового. Забезпечує "перемикання" між екранами у межах одного GUI.

**Запуск застосунку:**

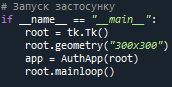
****

Рисунок 6.9. Приклад коду для запуску застосунку.

Цей код запускає графічний інтерфейс. Створюється вікно з розміром 300x300 пікселів. Створюється об'єкт застосунку AuthApp.

**6.6 Приклади тестування**

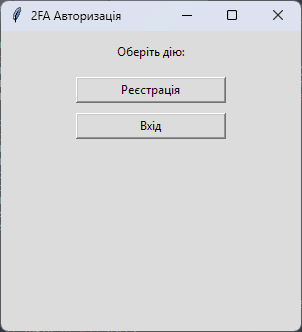


Рисунок 6.10. Приклад головного меню застосунку.

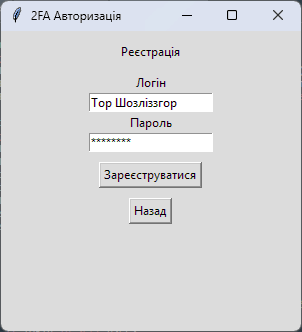
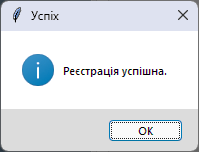
 

Рисунок 6.11. Приклад реєстрації у застосунку.

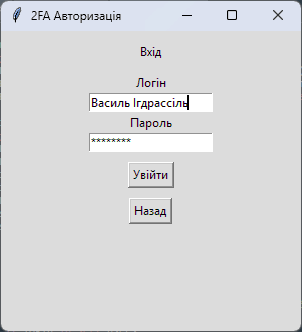
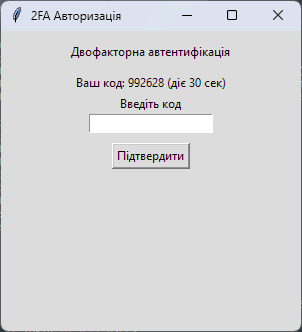
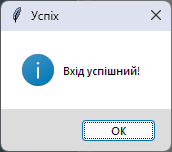
  

Рисунок 6.12. Приклад успішного входу до застосунку.

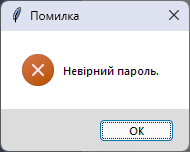
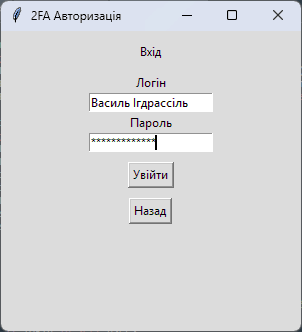


Рисунок 6.13. Приклад помилки при вводі паролю.

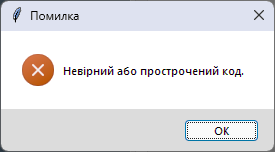
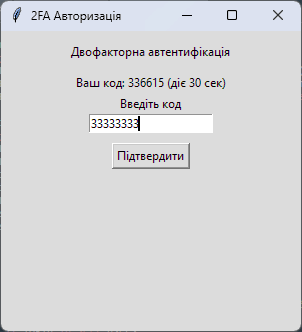


Рисунок 6.14. Приклад помилки при вводі коду.

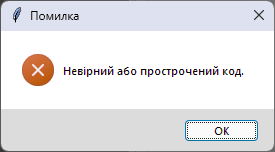
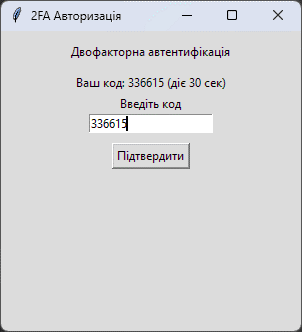


Рисунок 6.15. Приклад помилки при завершенні строку дії тимчасового пароля.

Вміст файлу ***users.json:***

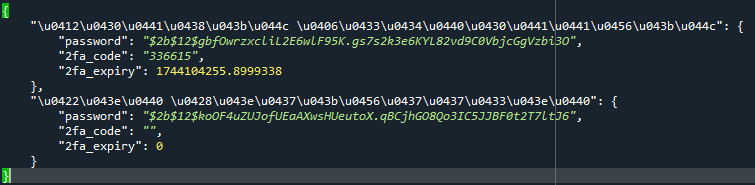


Рисунок 6.16. Приклад вмісту файлу users.json.

**Лістинг**

import tkinter as tk

from tkinter import messagebox

import json

import bcrypt

import os

import random

import time

USERS\_FILE = "users.json"

CODE\_VALID\_DURATION = 30 # секунд

# Завантаження користувачів з файлу

def load\_users():

if not os.path.exists(USERS\_FILE):

return {}

with open(USERS\_FILE, "r") as f:

return json.load(f)

# Збереження користувачів до файлу

def save\_users(users):

with open(USERS\_FILE, "w") as f:

json.dump(users, f, indent=4)

# Клас користувача

class User:

def \_\_init\_\_(self, username, password=None, data=None):

self.username = username

self.data = data or {}

if password:

salt = bcrypt.gensalt()

self.data['password'] = bcrypt.hashpw(password.encode(), salt).decode()

self.data.setdefault('2fa\_code', '')

self.data.setdefault('2fa\_expiry', 0)

def verify\_password(self, password):

hashed = self.data.get('password', '').encode()

return bcrypt.checkpw(password.encode(), hashed)

def generate\_2fa\_code(self):

code = f"{random.randint(100000, 999999)}"

expiry = time.time() + CODE\_VALID\_DURATION

self.data['2fa\_code'] = code

self.data['2fa\_expiry'] = expiry

return code

def verify\_2fa\_code(self, code\_input):

if time.time() > self.data.get('2fa\_expiry', 0):

return False

return self.data.get('2fa\_code') == code\_input

# Головний GUI застосунок

class AuthApp:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("2FA Авторизація")

self.users = load\_users()

self.active\_user = None

self.main\_menu()

def main\_menu(self):

self.clear()

tk.Label(self.root, text="Оберіть дію:").pack(pady=10)

tk.Button(self.root, text="Реєстрація", width=20, command=self.register\_window).pack(pady=5)

tk.Button(self.root, text="Вхід", width=20, command=self.login\_window).pack(pady=5)

def register\_window(self):

self.clear()

tk.Label(self.root, text="Реєстрація").pack(pady=10)

tk.Label(self.root, text="Логін").pack()

login\_entry = tk.Entry(self.root)

login\_entry.pack()

tk.Label(self.root, text="Пароль").pack()

password\_entry = tk.Entry(self.root, show="\*")

password\_entry.pack()

def register():

login = login\_entry.get()

password = password\_entry.get()

if not login or not password:

messagebox.showerror("Помилка", "Всі поля обов'язкові.")

return

if login in self.users:

messagebox.showerror("Помилка", "Користувач вже існує.")

return

user = User(login, password)

self.users[login] = user.data

save\_users(self.users)

messagebox.showinfo("Успіх", "Реєстрація успішна.")

self.main\_menu()

tk.Button(self.root, text="Зареєструватися", command=register).pack(pady=10)

tk.Button(self.root, text="Назад", command=self.main\_menu).pack()

def login\_window(self):

self.clear()

tk.Label(self.root, text="Вхід").pack(pady=10)

tk.Label(self.root, text="Логін").pack()

login\_entry = tk.Entry(self.root)

login\_entry.pack()

tk.Label(self.root, text="Пароль").pack()

password\_entry = tk.Entry(self.root, show="\*")

password\_entry.pack()

def login():

login = login\_entry.get()

password = password\_entry.get()

if login not in self.users:

messagebox.showerror("Помилка", "Користувача не знайдено.")

return

user = User(login, data=self.users[login])

if not user.verify\_password(password):

messagebox.showerror("Помилка", "Невірний пароль.")

return

self.active\_user = user

self.users[login] = user.data

code = user.generate\_2fa\_code()

save\_users(self.users)

self.show\_2fa\_window(code)

tk.Button(self.root, text="Увійти", command=login).pack(pady=10)

tk.Button(self.root, text="Назад", command=self.main\_menu).pack()

def show\_2fa\_window(self, code):

self.clear()

tk.Label(self.root, text="Двофакторна автентифікація").pack(pady=10)

tk.Label(self.root, text=f"Ваш код: {code} (діє {CODE\_VALID\_DURATION} сек)").pack()

tk.Label(self.root, text="Введіть код").pack()

code\_entry = tk.Entry(self.root)

code\_entry.pack()

def verify\_code():

if self.active\_user.verify\_2fa\_code(code\_entry.get()):

messagebox.showinfo("Успіх", "Вхід успішний!")

self.main\_menu()

else:

messagebox.showerror("Помилка", "Невірний або прострочений код.")

tk.Button(self.root, text="Підтвердити", command=verify\_code).pack(pady=10)

def clear(self):

for widget in self.root.winfo\_children():

widget.destroy()

# Запуск застосунку

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

root.geometry("300x300")

app = AuthApp(root)

root.mainloop()

**Питання для самоконтролю**

1. Що таке автентифікація? Чим вона відрізняється від авторизації?
2. Які існують основні типи факторів автентифікації? Наведіть приклади для кожного.
3. У чому полягає перевага двофакторної автентифікації над однофакторною?
4. Що таке одноразовий пароль (OTP)? Які типи OTP існують?
5. Як працює Time-based OTP (TOTP)? У чому його принцип?
6. Для чого використовується хешування паролів? Чим небезпечно зберігати паролі у відкритому вигляді?
7. Яка роль “солі” (salt) у хешуванні паролів? Як вона захищає від атак типу "rainbow table"?
8. Який алгоритм хешування паролів був використаний у вашій реалізації? Які його переваги?
9. Як реалізується перевірка терміну дії одноразового коду у Python?
10. Якими способами можна обмежити кількість спроб введення 2FA-коду?
11. Які переваги має зберігання облікових даних у форматі JSON? Які є альтернативи?
12. Які елементи інтерфейсу tkinter ви використали для реалізації GUI? Як вони працюють?
13. У чому різниця між методами pack(), grid() та place() в tkinter?
14. Як можна реалізувати таймер зворотного відліку дії 2FA-коду у графічному інтерфейсі?
15. Які є загальні принципи безпеки при розробці систем автентифікації?

**Література: [1-8]**

# 7. ЛАБОРАТОРНЕ ЗАНЯТТЯ №7. Реалізація цифрового сертифіката та PKI

**Мета –** ознайомитись з концепцією інфраструктури відкритих ключів (PKI), принципами довіри між суб'єктами в цифровому середовищі, створенням, підписанням і верифікацією цифрових сертифікатів, а також з основами роботи сертифікаційних центрів (CA) – як кореневих, так і проміжних.

**Завдання:**

1. Створити кореневий сертифікаційний центр (Root CA). Згенерувати пару ключів (приватний та публічний). Створити самопідписаний сертифікат для Root CA.
2. Створити проміжний сертифікаційний центр (Intermediate CA). Згенерувати ключі. Сформувати запит на підпис сертифіката (CSR). Підписати цей запит Root CA, сформувавши сертифікат Intermediate CA.
3. Видати сертифікат для кінцевого суб'єкта (користувача або сервісу). Згенерувати ключі. Сформувати CSR. Підписати його Intermediate CA. Додати специфічні розширення залежно від варіанту (наприклад, KeyUsage, ExtendedKeyUsage, SubjectAlternativeName тощо).
4. Побудувати повний ланцюжок сертифікатів (сертифікат користувача + Intermediate CA + Root CA).
5. Верифікувати кінцевий сертифікат. Програмно перевірити підписи в сертифікатах. Перевірити відповідність ключів. Перевірити терміни дії.
6. Продемонструвати застосування сертифіката для перевірки підпису або для шифрування повідомлення.

**Вимоги до завдання:**

* Усі ключі та сертифікати мають зберігатися у форматі PEM.
* Сертифікати повинні містити відповідні розширення, залежно від ролі (CA, кінцевий суб’єкт).
* Підписи повинні створюватися з використанням SHA256.
* Реалізована програмна генерація: ключів (RSA, 2048 або 4096 біт), сертифікатів, запитів CSR.
* Реалізована перевірка: валідності ланцюжка довіри, часових меж дії сертифіката, відповідності публічного ключа сертифікату.

**Обмеження:**

* Заборонено використовувати інтернет–сервіси або сторонні CA.
* Усі дані мають бути згенеровані локально, без залежності від зовнішніх PKI.

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом додатку.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. Тестування повинно налічувати як позитивні так і негативні сценарії. (Кожен скріншот повинен містити опис та підпис.)
5. Приклади вмісту вихідних, шифрованих та дешифрованих файлів, а також файлів паролів.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

Варіанти завдань до лабораторного заняття №7.

| № | Назва організації | Common Name | Призначення сертифіката | Валідність | Розмір ключа | SAN/Домен |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | SecureSoft | Alice SecureMail | Email–клієнт | 1 рік | 2048 | alice.securemail.com |
| 2 | TechTrust Ltd | Bob WebAPI | API–сервер | 2 роки | 4096 | api.techtrust.com |
| 3 | NovaCrypt | Admin Panel | Веб–інтерфейс адміністратора | 6 місяців | 2048 | admin.novacrypt.ua |
| 4 | UniCert | Student Auth | Клієнтська автентифікація | 1 рік | 2048 | student.unicert.edu |
| 5 | GovCA | State Registry | Державний реєстр | 3 роки | 3072 | registry.gov.ua |
| 6 | HealthSec | ClinicNode01 | IoT–пристрій медзакладу | 2 роки | 2048 | device01.clinic.local |
| 7 | DevLabs | CI/CD Agent | CI–сервер (GitLab Runner) | 1 рік | 2048 | ci.devlabs.net |
| 8 | EduRoot | Teacher Portal | Веб–портал викладача | 2 роки | 4096 | portal.eduroot.org |
| 9 | AutoCert | Car ECU | Автомобільна електроніка | 5 років | 2048 | car–unit01.local |
| 10 | SmartGrid PKI | MeterNode–123 | Smart Grid пристрій | 3 роки | 3072 | smart.meter.local |
| 11 | ChainID | IdentityService | Служба ідентифікації | 2 роки | 4096 | identity.chainid.net |
| 12 | MedDataTrust | Patient Records Service | Сервіс обробки медичних даних | 1 рік | 2048 | records.medtrust.org |
| 13 | SafeCampus | Dorm Access Controller | Контролер доступу гуртожитку | 2 роки | 2048 | access.safecampus.edu |
| 14 | BlockAuth | Blockchain Signing Node | Нода блокчейну для підпису | 3 роки | 4096 | node.blockauth.io |
| 15 | SecureBuild | BuildVerifier | Служба перевірки артефактів | 1 рік | 3072 | verify.securebuild.dev |

## Теоретичні відомості до виконання лабораторного заняття № 7.

1. **Загальні поняття Public Key Infrastructure (PKI).**

PKI (інфраструктура відкритих ключів, англ. Public Key Infrastructure) – це сукупність апаратних, програмних і організаційних засобів, яка забезпечує керування криптографічними ключами та випуск, розповсюдження і перевірку цифрових сертифікатів. PKI дозволяє забезпечити:

* Шифрування даних (конфіденційність),
* Цілісність (неможливість змін без виявлення),
* Автентифікацію користувачів та пристроїв,
* Незаперечність (підпис не можна заперечити після створення).

Основна мета PKI – надати довіру між учасниками цифрового обміну (користувачами, сервісами, пристроями), навіть якщо вони не знайомі між собою безпосередньо. Це досягається шляхом централізованої чи ієрархічної моделі довіри через сертифікати.

**Основні компоненти PKI:**

* Сертифікаційний центр (CA, Certificate Authority). Генерує та підписує сертифікати. Забезпечує довіру до ідентичності об'єктів.
* Реєстраційний центр (RA, Registration Authority). Підтверджує ідентичність заявника перед видачею сертифіката. RA часто делегується CA.
* Кінцеві суб’єкти (End Entities). Користувачі, пристрої, сервери, які використовують сертифікати (браузери, API–сервери, поштові клієнти, IoT–пристрої).
* Цифрові сертифікати. Сертифікат – це підписане повідомлення, яке прив'язує відкритий ключ до конкретного суб’єкта (людини, сервісу, пристрою). Основою для сертифіката є запит на сертифікат (CSR).
* Сховище довірених сертифікатів (Trust Store). Набір Root CA сертифікатів, яким система довіряє за замовчуванням (в ОС, браузерах тощо).
* Списки відкликання сертифікатів (CRL) / OCSP. CRL – це список сертифікатів, які були відкликані до завершення строку дії. OCSP (Online Certificate Status Protocol) – онлайн–перевірка статусу сертифіката.

**Ключові поняття.** Відкритий ключ використовується для шифрування даних або перевірки цифрового підпису. Приватний ключ використовується для розшифрування або створення цифрового підпису. Цифровий підпис – результат криптографічного хешування та шифрування повідомлення приватним ключем. Ланцюг довіри (*chain of trust*) – послідовність сертифікатів, де кожен наступний сертифікат підписаний попереднім.

PKI широко використовується у:

* Протоколах TLS/SSL для безпечного з'єднання (HTTPS),
* Електронному підписі документів (PDF, e–doc),
* VPN та Wi–Fi автентифікації (наприклад, EAP–TLS),
* Автентифікації пристроїв в IoT,
* Цифрових посвідченнях особи, паспортах, банківських системах,
* Контейнеризації та DevOps (підписування образів, Jenkins, GitHub Actions).

1. **Структура цифрового сертифіката.**

Цифровий сертифікат – це електронний документ, який засвідчує автентичність публічного ключа. Найпоширенішим стандартом сертифікатів є X.509, який використовується в TLS/SSL, електронному підписі, VPN, криптографії IoT–пристроїв тощо.

**Основні функції сертифіката:**

* Ідентифікація суб’єкта (користувача, сервісу, пристрою).
* Надання довіри до відкритого ключа.
* Визначення прав і призначення ключа.
* Установлення терміну дії ключа.
* Можливість перевірки справжності через ланцюжок сертифікації.

**Основні поля X.509–сертифіката:**

| **Поле** | **Опис** |
| --- | --- |
| *Version* | Версія формату сертифіката (переважно v3). |
| *Serial Number* | Унікальний серійний номер, який надає CA. |
| *Signature Algorithm* | Алгоритм, яким CA підписав сертифікат (наприклад, SHA256withRSA). |
| *Issuer* | Інформація про організацію, що видала сертифікат (CA). |
| *Validity* | Період дії сертифіката – notBefore та notAfter. |
| *Subject* | Ідентифікатор власника сертифіката (Common Name, Organization, Email тощо). |
| *Subject Public Key Info* | Публічний ключ користувача або пристрою. |
| *Extensions* | Додаткові поля, які описують права, обмеження та призначення сертифіката. |

**Приклад – структура сертифіката у вигляді схеми.**

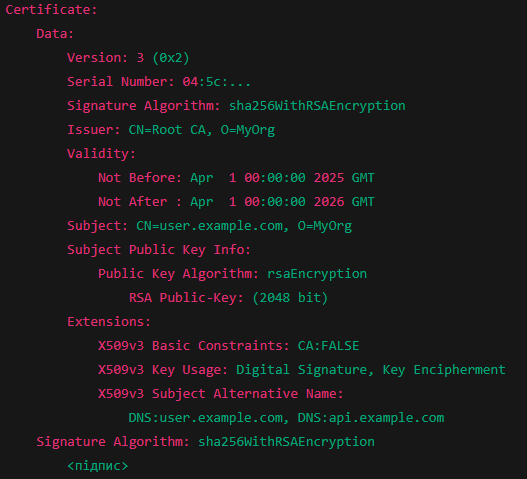
******

Рисунок 7.1. Приклад сертифіката у вигляді схеми.

Сертифікат не містить приватного ключа! Приватний ключ зберігається окремо та використовується для створення CSR (запиту на сертифікат) та підпису повідомлень або транзакцій.

**Найпоширеніші X.509–розширення (Extensions).**

* ***Basic Constraints.*** Визначає, чи може сертифікат бути CA. CA = TRUE – якщо сертифікат може видавати інші сертифікати.
* ***Key Usage.*** Обмежує, які операції дозволено виконувати з ключем: digitalSignature, keyEncipherment, keyCertSign (потрібне для CA), crlSign
* ***Extended Key Usage (EKU).*** Уточнює призначення сертифіката: serverAuth – для HTTPS–серверів, clientAuth – для автентифікації клієнтів, emailProtection, codeSigning, тощо.
* ***Subject Alternative Name (SAN).*** Альтернативні імена: домени, IP, email, URI. Сучасні клієнти (наприклад, браузери) використовують саме SAN для перевірки відповідності сертифіката URL–адресі.
* ***Authority Key Identifier / Subject Key Identifier.*** Дозволяє побудувати ланцюжок довіри, визначаючи, який ключ підписав сертифікат (AKI) і який ключ у самого сертифіката (SKI).
* ***CRL Distribution Points / OCSP.*** Місце, звідки можна завантажити список відкликаних сертифікатів (CRL) або отримати відповідь OCSP.

1. **Роль сертифікаційного центру (CA).**

Сертифікаційний центр (CA, Certification Authority) – це організація або система, що видає цифрові сертифікати. CA виступає як довірений третій учасник в інфраструктурі відкритих ключів (PKI) і забезпечує механізм, за допомогою якого користувачі та системи можуть вірити в автентичність публічних ключів. CA підтверджує особу власника публічного ключа або сервісу, підписуючи сертифікат, що містить цей публічний ключ.

**Сертифікаційний центр виконує кілька важливих функцій:**

* ***Видача сертифікатів.*** CA перевіряє ідентичність запитувача (користувача або сервісу), і на основі цієї перевірки видає цифровий сертифікат. Сертифікат містить публічний ключ, підписаний приватним ключем CA, що підтверджує його автентичність.
* ***Перевірка ідентичності.*** Для кожного запиту на сертифікат (CSR – Certificate Signing Request) сертифікаційний центр може проводити перевірку через реєстраційний центр (RA), який відповідає за збір і верифікацію інформації про власника сертифіката.
* ***Підтвердження довіри.*** Видаючи сертифікат, CA підтверджує, що публічний ключ належить зазначеному суб'єкту. Ця перевірка здійснюється через підпис сертифіката приватним ключем CA, що дозволяє іншим учасникам PKI бути впевненими в автентичності ключа.

Існують різні типи сертифікаційних центрів в залежності від їхнього рівня довіри та застосування:

* ***Root CA***. Це основний сертифікаційний центр в ієрархії PKI, якому довіряють всі інші учасники системи. Root CA самопідписаний, і його публічний ключ використовується для перевірки сертифікатів, підписаних нижчими рівнями CA (Intermediate CA або кінцевими користувачами).
* ***Intermediate CA.*** Проміжні сертифікаційні центри підписуються Root CA і використовуються для забезпечення додаткової гнучкості та безпеки. Проміжний CA може видавати сертифікати користувачам або сервісам, але не має повної довіри, як Root CA.
* ***Issuing CA.*** Це сертифікаційні центри, що видають сертифікати кінцевим користувачам або пристроям. Вони зазвичай підписуються Intermediate CA, який у свою чергу може бути підписаний Root CA.

Процес видачі сертифіката через сертифікаційний центр виглядає наступним чином:

1. ***Запит на сертифікат (CSR).*** Користувач або пристрій створює пару ключів (приватний і публічний ключ), а потім формує запит на підписання сертифіката (CSR), що містить публічний ключ та іншу ідентифікаційну інформацію.
2. ***Перевірка CA.*** Сертифікаційний центр перевіряє запит на відповідність вимогам, що включають перевірку ідентичності користувача або пристрою. Це може включати перевірку через зовнішні джерела, такі як документи, реєстраційні дані або DNS–записи.
3. ***Підписання сертифіката.*** Після успішної перевірки CA підписує сертифікат, що містить публічний ключ ідентифікованого користувача або сервісу. Цей підпис здійснюється приватним ключем CA.
4. ***Поширення сертифіката.*** Після підписання сертифікат відправляється користувачу або серверу для подальшого використання. Сертифікат може містити також ланцюжок сертифікатів (для Intermediate CA), щоб показати довіру до підписаного сертифіката.

Підпис сертифіката здійснюється приватним ключем сертифікаційного центру і служить для підтвердження достовірності сертифіката. Для перевірки достовірності сертифіката користувачі можуть використовувати публічний ключ Root CA, який є відомим та загальнодоступним. Це забезпечує ланцюжок довіри, коли кожен сертифікат може бути перевірений за допомогою сертифіката вищого рівня в ієрархії CA.

Безпека сертифікаційного центру є критично важливою для забезпечення довіри всієї інфраструктури PKI. Для забезпечення безпеки використовується кілька методів:

* ***Офлайн зберігання Root CA.*** Root CA зазвичай зберігається офлайн, щоб мінімізувати ризик його компрометації. Це може бути фізичний пристрій або комп'ютер, що не підключений до інтернету.
* ***Захист приватних ключів.*** Приватні ключі сертифікаційних центрів повинні бути зберігані в безпечних середовищах, таких як спеціалізовані апаратні модулі (HSM).
* ***Регулярні аудити.*** Сертифікаційні центри підлягають регулярним аудитам для перевірки їхньої діяльності та безпеки.

У разі компрометації ключа або змін в умовах використання сертифіката, сертифікаційний центр може відкликати сертифікат. Для цього існують два основних механізми:

* ***CRL (Certificate Revocation List).*** Список відкликаних сертифікатів, який публікується сертифікаційним центром.
* ***OCSP (Online Certificate Status Protocol).*** Протокол для перевірки статусу сертифіката в режимі реального часу.

1. **Цикл життя сертифіката.**

Цикл життя сертифіката охоплює всі етапи, починаючи від його створення до моменту закінчення терміну дії або анулювання. Це важливий процес у PKI, що гарантує надійність і безпеку використання цифрових сертифікатів. Він включає кілька ключових етапів:

1. ***Генерація ключової пари.*** Перед тим, як отримати сертифікат, користувач або пристрій генерує пару ключів: приватний і публічний. Приватний ключ зберігається в безпечному місці (наприклад, на пристрої користувача або в апаратному модулі безпеки), а публічний ключ буде включений у сертифікат. Ці ключі використовуються для криптографічних операцій, таких як підпис повідомлень або перевірка підпису.
2. ***Створення запиту на сертифікат (CSR).*** Запит на сертифікат (Certificate Signing Request, CSR) є запитом до сертифікаційного центру на видачу сертифіката, що містить публічний ключ, а також ідентифікаційні дані про власника сертифіката. CSR зазвичай створюється на основі публічного ключа і містить такі дані: Common Name (CN), організація, підрозділ, місто, країна, публічний ключ, підпис користувача, що підтверджує запит.
3. ***Підписання сертифіката сертифікаційним центром.*** Після того як CSR надійшов до сертифікаційного центру, він перевіряє автентичність запиту та надає підпис. Підписання сертифіката підтверджує, що публічний ключ в сертифікаті дійсно належить вказаному власнику і що сертифікат відповідає стандартам PKI. Після підписання сертифікат отримує цифровий підпис сертифікаційного центру (CA), що гарантує його автентичність.
4. ***Видача сертифіката користувачеві або пристрою.*** Після того як сертифікат підписаний, він передається користувачу або пристрою, який подавав CSR. Сертифікат містить публічний ключ, дані про організацію та сертифікаційну інформацію про центр підпису. Зазначимо, що сертифікати можуть бути самопідписаними (Root CA) або підписаними іншими сертифікаційними центрами (Intermediate CA).
5. ***Використання сертифіката.*** Після отримання сертифіката користувач може використовувати його для різних цілей: підписування повідомлень (цифрові підписи), захищене з'єднання через HTTPS, встановлення VPN з'єднань, аутентифікація пристроїв чи користувачів у системах.
6. ***Перевірка сертифіката.*** Для забезпечення безпеки і довіри до сертифікатів необхідно перевірити їх достовірність. Перевірка включає: валідацію терміну дії сертифіката, перевірку підпису сертифікаційного центру, перевірку ланцюга довіри (Root CA → Intermediate CA → Кінцевий сертифікат), перевірку поля Subject Alternative Name (SAN) (якщо використовується), перевірку анульованих сертифікатів через CRL (Certificate Revocation List) або OCSP (Online Certificate Status Protocol).
7. ***Анулювання сертифіката.*** Коли сертифікат більше не є дійсним, його може бути анульовано. Причини анулювання включають: порушення безпеки (наприклад, компрометація приватного ключа), технічні помилки у сертифікаті, закінчення терміну дії сертифіката, зміна даних власника сертифіката. Анулювання сертифікатів здійснюється через CRL (Certificate Revocation List) – список анульованих сертифікатів, що регулярно оновлюється сертифікаційним центром, або за допомогою OCSP, який дає можливість отримувати інформацію про статус сертифіката в реальному часі.
8. ***Перевипуск сертифіката.*** Іноді сертифікат потрібно перевипустити. Це може статися, якщо змінюється інформація у сертифікаті (наприклад, назва організації або домен) або сертифікат підходить до кінця терміну дії. Процес перевипуску аналогічний початковому, включаючи створення нового CSR і підписання сертифіката сертифікаційним центром.
9. ***Завершення терміну дії сертифіката.*** Кожен сертифікат має визначений термін дії. Після його завершення сертифікат більше не є дійсним. Важливо своєчасно перевіряти дату закінчення терміну дії сертифікатів і оновлювати їх до завершення терміну.
10. Цикл життя сертифіката включає не тільки створення та підписання, а й активне використання, перевірку на достовірність та можливість анулювання. Кожен етап є важливим для забезпечення безпеки цифрових транзакцій і автентичності в цифровому світі.
11. **Сценарії застосування PKI.**

Public Key Infrastructure (PKI) – це основа для багатьох криптографічних технологій, що забезпечують безпеку комунікацій та аутентифікацію в сучасних інформаційних системах. PKI активно використовується в різних сферах, від веб–безпеки до захисту даних у мережах і забезпечення автентичності програмного забезпечення. Розглянемо деякі з основних сценаріїв застосування PKI.

Одним з найпоширеніших застосувань PKI є захист веб–трафіку за допомогою протоколу HTTPS (HyperText Transfer Protocol Secure). У цьому випадку PKI забезпечує захист конфіденційності та цілісності даних, що передаються між веб–сервером і клієнтом (браузером). Веб–сайт видає сертифікат для підтвердження своєї автентичності. Клієнт перевіряє підпис сертифіката за допомогою публічного ключа відомого сертифікаційного центру (CA). Якщо перевірка успішна, встановлюється захищене з'єднання між браузером та сервером за допомогою протоколу TLS/SSL. Важливість:

* Забезпечує конфіденційність даних, використовуючи шифрування.
* Забезпечує цілісність даних, гарантуючи, що інформація не була змінена під час передачі.
* Дозволяє перевірити автентичність сервера, захищаючи від атак типу "man–in–the–middle" (MITM).

PKI також використовується для забезпечення електронних підписів, які гарантують автентичність, цілісність та непідробність документів. Електронні підписи є юридично обов'язковими в багатьох країнах і використовуються для підтвердження автентичності документів в електронному вигляді. Користувач підписує документ своїм приватним ключем. Підписаний документ можна перевірити за допомогою публічного ключа користувача, який міститься в його сертифікаті. Підпис забезпечує цілісність документа (будь–які зміни в документі після підпису роблять підпис недійсним).

Приклад:

* S/MIME (Secure/Multipurpose Internet Mail Extensions) використовується для підпису та шифрування електронної пошти.
* PAdES (PDF Advanced Electronic Signatures) застосовується для підпису PDF–документів.

Virtual Private Network (VPN) використовує PKI для забезпечення безпечного доступу до віддалених ресурсів в Інтернеті або корпоративних мережах. Використання цифрових сертифікатів дозволяє забезпечити автентифікацію користувачів та серверів, а також захищене з'єднання через незахищену мережу. Користувач або пристрій отримує сертифікат від корпоративного CA для підключення до VPN. Сертифікат використовується для автентифікації пристрою або користувача під час підключення до серверу. Для забезпечення конфіденційності даних використовується шифрування з використанням ключів, пов'язаних із сертифікатами. Приклад застосування –

SSL/TLS VPN використовує цифрові сертифікати для захищеного доступу до мережі через SSL/TLS протокол.

Взаємна автентифікація (mutual authentication) за допомогою PKI широко використовується в інтернеті речей (IoT) та мікросервісах, де пристрої та сервіси повинні взаємно перевіряти свою автентичність перед обміном даними. Кожен пристрій або мікросервіс має цифровий сертифікат, який підтверджує його ідентичність. Сервер і клієнт обмінюються сертифікатами, перевіряючи, чи є сертифікат дійсним. Якщо автентифікація успішна, з'єднання встановлюється, і починається обмін даними. Приклад:

* MQTT з SSL/TLS для захищеного зв'язку між IoT пристроями.
* mTLS (mutual TLS) для автентифікації клієнтів і серверів в мікросервісах.

PKI активно використовується для підпису програмного забезпечення та коду, щоб гарантувати, що програмний продукт або файл не були змінені або підроблені. Це забезпечує користувачам впевненість у безпеці отриманих програм. Розробник або організація підписує програму своїм приватним ключем. Користувач може перевірити підпис за допомогою публічного ключа, що вказаний у сертифікаті розробника. Підпис гарантує, що програма не була змінена після її підписання. Приклад:

* Microsoft Code Signing для підпису драйверів та додатків.
* Android або iOS Signatures для мобільних додатків.

PKI має величезне значення в багатьох сферах, від захисту персональних даних і забезпечення безпеки в Інтернеті до верифікації програмного забезпечення та автентифікації в системах IoT. Його застосування дозволяє реалізувати масштабовані та безпечні механізми для забезпечення довіри в сучасних технологічних середовищах.

1. **Ланцюжок сертифікатів.**

Ланцюжок сертифікатів (англ. Certificate Chain) – це структура, що описує взаємозв'язок між сертифікатами в ієрархії сертифікаційних центрів (CA). У класичній системі PKI (Public Key Infrastructure) ланцюжок сертифікатів починається з кінцевого сертифіката, який належить користувачеві чи організації, і веде до кореневого сертифіката (Root CA), який підписує всі сертифікати у системі.

Кожен сертифікат у ланцюжку має бути підписаний сертифікатом, що є вище в ієрархії. Таким чином, довіра до кінцевого сертифіката ґрунтується на довірі до кореневого сертифіката, який зазвичай знаходиться в списку довірених сертифікатів на стороні клієнта (наприклад, в браузері або на сервері). Основні елементи ланцюжка сертифікатів:

* ***Кінцевий сертифікат (End–Entity Certificate).*** Це сертифікат користувача, сервісу або пристрою. Він підписаний проміжним сертифікатом (Intermediate CA). Кінцевий сертифікат містить публічний ключ і додаткову інформацію (поля Subject, Validity, SAN, тощо).
* ***Проміжний сертифікат (Intermediate Certificate).*** Проміжний сертифікат підписаний кореневим сертифікатом або іншим проміжним сертифікатом. Він дозволяє зменшити ризик компрометації кореневого сертифіката. Проміжні сертифікати дозволяють організувати ієрархію сертифікації.
* ***Кореневий сертифікат (Root Certificate).*** Це сертифікат, що стоїть на вершині ієрархії. Кореневий сертифікат самопідписаний і є основним елементом довіри для всієї системи PKI. Він повинен зберігатися в дуже захищеному середовищі, оскільки компрометація кореневого сертифіката може підривати безпеку всієї інфраструктури.

Кожен сертифікат у ланцюжку має бути перевірений за допомогою сертифіката, що знаходиться вище в ієрархії. Приклад такої перевірки:

* ***Перевірка підпису сертифіката.*** Кожен сертифікат у ланцюжку має бути підписаний сертифікатом, що знаходиться вищим (як правило, це проміжний сертифікат або Root CA). Перевірка підпису є критичним кроком, оскільки вона гарантує, що сертифікат не був змінений і є дійсним.
* ***Перевірка терміну дії сертифікатів.*** Кожен сертифікат має вказаний період дії (з датою початку та кінця). Якщо термін дії сертифіката завершився, він не є дійсним і не може бути використаний для перевірки підпису.
* ***Перевірка ланцюга довіри.*** Для кожного сертифіката в ланцюжку перевіряється, чи його підпис дійсно підписаний сертифікатом вищого рівня. Ланцюжок довіри завершується перевіркою Root CA, який повинен бути в списку довірених сертифікатів на стороні клієнта (наприклад, в браузері чи сервері).

Важливість ланцюжка сертифікатів:

* ***Довіра та безпека.*** Ланцюжок сертифікатів дозволяє забезпечити, що підпис і публічний ключ кінцевого сертифіката дійсно належать його власнику, а не зловмиснику. Якщо користувач або сервер може перевірити ланцюжок довіри, це гарантує, що з'єднання є захищеним і що відсутні загрози, пов'язані з підробленими сертифікатами.
* ***Анонімність та приватність.*** Ланцюжок сертифікатів дозволяє захистити приватні дані (наприклад, під час веб–сесій) від перехоплення або маніпуляцій з боку третьої сторони.
* ***Застосування у реальних сценаріях.*** Ланцюжок сертифікатів використовується в таких протоколах, як SSL/TLS для безпечної комунікації через Інтернет. Інші типи автентифікації, такі як VPN, S/MIME (для електронної пошти), і навіть мобільні та IoT пристрої використовують PKI для захисту з’єднань і даних.

Для перевірки ланцюжка сертифікатів використовуються спеціалізовані бібліотеки та утиліти, такі як:

* OpenSSL: для перевірки та аналізу сертифікатів.
* Python (Cryptography library): для програмного управління ланцюжками сертифікатів.

При перевірці ланцюжка важливо враховувати, що всі сертифікати повинні бути дійсними, а також мати правильні підписи та терміни дії.

Ланцюжок сертифікатів – це критичний елемент інфраструктури PKI, який гарантує безпеку та достовірність цифрових сертифікатів. Він дозволяє підтримувати довіру до кінцевого сертифіката, перевіряючи підпис кожного сертифіката в ієрархії від кінцевого до кореневого. Без правильно налаштованого ланцюжка довіри неможливо забезпечити належну безпеку для багатьох сучасних протоколів та технологій.

1. **Практичні інструменти для роботи з сертифікатами.**

Python надає кілька потужних бібліотек для роботи з криптографією та сертифікатами. Дві основні бібліотеки для роботи з цифровими сертифікатами – *cryptography* та *pyOpenSSL*.

***Cryptography***. Бібліотека *cryptography* є однією з найпопулярніших у Python для роботи з криптографічними операціями, включаючи генерацію ключів, підписування та перевірку сертифікатів. Вона дозволяє працювати з такими форматами як PEM та DER.

***pyOpenSSL.*** Інша популярна бібліотека, *pyOpenSSL*, є Python–обгорткою для бібліотеки *OpenSSL* і дозволяє створювати та працювати з сертифікатами, а також реалізувати криптографічні операції. Вона забезпечує більшу сумісність з *OpenSSL*, і дозволяє виконувати більше операцій, зокрема генерацію ключів, створення запитів на сертифікати (CSR), підписування сертифікатів та перевірку ланцюга сертифікатів.

***OpenSSL*** є стандартним інструментом для роботи з криптографією, сертифікатами та ключами. Це потужна утиліта командного рядка, яка надає можливість генерувати, підписувати, перевіряти сертифікати, а також виконувати інші операції з криптографічними даними. OpenSSL використовується як у серверних налаштуваннях (наприклад, для налаштування HTTPS), так і в криптографічних додатках.

***Keytool –*** це інструмент, що входить до складу Java Development Kit (JDK). Він використовується для управління сертифікатами в Java KeyStore (JKS). З його допомогою можна додавати сертифікати в хранилище ключів, генерувати пари ключів та перевіряти сертифікати.

***cfssl*** – це інструмент для створення сертифікатів і управління ними, розроблений компанією CloudFlare. Він є зручним для розгортання власних сертифікаційних центрів і створення сертифікатів у автоматизованих системах. Включає в себе набір утиліт для управління сертифікатами, ключами та запитами на сертифікати.

Визначення розширень сертифікатів (X.509 Extensions):

* ***Basic Constraints.*** Вказує, чи є сертифікат сертифікаційним центром (CA=true) або кінцевим сертифікатом (CA=false).
* ***Key Usage.*** Визначає, для яких цілей призначений ключ (наприклад, підписування, шифрування).
* ***Extended Key Usage.*** Дозволяє більш конкретно визначити цілі використання ключа (наприклад, для серверної автентифікації або підпису коду).
* ***Subject Alternative Name (SAN)***. Додає альтернативні ідентифікатори (доменні імена, IP–адреси) для сертифіката.

Сертифікати використовуються в різних сферах, таких як захист веб–трафіку через HTTPS, електронний підпис документів, автентифікація користувачів та пристроїв у мережі. Для кожного застосування потрібні різні інструменти і підходи до налаштування сертифікатів.

1. **Приклад виконання**

Реалізуємо приклад з використанням бібліотеки *cryptography*, у якому всі об’єкти сертифікатів і ключів зберігаються у форматі PEM, а файли зберігаються у відповідних папках (root\_ca/, intermediate\_ca/, user/).

**Загальні імпорти.**

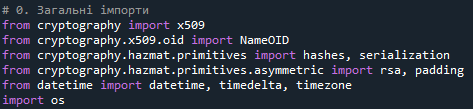
****

Рисунок 7.2. Приклад загальних імпортів.

* 1. **Створення Root CA.**

Root CA – це самопідписаний сертифікат, який довіряє тільки собі і яким підписуються всі інші сертифікати (безпосередньо або опосередковано) (рис. 7.3).

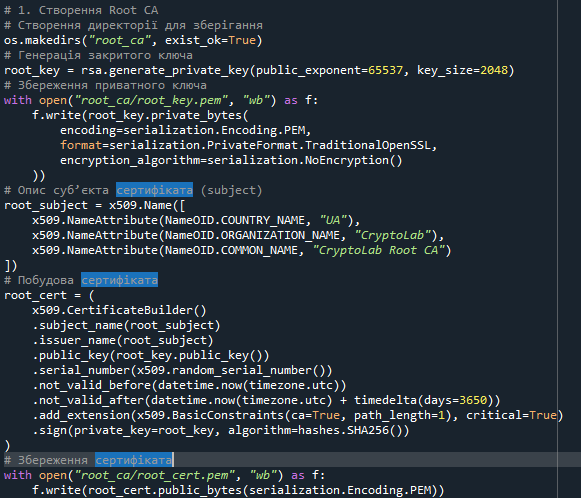


Рисунок 7.3. Приклад створення Root CA.

**Створення директорії для зберігання.**Створюємо каталог ***root\_ca***, якщо він ще не існує. Там будуть зберігатися: приватний ключ ***root\_key.pem,*** сертифікат ***root\_cert.pem.***

**Генерація закритого ключа.**Генеруємо пару ключів RSA:

* public\_exponent=65537 – стандартне значення, яке забезпечує безпеку і ефективність.
* key\_size=2048 – мінімально безпечний розмір на сьогодні (можна збільшити до 3072 або 4096 для більш високих вимог).

**Збереження приватного ключа:**

* serialization.Encoding.PEM – зберігаємо у зручному текстовому форматі.
* TraditionalOpenSSL – класичний формат PEM для OpenSSL.
* NoEncryption() – ключ зберігається без шифрування (лише для навчальних цілей!). У реальних умовах потрібно використовувати пароль.

**Опис суб’єкта сертифіката (subject).** Це «ім’я» сертифіката – хто є власником: C (Country), O (Organization), CN (Common Name –загальне ім’я, яке використовується для ідентифікації).

**Побудова сертифіката:**

* ***.subject\_name(...):*** встановлює дані про власника сертифіката.
* ***.issuer\_name(...):*** хто видав сертифікат. У Root CA це те саме, що й subject\_name (самопідписаний).
* ***.public\_key(...):*** публічний ключ, який буде міститися в сертифікаті.
* ***.serial\_number(...):*** унікальний номер сертифіката (генерується випадково).
* ***.not\_valid\_before(...):*** початок дії сертифіката (встановлюється як зараз).
* ***.not\_valid\_after(...):*** дата завершення дії сертифіката (10 років).
* ***.add\_extension(...):*** додаємо розширення BasicConstraints: ***ca=True*** – це сертифікат центру сертифікації (CA), ***path\_length=1*** – максимальна глибина підлеглих CA (можна підписати ще одну проміжну CA, але не більше).
* ***.sign(...):*** підписуємо сертифікат закритим ключем Root CA. Використовується хеш–алгоритм SHA–256.

**Збереження сертифіката.** Сертифікат кодується у формат PEM і записується у файл. Цей файл містить публічну інформацію: ім’я, публічний ключ, термін дії, цифровий підпис тощо.

В результаті отримуємо:

| **Файл** | **Вміст** |
| --- | --- |
| root\_ca/root\_key.pem | приватний ключ Root CA (тільки для CA) |
| root\_ca/root\_cert.pem | самопідписаний сертифікат Root CA |

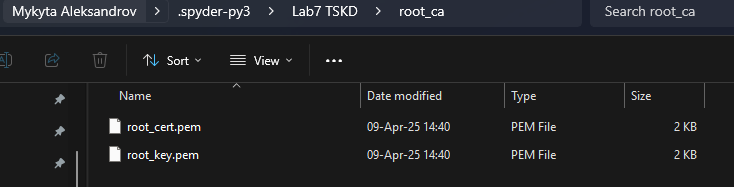


Рисунок 7.4. Вміст root\_ca.

* 1. **Створення проміжного сертифікаційного центру (Intermediate CA).**

Intermediate CA виконує роль посередника між Root CA і кінцевими користувачами. Це дозволяє захистити Root CA (він не використовується для повсякденного підписування сертифікатів) (рис. 7.5).

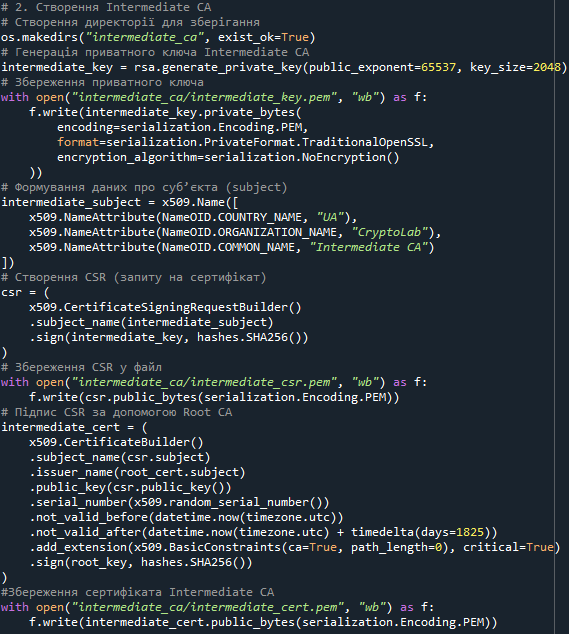


Рисунок 7.5. Приклад створення Intermediate CA.

**Створення директорії для зберігання.** Каталог ***intermediate\_ca*** міститиме: приватний ключ, запит на сертифікат (CSR), сам сертифікат Intermediate CA, підписаний Root CA

**Генерація приватного ключа Intermediate CA.** Такий самий підхід, як і для Root CA. Генеруємо пару ключів для проміжного сертифіката (ці ключі не мають стосунку до Root CA – це інша пара ключів).

**Збереження приватного ключа.**Зберігаємо закритий ключ у форматі PEM. У реальних умовах – варто шифрувати.

**Формування даних про суб’єкта (subject).**Ім’я Intermediate CA (організація, країна, загальне ім’я). Це буде відображатися в сертифікаті та в ланцюжку довіри.

**Створення CSR (запиту на сертифікат).** CSR (Certificate Signing Request) – запит, який надсилається до Root CA для підпису. У ньому міститься: ім’я суб’єкта, публічний ключ, цифровий підпис, створений приватним ключем Intermediate CA.

**Збереження CSR у файл.** CSR можна було б надіслати Root CA як файл або в електронній формі. У нашому випадку підпис буде одразу в коді.

**Підпис CSR за** допомогою **Root CA:**

* ***subject\_name(csr.subject)*** – використовуємо дані з CSR.
* ***issuer\_name(root\_cert.subject)*** – видавцем є Root CA.
* ***public\_key(...)*** – отримуємо публічний ключ з CSR.
* not\_valid\_after(...) – строк дії на 5 років.
* BasicConstraints(ca=True, path\_length=0): ca=True – це сертифікат для CA, path\_length=0 – він не може видавати інші сертифікати CA, лише для кінцевих користувачів.
* .sign(...) – сертифікат підписується приватним ключем Root CA.

**Збереження сертифіката Intermediate CA**. Зберігаємо готовий сертифікат, підписаний Root CA, у файл.

В результаті отримуємо:

| **Файл** | **Вміст** |
| --- | --- |
| intermediate\_key.pem | приватний ключ Intermediate CA |
| intermediate\_csr.pem | запит на сертифікат (CSR) |
| intermediate\_cert.pem | сертифікат Intermediate CA, підписаний Root CA |

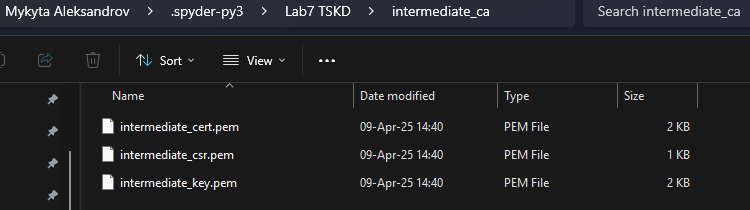


Рисунок 7.6. Вміст intermediate\_ca.

На цьому етапі Root CA вже не потрібен у щоденній роботі. Він може зберігатися офлайн. Всі користувацькі сертифікати тепер будуть підписуватися Intermediate CA.

* 1. **Створення сертифіката користувача (end–entity certificate).**

Це вже не CA, а звичайний користувач або сервер. Сертифікат використовується для: шифрування (наприклад, TLS), цифрового підпису (наприклад, email), автентифікації (наприклад, VPN) (рис. 7.7).

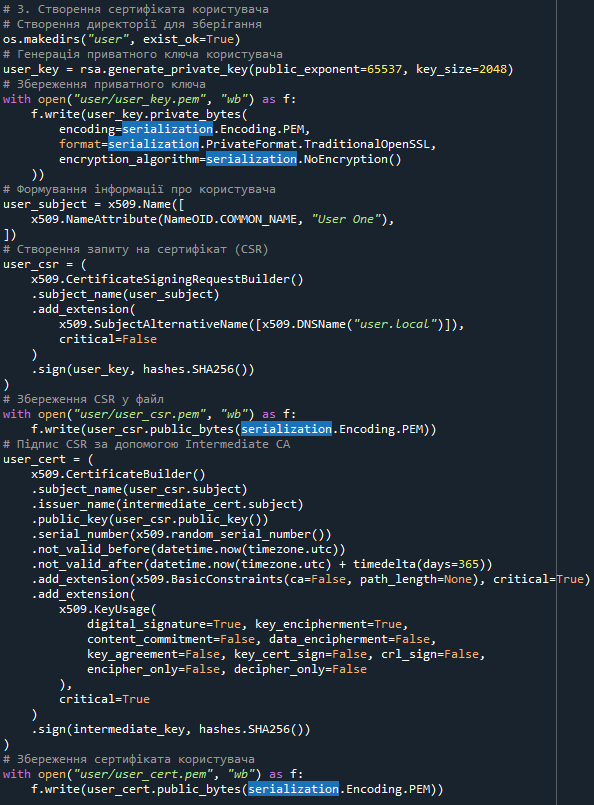


Рисунок 7.7. Приклад створення end–entity certificate.

**Створення директорії для зберігання**. Каталог user зберігатиме: приватний ключ користувача, CSR (запит), сертифікат, підписаний Intermediate CA.

**Генерація приватного ключа користувача.** Генерується пара ключів для користувача. Це вже не CA, тому він не може видавати інші сертифікати.

**Збереження приватного ключа.** Приватний ключ користувача зберігається у відкритому вигляді (тільки для демонстрації, у реальному житті – з паролем!).

**Формування інформації про користувача*.*** Країна, організація, загальне ім’я (ім’я користувача або ім’я пристрою). Це інформація, що відображатиметься в сертифікаті.

**Створення запиту на сертифікат (CSR).** CSR створюється з підписом приватного ключа користувача, із зазначенням його імені та публічного ключа.

**Збереження CSR у файл.** CSR можна передати до Intermediate CA для видачі сертифіката.

**Підпис CSR за допомогою Intermediate CA:**

* ***.issuer\_name(...) –*** сертифікат видає Intermediate CA.
* ***.BasicConstraints(ca=False) –*** вказує, що це не CA, а користувач.
* ***.sign(...) –*** сертифікат підписується закритим ключем Intermediate CA.

**Збереження сертифіката користувача.**Збереження підписаного сертифіката користувача у файл.

**У результаті маємо:**

| **Файл** | **Вміст** |
| --- | --- |
| user\_key.pem | приватний ключ користувача |
| user\_csr.pem | запит на сертифікат |
| user\_cert.pem | сертифікат користувача, підписаний Intermediate CA |

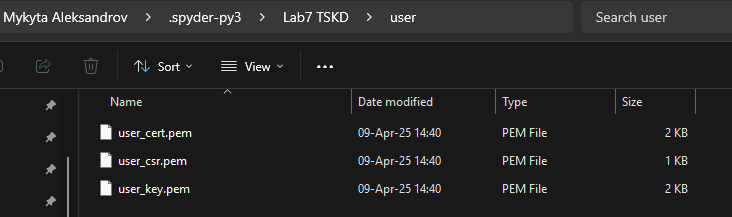


Рисунок 7.8. Вміст user.

Довірчий ланцюг сертифікатів –

***user\_cert.pem*** 🡨 ***intermediate\_cert.pem*** 🡨 ***root\_cert.pem***

цей ланцюг можна перевірити у браузерах, серверах, OpenSSL, тощо. Він засвідчує, що користувач довірений, бо його сертифікат підписаний довіреним CA.

* 1. **Перевірка ланцюжка сертифікатів (вручну).**

На цьому кроці слід переконатися, що сертифікати утворюють правильний ланцюжок довіри. Сертифікат користувача (*end–entity*) підписаний Intermediate CA, Intermediate CA підписаний Root CA (рис. 7.9).

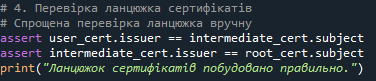


Рисунок 7.9. Приклад перевірки ланцюжка сертифікатів.

***assert user\_cert.issuer == intermediate\_cert.subject.*** Це перевіряє чи дійсно сертифікат користувача був виданий Intermediate CA.

***user\_cert.issuer*** – об’єкт типу x509.Name, який вказує, хто підписав сертифікат користувача.

***intermediate\_cert.subject*** – об’єкт x509.Name, який вказує, хто є власником сертифіката Intermediate CA.

Якщо вони збігаються – сертифікат користувача дійсно був виданий Intermediate CA.

***assert intermediate\_cert.issuer == root\_cert.subject.*** Це перевіряє чи був сертифікат Intermediate CA підписаний Root CA.

***intermediate\_cert.issuer*** – видавець Intermediate CA.

***root\_cert.subject*** – власник Root CA.

Якщо збігаються → сертифікат Intermediate CA підписав саме Root CA.

**Що не перевіряється в цьому коді:**

* Чи дійсно підпис сертифіката валідний (тобто криптографічна перевірка).
* Чи не закінчився строк дії сертифікатів.
* Чи не відкликані сертифікати (через CRL або OCSP).
* Наявність розширень (KeyUsage, BasicConstraints, ExtendedKeyUsage тощо).

Для повної перевірки потрібен окремий парсер/бібліотека або зовнішній інструмент, наприклад, OpenSSL.

Цей ручний тест надасть базову впевненість, що структура сертифікатів логічно правильна. Це корисно на рівні навчання та валідації створення PKI.

* 1. **Підпис та перевірка повідомлення.**

На цьому кроці потрібно переконатися у цілісності та автентичності даних. Підпис дозволяє перевірити, що дані дійсно були надіслані власником ключа і не були змінені. Повідомлення хешується (SHA–256). Перевіряється, чи цей хеш збігається з тим, що був зашифрований у підписі (з використанням публічного ключа). Якщо так – підпис дійсний, повідомлення не було змінено, і воно точно було створене власником приватного ключа (рис. 7.10).

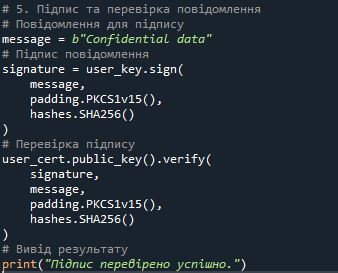
****

Рисунок 7.10. Приклад перевірки ланцюжка сертифікатів.

**Повідомлення для підпису*.*** Ми визначаємо байтовий рядок (тобто b"") – це важливо, бо функція підпису працює тільки з байтами, не з рядками. Повідомлення може бути будь–яке: текст, хеш, JSON, бінарний файл.

**Підпис повідомлення:**

* ***user\_key.sign(...)*** – метод створення підпису приватним ключем користувача.
* ***message*** – дані, які підписуються.
* ***padding.PKCS1v15()*** – обгортка для безпечного використання RSA.
* ***hashes.SHA256()*** – використовується хеш–функція SHA–256. Дані хешуються перед підписом (підпис створюється не з усіх даних, а з їхнього хешу).

Результат – цифровий підпис. Байтова послідовність, прив’язана до цього конкретного повідомлення та цього приватного ключа.

**Перевірка підпису:**

***user\_cert.public\_key()*** – отримуємо публічний ключ із сертифіката користувача.

***.verify(...)*** – метод перевірки цифрового підпису.

***signature*** – підпис, який перевіряємо.

***message*** – оригінальне повідомлення.

***padding.PKCS1v15()*** та ***hashes.SHA256()*** – мають збігатися з тими, що використовувались при підписі.

Якщо підпис валідний – метод ***verify(...)*** не повертає нічого (це нормально). Якщо підпис неправильний – буде викликано виняток ***InvalidSignature.***

**Вивід результату.** Це повідомлення виводиться тільки якщо перевірка підпису не згенерувала виняток – тобто все добре.

**Запуск застосунку:**

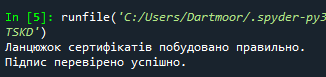
****

Рисунок 7.11. Приклад коду для запуску застосунку.

**Питання для самоконтролю**

1. Що таке Public Key Infrastructure (PKI) і які її основні компоненти?
2. Як PKI забезпечує безпеку передачі даних?
3. Яку роль відіграють сертифікаційні центри (CA) в системі PKI?
4. В чому полягає різниця між root CA та intermediate CA?
5. Які основні поля містить сертифікат X.509?
6. Що таке Subject Alternative Name (SAN) і для чого його використовують в сертифікатах?
7. Як можна визначити, чи є сертифікат самопідписаним (self-signed)?
8. Як довго діє сертифікат і що впливає на його термін дії?
9. Які операції можуть виконувати сертифікаційні центри (CA)?
10. Як забезпечується безпека ключів сертифікаційного центру?
11. Які кроки необхідно виконати для отримання сертифіката від сертифікаційного центру?
12. Які кроки входять в процес створення та підписання сертифіката?
13. Як здійснюється перевірка сертифіката за допомогою ланцюга довіри?
14. Що таке CRL та OCSP і як вони використовуються для анулювання сертифікатів?
15. Що таке Key Usage в сертифікаті і які можливі значення цього поля?
16. Яка роль поля Extended Key Usage в сертифікаті?
17. Для чого використовується поле Basic Constraints в сертифікатах?
18. Які основні команди в OpenSSL використовуються для генерації та перевірки сертифікатів?
19. Як в Python можна створити запит на сертифікат (CSR) за допомогою бібліотеки cryptography?
20. Що таке pyOpenSSL і чим ця бібліотека відрізняється від cryptography?
21. Як створити самопідписаний сертифікат за допомогою OpenSSL?
22. Як перевірити, чи відповідає сертифікат домену або IP-адресі, за допомогою SAN?
23. Які помилки можуть виникнути при відсутності або невірному вказанні поля SAN в сертифікаті?
24. Як OpenSSL дозволяє перевірити ланцюжок сертифікатів і що таке Chain of Trust?
25. Як використовується цифровий сертифікат для захисту HTTPS трафіку?
26. Яка роль цифрових сертифікатів у електронному підписі документів?
27. Які інші технології (окрім HTTPS) можуть використовувати сертифікати для забезпечення безпеки?

**Література: [1-8]**

# 8. ЛАБОРАТОРНЕ ЗАНЯТТЯ №8. Написання простого блокчейну

**Мета –** набуття практичних навичок побудови простої моделі блокчейну. Закріплення знань про хеш–функції, цілісність даних, структуру блоку, цифрові підписи та перевірку ланцюга. Ознайомлення з базовими принципами, на яких працюють криптовалютні мережі та інші децентралізовані системи.

**Завдання -** реалізувати простий блокчейн, що підтримує наступну функціональність:

1. ***Створення ланцюга блоків.*** Кожен блок повинен мати унікальний індекс, мітку часу, дані про транзакції, хеш попереднього блоку, власний хеш та інші параметри згідно варіанту. Перший блок (генезис–блок) – створюється при ініціалізації системи.
2. ***Збереження та перевірка цілісності.*** Хеш блоку повинен обчислюватись з урахуванням усіх основних полів. Ланцюг повинен мати метод для перевірки валідності (перевірка зв'язків між блоками, перевірка хешів, а також додаткові перевірки за варіантом).
3. ***Додавання нових блоків.*** Має бути реалізований метод додавання нового блоку з автоматичним розрахунком хешу та зв’язком із попереднім. Транзакції (у вигляді списку словників) додаються як "навантаження" у блок.

**Вимоги до завдання:**

* Реалізація на мові Python.
* Класова реалізація (принаймні два класи – Block та Blockchain).
* Дані транзакцій – у форматі словників (наприклад, {"sender": "Alice", "receiver": "Bob", "amount": 100}).
* Для хешування використовується SHA–256 (бібліотека hashlib).
* Усі обчислення, пов’язані з хешами, підписами, тощо, мають бути відтворюваними (тобто детермінованими).

**Вимоги до звіту:**

1. Мета, завдання та варіант.
2. Опис заданого за варіантом додатку.
3. Опис процесу розробки додатку. (Кожен етап повинен бути описаний)
4. Опис процесу тестування роботи розробленого додатку. Тестування повинно налічувати як позитивні так і негативні сценарії. (Кожен скріншот повинен містити опис та підпис.)
5. Приклади вмісту вихідних та результуючих даних.
6. Висновки.

Варіант завдання для виконання роботи узгоджується з викладачем.

Варіанти завдань до лабораторного заняття №8.

| № | Варіант |
| --- | --- |
| 1 | Використовуйте SHA–256. Складність PoW – п’ять нулів (00000). |
| 2 | Перевірка цілісності включає не лише ланцюг, а й контроль сум транзакцій (сума вхідних = вихідним). |
| 3 | Використовуйте SHA3–256 для хешування (через hashlib.new ("sha3\_256")). Складність – три нулі. |
| 4 | Реалізуйте підрахунок вхідного та вихідного балансу кожного користувача після кожного блоку. |
| 5 | Використовуйте BLAKE2s (через hashlib.blake2s) для хешування. Складність – три нулі. |
| 6 | Реалізуйте систему нагороди за блок. Кожен новий блок додає 50 одиниць "винагороди" майнеру ("miner": "user1"). |
| 7 | Для кожного блоку зберігайте хеш двох попередніх блоків, а не одного (previous\_hash\_1, previous\_hash\_2). |
| 8 | Використовуйте SHA–256 для хешування. Складність PoW – хеш починається з трьох нулів (000). |
| 9 | У кожній транзакції додається випадковий ідентифікатор (txid), який хешується разом із іншими даними блоку. |
| 10 | Використовуйте SHA3–256. Складність PoW – чотири нулі (0000). |

## Теоретичні відомості до виконання лабораторного заняття № 8.

1. **Визначення блокчейну.**

***Блокчейн*** (від англ. blockchain – ланцюг блоків) – це розподілена база даних або система реєстрації транзакцій, в якій дані зберігаються у вигляді послідовних блоків. Кожен блок містить набір транзакцій, а також хеш попереднього блоку. Це створює ланцюг блоків, у якому кожен новий блок залежить від попереднього.

Ключова особливість блокчейну полягає в тому, що він забезпечує незмінність (immutability). Змінити вже додані в блокчейн дані дуже складно, що робить цю технологію надзвичайно корисною для зберігання важливої інформації, де безпека та прозорість є критичними, наприклад, у фінансових транзакціях, юридичних документах, та ін.

**Основні компоненти блокчейну.**

Блокчейн складається з кількох основних компонентів:

1. ***Блок.*** *Дані блоку* – кожен блок містить набір транзакцій або інших даних. Транзакції можуть бути передачею цифрових активів, повідомлень, контрактів тощо. *Хеш блоку* – унікальний ідентифікатор блоку, який генерується за допомогою криптографічної хеш–функції (SHA–256 у нашому випадку). *Хеш попереднього блоку* – цей хеш забезпечує зв'язок між блоками і є критичним елементом для забезпечення незмінності ланцюга. *Nonce* ***–*** число, яке використовується при майнінгу блоку. Воно є частиною процесу пошуку правильного хешу, який задовольняє умови складності. *Мітка часу* вказує час створення блоку.
2. ***Ланцюг блоків.*** Кожен блок у ланцюгу містить хеш попереднього блоку. Це забезпечує зв'язок між блоками і створює послідовність, що робить блокчейн незмінним. Зміна одного блоку змусить змінити всі наступні блоки.
3. ***Майнінг.*** Процес додавання нового блоку в ланцюг. Майнери вирішують складні криптографічні задачі, щоб знайти значення nonce, яке дає правильний хеш блоку. Цей процес забезпечує консенсус і визначає, який блок буде доданий наступним. Майнінг часто включає в себе отримання нагороди (наприклад, у вигляді криптовалюти) для майнера, який успішно підтвердить блок.
4. ***Транзакції****.* Основним призначенням блокчейну є обробка транзакцій – це записи про передачу даних або цифрових активів між учасниками системи. Транзакції зберігаються в блоках, і кожен блок може містити кілька транзакцій. Транзакції можуть включати в себе такі дані, як відправник, отримувач і сума (наприклад, в разі криптовалют – кількість монет).

**Принципи роботи блокчейну:**

* ***Децентралізація.*** Блокчейн не має єдиного контролюючого органу або адміністратора. Всі учасники мережі є рівноправними, і кожен має доступ до копії блокчейну. Це робить систему стійкою до цензури та атаки, оскільки немає єдиного уразливого вузла.
* ***Незмінність (immutability).*** Після того, як блок було додано до ланцюга, змінити його практично неможливо. Це забезпечується тим, що кожен блок містить хеш попереднього, і зміна даних в одному блоці змусить змінити всі наступні блоки, що є технічно складним та затратним процесом.
* ***Прозорість і безпека.*** Всі транзакції, що відбуваються в межах блокчейну, доступні для перевірки. Це гарантує прозорість і дозволяє учасникам перевіряти, чи не були змінені дані. Використання криптографії, зокрема хешування, гарантує високий рівень безпеки.
* ***Консенсус.*** Блокчейн використовує різні механізми для досягнення консенсусу між учасниками мережі, щоб визначити, який блок є наступним у ланцюзі. В більшості блокчейнів використовується Proof of Work (PoW) або Proof of Stake (PoS) як механізм консенсусу. У нашому прикладі ми використовуємо PoW, де майнери знаходять правильний nonce, щоб додати блок до ланцюга.

Блокчейн – це потужна технологія, що змінила підходи до обробки транзакцій, зберігання даних та забезпечення безпеки в розподілених системах. Його незмінність, децентралізація та прозорість роблять його важливим інструментом у сучасному цифровому світі. Застосування блокчейну вже набуло популярності не лише у фінансових системах, але й у інших сферах, таких як логістика, медицина, державне управління та багато інших.

1. **Структура блокчейну.**

Блокчейн – це ланцюг блоків, де кожен блок містить набір транзакцій, і має криптографічний хеш попереднього блоку. Така структура забезпечує незмінність та безпеку даних. Важливими характеристиками блокчейну є:

**Складові блоку:**

* ***Індекс (index).*** Це унікальний номер блоку в ланцюзі. Перший блок зазвичай має індекс 0 (генезис–блок), наступний — індекс 1, і так далі. Індекс допомагає визначити порядок блоків у ланцюзі та сприяє їхній правильній організації.
* ***Часова мітка (timestamp).*** Це точний час створення блоку. Мітка часу дозволяє відстежити, коли саме блок був доданий до блокчейну. Це важливий елемент для визначення порядку блоків та транзакцій.
* ***Транзакції (transactions)****.* Це основний вміст блоку. Кожен блок може містити кілька транзакцій. Транзакція описує передачу чогось (наприклад, криптовалюти) від одного користувача до іншого. Формат транзакції може бути різним, але зазвичай включає відправника, отримувача та суму.
* ***Попередній хеш (previous\_hash).*** Це хеш попереднього блоку в ланцюзі. Кожен блок містить хеш попереднього блоку, що забезпечує зв’язок між блоками і створює ланцюг. Якщо хеш попереднього блоку зміниться, то зміни торкнуться всіх наступних блоків, що робить блокчейн незмінним і захищеним від маніпуляцій.
* ***Nonce (Number used once).*** Це число, яке майнер шукає, щоб знайти правильний хеш для блоку, що відповідає умові складності. В процесі майнінгу майнер спробує різні значення nonce, поки не отримає хеш, який задовольняє умови, наприклад, починається з певної кількості нулів. Nonce використовується для підтвердження того, що блок був створений чесно, і його не можна підробити.
* ***Хеш блоку (hash).*** Хеш – це криптографічний підсумок вмісту блоку. Він унікальний для кожного блоку, і навіть невелика зміна в блоці змінить його хеш. Хеш блоку є результатом застосування хеш–функції (SHA–256). Він використовується для перевірки цілісності блоку. Хеш блоку гарантує, що його вміст не був змінений після його створення. Якщо хоча б один символ вмісту зміниться, хеш також зміниться.

**Блокчейн** представляє собою ланцюг блоків. Перший блок (генезис–блок) створюється без попереднього блоку, тому він містить спеціальний хеш "0" у полі previous\_hash. Кожен наступний блок містить хеш попереднього блоку у полі previous\_hash, що дозволяє зв’язати їх у ланцюг. Це створює залежність між блоками, і якщо хоча б один блок буде змінений, зміняться й усі наступні блоки, оскільки їх хеші містять хеш попередніх блоків.

**Майнінг** – це процес знаходження правильного значення nonce, яке дозволяє хешу блоку відповідати умові складності. Складність визначає, скільки нулів на початку повинно бути у хеші блоку. Чим більше нулів, тим складніше знайти правильний хеш. Складність – відома як рівень "PoW" (Proof of Work) – доведення роботи. Наприклад, блок може мати умову: хеш повинен починатися з 4 нулів, тобто хеш повинен виглядати, наприклад, як 0000xxxx.... Майнінг включає проби значень nonce до того, як буде знайдений правильний хеш. Це вимагає значних обчислювальних ресурсів і часу, що робить систему безпечною від атак.

Оскільки кожен блок містить хеш попереднього блоку, кожен блок у ланцюзі залежить від попереднього. Це забезпечує цілісність блокчейну, тому що навіть одна зміна вмісту блоку змінить його хеш, і вся структура блоків буде пошкоджена.

**Цілісність ланцюга** – якщо один блок буде змінений (наприклад, зміниться транзакція або майнінг), це призведе до зміни хешу блоку. Всі наступні блоки в ланцюгу також зміняться, і перевірка цілісності за допомогою хешів не буде пройдена.

1. **Основні компоненти блокчейну в задачі.**

**Транзакції –** це основний елемент, який забезпечує рух вартості або інформації між учасниками блокчейн–мережі. У нашому випадку кожна транзакція містить відомості про відправника, одержувача та суму або інші дані, які передаються. У реальному житті транзакції можуть бути набагато складнішими (наприклад, з підписами або додатковими умовами), але для цієї задачі ми зберігаємо основну інформацію.

***Формат транзакції.*** Транзакція в нашій реалізації виглядає як словник з трьох основних полів:

* sender – ім'я або адреса відправника (наприклад, "alice").
* receiver – ім'я або адреса одержувача (наприклад, "bob").
* amount – кількість валюти (або іншої інформації), яку відправляє відправник (наприклад, 10).

Транзакції додаються до блоку як список, і блок має атрибут transactions, що містить усі транзакції цього блоку. Це дозволяє здійснювати передавання даних через блоки, що зберігаються в ланцюгу.

**Майнінг** – це процес додавання нового блоку до блокчейну шляхом пошуку правильного nonce. Nonce – це число, яке при додаванні до вмісту блоку дає такий хеш, який відповідає умовам складності мережі. Майнінг блоку полягає в знаходженні значення nonce, при якому хеш блоку починається з певної кількості нулів. Це дозволяє контролювати складність роботи мережі. Наприклад, якщо складність блоку визначена як 4 нулі на початку хешу, майнер має перебирати значення nonce, доки хеш блоку не задовольнить цю умову.

***Алгоритм майнінгу:***

* Майнінг блоку є частиною процесу створення блоку.
* Кожен блок містить:
  + *index*: порядковий номер блоку.
  + *timestamp*: час створення блоку.
  + *transactions*: список транзакцій.
  + *previous*\_*hash*: хеш попереднього блоку (це зв'язує блоки в ланцюгу).
  + *nonce*: число, яке майнер шукає, щоб забезпечити відповідність складності.
  + *hash*: кінцевий хеш блоку, що буде записано у ланцюг.
* Складність майнінгу визначається параметром DIFFICULTY. У нашій задачі це визначає, скільки нулів має бути на початку хешу для того, щоб блок вважався дійсним.
* Процес майнінгу. Для кожного блоку майнер починає з певного значення nonce і обчислює хеш. Якщо хеш не задовольняє умови складності, майнер змінює nonce і пробує знову. Це важливий етап, який потребує значних обчислювальних ресурсів, що робить мережу безпечною від атак.

**Merkle Tree (Дерево Меркля).** Merkle Tree – це структура даних, яка використовується для ефективного зберігання транзакцій у блоці. Вона дозволяє перевіряти, чи належить конкретна транзакція до блоку, без необхідності перевіряти всі транзакції.

Як працює дерево Меркля:

* Кожна транзакція хешується.
* Потім хеші з’єднуються попарно, і хеші пари обчислюються разом.
* Це повторюється, доки не буде отримано єдиний хеш, який є root hash дерева.

У випадку, що розглянуто у прикладі нижче, не використовуємо складне дерево Меркля, а лише зберігаємо всі транзакції у вигляді списку. Але для реального блокчейну цей підхід дозволяє швидко перевіряти наявність транзакції без перегляду всіх даних.

**Ланцюг блоків (Blockchain).** Ланцюг блоків – це зв’язок усіх блоків у блокчейні. Кожен новий блок містить хеш попереднього блоку, що забезпечує незмінність ланцюга: якщо спробувати змінити будь–який блок, хеш зміниться, і всі наступні блоки також стануть недійсними.

Як ланцюг зберігає блоки:

* Додавання блоку. Кожен новий блок додається до ланцюга через метод *add\_block()* в класі *Blockchain*. Кожен блок в ланцюгу містить хеш попереднього блоку в полі *previous\_hash*.
* Перевірка валідності ланцюга. Використовується метод *is\_valid(),* який перевіряє: правильність хешів попередніх блоків, чи відповідає хеш кожного блоку умові складності (відповідність nonce).
* Генезис–блок. Перший блок у блокчейні, який зазвичай не має попереднього блоку (наприклад, previous\_hash = 0). Цей блок використовується для ініціалізації ланцюга.

**Алгоритм консенсусу (Proof of Work).** Алгоритм Proof of Work (PoW) – це механізм досягнення консенсусу в мережі, де майнери змагаються за право додавати нові блоки в блокчейн, вирішуючи складні обчислювальні задачі. У нашій задачі це виражено у вигляді пошуку правильного nonce для блоку. Proof of Work гарантує, що майнери витрачають ресурси на обчислення та доводять, що вони виконали певну роботу для створення блоку. Тільки після того, як блок знайдений і задовольняє умови складності, він додається до блокчейну.

1. **Алгоритми та методи.**

**SHA–256 (Криптографічна хеш–функція).**

SHA–256 (Secure Hash Algorithm 256–bit) — це одна з хеш–функцій сімейства SHA, що використовує алгоритм для перетворення вхідних даних (повідомлень, файлів, тощо) у вихідний хеш довжиною 256 біт (32 байти). Хеш–функція забезпечує:

* ***Односторонність*** – з отриманого хешу неможливо отримати початкові дані.
* ***Унікальність*** – навіть невелика зміна у вхідних даних призводить до значної зміни в хеші.
* ***Швидкість*** – хешування даних виконується дуже швидко, що робить SHA–256 ідеальним для блокчейнів.

В блокчейні SHA–256 використовується для:

* Обчислення хешу кожного блоку (включаючи транзакції, попередній хеш та інші дані).
* Перевірки цілісності блокчейну – якщо хеш блоку змінюється, змінюється й хеш усіх наступних блоків.
* Майнінгу – майнер шукає таке значення nonce, яке дасть хеш, що задовольняє умови складності (наприклад, починається з кількох нулів).

**Proof of Work (PoW).**

Proof of Work (PoW) — це алгоритм консенсусу, який використовує обчислювальну роботу для забезпечення безпеки та підтвердження блоків у блокчейні.Майнінг в PoW вимагає, щоб учасники мережі (майнери) виконували обчислення, що ґрунтуються на складності (наприклад, знаходження хешу блоку з певною кількістю нулів на початку).

***Принцип роботи PoW.*** Майнер вибирає значення nonce (число, яке додається до даних блоку). Майнер перераховує різні значення nonce, обчислюючи хеш кожного варіанту. Якщо хеш блоку задовольняє умови складності (наприклад, починається з 4 нулів), майнер підтверджує блок. Блок додається до ланцюга, і майнер отримує нагороду (нові монети).

**Merkle Tree (Дерево Меркля).**

Merkle Tree – це структура даних, яка дозволяє ефективно та безпечно перевіряти цілісність великої кількості даних. Вона складається з хешів, які об’єднуються в дерево. Кожен лист дерева — це хеш транзакцій. Батьківські вузли — хеші пари дочірніх елементів. У блокчейні дерево Меркля використовується для зберігання транзакцій у блоці. За допомогою дерева Меркля можна швидко перевірити, чи є транзакція в блоці, перевіривши тільки хеши з дерева, замість того щоб порівнювати всі транзакції.

1. **Перевірка валідності блокчейну.**

Перевірка дозволяє впевнитися, що:

* жоден блок у ланцюгу не змінено після його додавання;
* хеші відповідають вмісту блоків;
* дотримано правил зв’язування блоків (кожен блок повинен знати хеш попереднього);
* виконано умови доказу виконаної роботи (Proof of Work);
* порядок транзакцій логічний, і не порушено загальні правила протоколу.

**Алгоритми перевірки валідності блокчейну:**

* ***Перевірка хешів.*** Щоб переконатися в цілісності блокчейну, треба перевірити, що хеш кожного блоку правильний. Якщо хоча б один блок змінений, його хеш також зміниться, що вплине на всі наступні блоки.
* ***Перевірка складності.*** Перевірка складності полягає у визначенні того, чи задовольняє хеш блоку умови, такі як кількість нулів на початку. Це гарантує, що майнінг був складним і що блок не може бути підроблений.

1. **Приклад виконання.**

Приклад містить реалізацію *Block* та *Blockchain*, підтримує *Proof of Work* зі змінною складністю, використовує SHA–256 (але легко замінити на SHA3, BLAKE2s, тощо), має можливість додавати нагороду за блок, підготовлений до додавання *Merkle Root*, подвійного хешування тощо.

**Структура проєкту:**

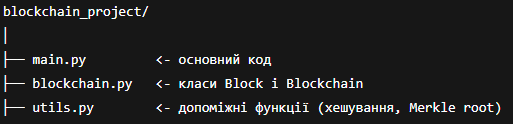


Рисунок 8.1. Приклад структури проєкту.

* 1. **Файл – blockchain.py.**

Файл blockchain.py – реалізує повністю працездатний блокчейн із Proof of Work. Він гнучкий, бо в ньому легко змінюються алгоритми, складність, структура блоку. Тому він є основою для роботи системи.

**Імпорт бібліотек:**

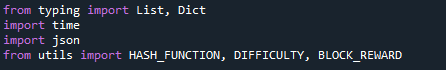
****

Рисунок 8.2. Приклад імпорту бібліотек.

**Опис:**

* ***typing*** – типи ***List, Dict*** – для читабельності та автодоповнення.
* ***time*** – потрібен для збереження мітки часу у блоці.
* ***json*** – використовуємо для серіалізації блоку перед хешуванням.
* ***utils*** – всі параметри (***DIFFICULTY, BLOCK\_REWARD***) та хеш–функція винесені в окремий файл, щоб легко підлаштовувати їх під варіанти.

**Клас Block.**

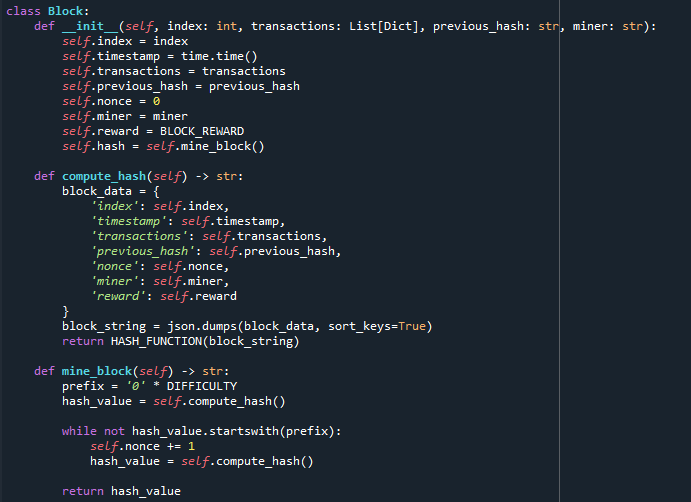
****

Рисунок 8.3. Приклад коду класу Block.

**Опис:**

* ***\_\_init\_\_*** – конструктор:
* ***іndex*** – номер блоку в ланцюзі.
* ***transactions*** – список транзакцій (формат: словники з sender, receiver, amount).
* ***previous\_hash*** – хеш попереднього блоку.
* ***miner*** – ім’я майнера, який "знайшов" блок.
* ***timestamp*** – час створення блоку (Unix–час).
* ***nonce –*** число, яке підбирається для задоволення умови складності.
* **reward** – нагорода за майнінг (додається в transactions вручну).
* ***hash*** – зберігається результат функції mine\_block() – обчислений хеш блоку.
* ***compute\_hash*** – обчислення хешу блоку, Формується словник з усіма полями блоку, які впливають на хеш. **nonce** та **miner** обов’язково включаються – інакше хеш не змінюватиметься при спробі майнінгу. Словник серіалізується у строку з ***sort\_keys=True*** для однакового порядку ключів. Рядок подається на вхід хеш–функції (SHA–256 за замовчуванням).
* **mine\_block** – реалізація Proof of Work. Задається префікс із DIFFICULTY нулів. **nonce** поступово збільшується, поки хеш не відповідатиме умові (починається з кількох нулів). Після вдалого підбору nonce повертається валідний хеш.

**Клас Blockchain.**

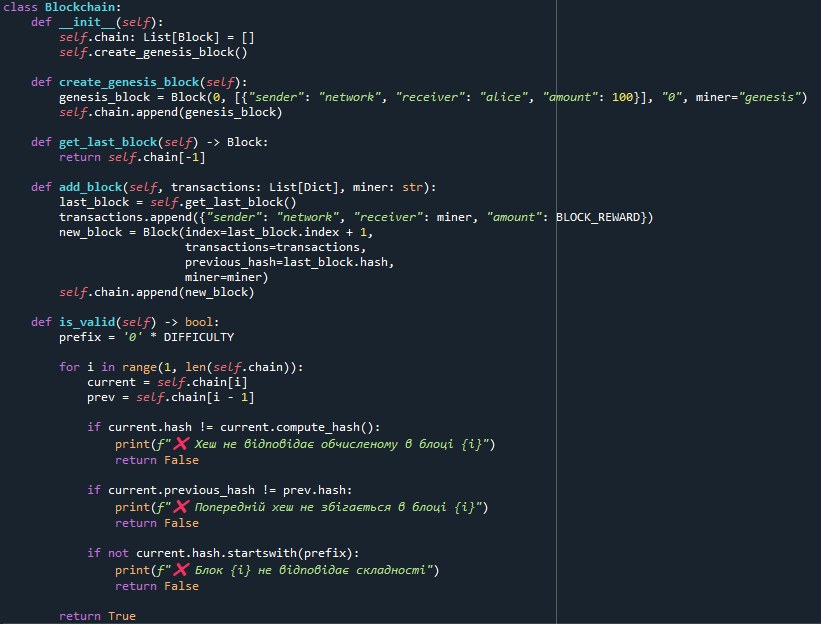
****

Рисунок 8.4. Приклад коду класу **Blockchain**.

**Опис:**

* ***Конструктор.*** Ініціалізується порожній список chain. Викликається метод створення генезис–блоку.
* ***create\_genesis\_block*** – перший блок ланцюга. Має ***index = 0***. Включає одну транзакцію (наприклад, видача початкового балансу). ***previous\_hash = "0"*** – бо це перший блок. ***miner = "genesis"*** – умовно.
* ***get\_last\_block*** – повертає останній блок із ланцюга.
* ***add\_block*** – додає новий блок. Додається транзакція з нагородою для майнера. Створюється новий Block (index – наступний, transactions – з урахуванням нагороди, previous\_hash – хеш попереднього блоку, miner – передається як аргумент), блок додається до chain.
* ***is\_valid***  – перевіряє цілісність ланцюга. Перевірка починається з другого блоку (генезис не перевіряється, бо задається вручну). Проводиться перевірка, що збережений хеш збігається з актуальним обчисленим. Проводиться перевірка зв’язності ланцюга – чи дійсно поточний блок посилається на хеш попереднього. Проводиться перевірка, що хеш задовольняє умову складності. Якщо всі перевірки пройдені – ланцюг валідний.
  1. **Файл – utils.py**

Файл utils.py – централізує всі змінні, які визначають параметри блокчейну. Дає змогу легко створювати варіанти завдань без зміни основного коду.Містить хеш–функцію та Merkle Root як приклад складніших структур.

**Імпорт бібліотек:**

****

Рисунок 8.5. Приклад імпорту бібліотек.

**Опис:**

* ***hashlib*** – стандартний модуль Python для роботи з хеш–функціями (SHA–256, SHA–3, BLAKE2s тощо).
* ***typing.List –*** використовується для анотації типів списків, наприклад у функції merkle\_root.

**Параметри блокчейну:**

****

Рисунок 8.6. Приклад параметрів блокчейну.

**Опис:**

* ***DIFFICULTY –*** визначає складність майнінгу. Чим більше число – тим складніше знайти підходящий nonce, бо хеш має починатись з більшої кількості нулів.
* ***BLOCK\_REWARD –*** скільки "монет" отримає майнер за доданий блок. Використовується при формуванні додаткової транзакції типу {"sender": "network", "receiver": "miner", "amount": BLOCK\_REWARD}.

**Базова хеш–функція:**



Рисунок 8.7. Приклад коду базової хеш–функції.

**Опис.**

Приймає на вхід рядок s. Кодує його у байти – s.encode(). Хешує з допомогою SHA–256: hashlib.sha256(...). Повертає шістнадцяткове представлення: .hexdigest(). Виносимо хеш–функцію в окрему змінну, щоб легко змінювати її в інших частинах коду. В blockchain.py не викликаємо напряму hashlib.sha256(...), а використовуємо HASH\_FUNCTION(...), що дає змогу підміняти її без змін основного коду.

**Функція merkle\_root.**

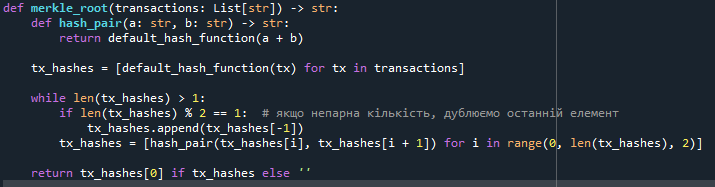
****

Рисунок 8.8. Приклад коду функції merkle\_root.

**Опис.**

Merkle Root – це одне хеш–значення, яке представляє собою узагальнення всіх транзакцій у блоці. Воно часто використовується в блокчейнах для: зменшення обсягу даних для перевірки, спрощення доказів включення (Merkle proof), додаткової перевірки цілісності. Допоміжна функція конкатенує два хеші і знову хешує результат. Після цього хешуються всі транзакції та отримується список хешів. Щоб побудувати повноцінне бінарне дерево, кількість хешів має бути парною. Якщо ні – дублюється останній. В циклі проходимо по парах і будуємо новий рівень дерева, а коли залишився лише один елемент – це і є Merkle Root.

* 1. **Файл – main.py**

Файл main.py показує повний цикл роботи з блокчейном. Дає змогу протестувати логіку без додаткових інтерфейсів. Є вихідною точкою для розширення: взаємодії з користувачем, тестування, підключення візуалізації або графічного інтерфейсу.

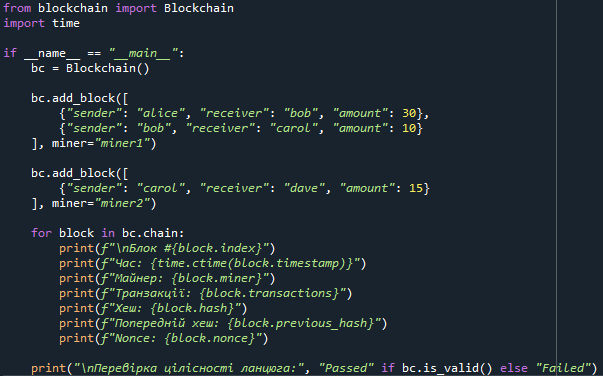
****

Рисунок 8.9. Приклад коду файлу main.py.

**Опис:**

* Імпортується клас *Blockchain* з *blockchain*.*py*.
* Створюється екземпляр класу *Blockchain*. Автоматично генерується генезис–блок.
* Додається перший користувацький блок. У блоці дві транзакції. Майнить блок miner1, і отримає нагороду (BLOCK\_REWARD), яка додається автоматично.
* Додається другий блок – знову кілька транзакцій, інший майнер.
* Ітеруємося по всьому ланцюгу *blockchain.chain* і виводимо детальну інформацію про кожен блок. Включає: індекс, часову мітку, ім’я майнера, підібраний nonce, транзакції, попередній хеш, фінальний хеш блоку.
* Викликається метод *is\_valid(),* який перевіряє: правильність збережених хешів, відповідність previous\_hash , виконання умов складності (DIFFICULTY).
* Виводиться відповідне повідомлення (рис. 8.10).

**Приклад виводу результатів у консолі.**

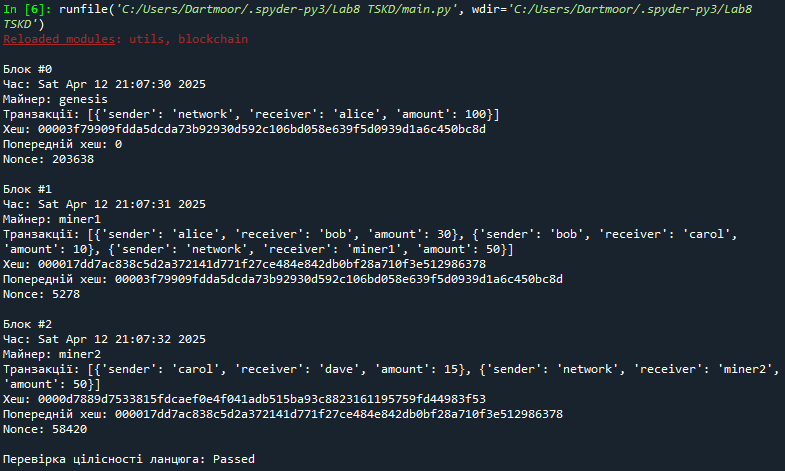
****

Рисунок 8.10. Приклад роботи програми.

**Питання для самоконтролю**

1. Що таке блокчейн? Які його основні властивості?
2. Які елементи входять до складу одного блоку в блокчейні?
3. Яку роль відіграє поле previous\_hash в структурі блоку?
4. Чим хеш блоку відрізняється від Merkle Root?
5. Що відбудеться, якщо змінити дані в одному з блоків у вже згенерованому ланцюгу?
6. Яка роль хеш–функції SHA–256 у блокчейні?
7. Чому SHA–256 вважається надійною для використання в криптографії?
8. Що таке "одностороння функція" і як це пов’язано з хешуванням?
9. Як забезпечується унікальність хешу для кожного блоку?
10. Як зміниться хеш, якщо змінити лише один символ у транзакції?
11. Що таке алгоритм Proof of Work і яка його мета?
12. Яку роль відіграє параметр nonce у процесі майнінгу?
13. Як складність (difficulty) впливає на швидкість майнінгу?
14. Чому обчислення nonce потребує багато часу, а перевірка - майже миттєва?
15. Що буде, якщо встановити занадто високу або низьку складність?
16. Для чого використовується Merkle Tree у блокчейні?
17. Як побудувати Merkle Root зі списку транзакцій?
18. Чому при непарній кількості транзакцій дублюють останню перед побудовою дерева?
19. Як за допомогою Merkle Tree можна перевірити належність транзакції до блоку?
20. Яка різниця між хешем блоку та Merkle Root?
21. Як перевірити, що блокчейн не був змінений?
22. Що перевіряє метод is\_valid() у нашій реалізації?
23. Як порушення ланцюжка хешів впливає на валідність блокчейну?
24. Чи можна видалити або змінити блок у центрі ланцюга без впливу на наступні?
25. Чому перевірка валідності є критично важливою в публічних блокчейнах?
26. Як створити новий блок і додати його до блокчейну?
27. Як виглядає типовий формат транзакції в нашому прикладі?
28. Які дані використовуються для обчислення хешу блоку?
29. Як змінити складність майнінгу в нашій реалізації?
30. Які обмеження має запропонована реалізація в порівнянні з реальними блокчейн–системами?

**Література: [1-8]**

# СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Кац Джонатан Introduction to Modern Cryptography / Джонатан Кац,Єгуда Лінделл. — Taylor & Francis, Chapman & Hall/CRC Cryptography and Network Security Series, 2020. — 648 p.
2. Щур Н. О. Основи криптології: навч. посіб. /Н. О. Щур, О. А.Покотило. – Житомир: Державний університет «Житомирська політехніка», 2021. 120 с.
3. Сазонець О. Інформаційні системи та технології в управлінні зовнішньоекономічною діяльністю: навч. посіб. – Київ :Центр навчальної літератури, 2017. – 256 с.
4. Онацький О. В. Криптографічні системи : навч. посіб. з дисциплін "Криптографія та криптоаналіз" для освітньо-професійної підготовки бакалаврів в галузі знань 12 "Інформаційні технології" за спеціальністю 125 "Кібербезпека" / О. В. Онацький, Л. Г. Йона. – Одеса : Міжнародний гуманітарний університет, 2023. – 156 с.
5. Теорія та практика розслідування злочинів в умовах протидії: навч.-метод. посіб. / Л. І. Аркуша, Л. М. Гуртієва, М. О. Д’ячкова, та ін. – Одеса: Фенікс, 2021.
6. Вишняков В. М. Захист інформації в комп'ютерних системах : навч. посіб. / В. М. Вишняков. - Київ : КНУБА, 2022. - 119 с.
7. Безпека інформаційних систем : підручник / Ю. А. Тарнавський; КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2023. – 161 с.
8. Методологія і технології захисту інформації: навч. посіб. / А. Н. Аль-Амморі, Н. М. Наумова, П. В. Дяченко, та ін. – Київ: НТУ, 2020. – 147 с.

# ІНФОРМАЦІЙНІ РЕСУРСИ

1. Види шифрування інформації [Електронний ресурс]. – Режим доступу: https:// [ua5.org/protect/395-vidi-shifruvannya-informaciyi.html](http://ua5.org/protect/395-vidi-shifruvannya-informaciyi.html). - Назва з екрана.
2. Криптографічні методи захисту інформації. Контроль цілісності програмних та інформаційних ресурсів [Електронний ресурс]. – Режим доступу : [https://sites.google.com/view/dcptoinformat/інформатика](https://sites.google.com/view/dcptoinformat/%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0). - Назва з екрана.