НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра Інформаційної Безпеки

Лабораторна робота №4

з дисципліни

«МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ»

Виконали:

Студени 6 курсу ФТІ

групи ФБ-41мн

Бондаренко О.Ю., Кригін Д.О.

Перевірила:

асистент

Байденко П.В.

Київ 2025

Лабораторна робота №4
Дослідження особливостей реалізації існуючих програмних систем, які використовують криптографічні механізми захисту інформації.

Мета: Отримання практичних навичок побудови гібридних криптосистем.

Завдання: Розробити реалізацію асиметричної криптосистеми у відповідності до стандартних вимог Crypto API або стандартів PKCS та дослідити стійкість стандартних криптопровайдерів до атак, що використовують недосконаліть механізмів захисту операційної системи. Підгрупа 2B. Бібліотека PyCrypto під Linux платформу. Стандарт ECDSA.

Хід Роботи

Використані стандарти, характеристики та технічна реалізація

Стандарт	Опис	Технічна Реалізація
FIPS 186-4/FIPS 186-5 (DSS)	Стандарт цифрового підпису визначає набір алгоритмів цифрового підпису (включаючи ECDSA).	Використання DSS.new(key, “fips-186-3”) безпосередньо викликає реалізацію бібліотеки, що відповідає процедурам цього стандарту для генерації та перевірки підпису.
NIST P-256 (secp256r1)	Спеціальна еліптична крива, що забезпечує 128-бітний рівень безпеки.	ECC.generate(curve=“P-256”) вибирає параметри кривої, визначені NIST.
FIPS 180-4 (SHA-2)	Стандарт Secure Hash Standard вимагає використання затверджених хеш-функцій для	SHA256.new(message) використовує хеш-функцію, затверджену FIPS, для генерації дайджесту

	цифрових підписів.	повідомлення перед підписанням.
PKCS#8	Визначає синтаксис для зберігання незашифрованого або зашифрованого приватного ключа.	private_key.export_key(... , use_pkcs8=True, passphrase=...) гарантує відповідність згенерованого файлу вимогам.
PKCS#5	Визначає криптографію на основі пароля (наприклад, PBKDF2, Scrypt) для шифрування приватного ключа.	Аргумент пароля, що використовується під час export_key, запускає процес шифрування, який регулюється механізмами PKCS#5.

Генерація ключ-пари

Почнемо реалізацію асиметричної криптосистеми із виконання вимог PKCS і NIST щодо створення ключа і його безпечного зберігання. На етапі генерації ключів вибір кривої є найважливішим рішенням.

Використаємо NIST P-256 криву (NIST P-256/ECC Standard), щоб згенерувати 256-bit ECDSA ключ. Ключі, згенеровані за допомогою цієї кривої матимуть стійкість, еквівалентну 3072-bit RSA ключам.

```
def generate_and_store_key():
    print("\n--- 1. Key Generation and Export (NIST P-256, PKCS#8, SHA-512/AES-256) ---")

    private_key_file = input("Enter the filename for the encrypted private key (e.g., ecdsa_private.pem): ")
    passphrase_str = getpass.getpass("Enter a secure passphrase for key encryption: ")

    passphrase_bytes = passphrase_str.encode('utf-8')

    PROTECTION_ALGORITHM_STR = "PBKDF2WithHMAC-SHA512AndAES256-CBC"

    # NIST P-256/ECC Standard
```

```
    private_key_object = ECC.generate(curve='P-256')
    public_key_object = private_key_object.public_key() # Derive public key
    print("Generated new ECDSA key pair (NIST P-256).")

    # PKCS#8 and PKCS#5 Standards (Private Key Export)
    try:
        encrypted_pem_key_str = private_key_object.export_key(
            format='PEM',
            use_pkcs8=True,
            passphrase=passphrase_bytes,
            protection=PROTECTION_ALGORITHM_STR
        )

        with open(private_key_file, "w") as f:
            f.write(encrypted_pem_key_str)

        print(f"Private key securely stored in {private_key_file} (Encrypted PKCS#8
PEM).")

        public_key_file = private_key_file.replace(".pem", ".pub") if ".pem" in
private_key_file else private_key_file + ".pub"
        public_pem_str = public_key_object.export_key(format='PEM')

        with open(public_key_file, "w") as f:
            f.write(public_pem_str)

        print(f"Public key exported to {public_key_file} for distribution.")
        return public_key_object

    except Exception as e:
        print(f"Error during key export: {e}")
        return None
```

Безпечне зберігання

Після отримання пари ключів, необхідно забезпечити захищене зберігання приватного ключа. Стандартною практикою є використання паролю для захисту, що зменшить ризики пов’язані з недосконалістю безпеки ОС, викрадення файлу ключа не дозволить його використати. Це забезпечує виконання стандартів PKCS#8 і PKCS#5

Експортований ключ форматується за допомогою PKCS#8 для забезпечення сумісності. Він шифрується (PKCS#5) за допомогою AES-256, отриманого з парольної фрази.

```
encrypted_pem_key_str = private_key_object.export_key(
        format='PEM',
        use_pkcs8=True,
        passphrase=passphrase_bytes,
        protection=PROTECTION_ALGORITHM_STR
    )
```

```
(.venv) user@debian:~/Labs/MRKM/Lab4$ python3 lab4.py

============================================
     ECDSA Cryptosystem Protocol Menu
============================================
Internally Stored Signatures: 0
1. Generate and Securely Store New Private Key (Exports Private/Public Keys)
2. Sign New Message (Allows Export of Message/Signature)
3. Verify Signature (Requires Public Key, Message, and Signature Files)
4. Exit
--------------------------------------------
Select an option (1-4): 1

--- 1. Key Generation and Export (NIST P-256, PKCS#8, SHA-512/AES-256) ---
Enter the filename for the encrypted private key (e.g., ecdsa_private.pem): lab4_private.pem
Enter a secure passphrase for key encryption:
Generated new ECDSA key pair (NIST P-256).
Private key securely stored in lab4_private.pem (Encrypted PKCS#8 PEM).
Public key exported to lab4_private.pub for distribution.
```

```
user@debian:~/Labs/MRKM/Lab4$ file lab4_*
lab4_private.pem: ASCII text
lab4_private.pub: ASCII text
user@debian:~/Labs/MRKM/Lab4$ cat lab4_private.pub
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEM7Cwb0YuN1iWURoXkNVcF/QPM1n4
hHuRXhLgYI7ODNanLMTNO5H5qdQyGhl0Ghmjn4HEI6sitsk5Y/vPXVgdfQ==
-----END PUBLIC KEY-----user@debian:~/Labs/MRKM/Lab4$ cat lab4_private.pem
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIHqMFUGCSqGSIb3DQEFDTBIMCcGCSqGSIb3DQEFDDAaBAhubYMLMeLCbwICA+gw
CgYIKoZIhvcNAgswHQYJYIZIAWUDBAEqBBB1sEDdJr7jbvRUEYbOqa6qBIGQ/bSz
itQWuqeGG3zxKnIlUTKcYqYWnfX5hE/tBZwnAVX/zsXzEuxVMI3umYsKPU2WRdN3
iNyjBZbIm2Hj6s3pYzhNMr03jOP4DF6t2/uqNcEHuSXkdbHw7cgO/sqDFkuORQUo
t55L0C1q+zVi33uQwblQMPfPNRDABhyocWlapHu8sg2gkt+jH0hETeqmIfzn
-----END ENCRYPTED PRIVATE KEY-----user@debian:~/Labs/MRKM/Lab4$
```

Використання ключа для підпису

Для дотримання вимог FIPS 186-4 ми маємо забезпечити безпечне використання ключа на етапі його використання для підпису.

Використовуючи зашифрований приватний ключ (стандарти PKCS#8 PKCS#5)

та пасфразу користувача, завантажуємо та розшифровуємо ключ у пам’яті.

Для створення підпису та дотримання вимог FIPS 180-4 (SHA-2 стандарт) FIPS 186-3/FIPS 186-4 (DSS Standard Mode) необхідно використовувати затверджену геш-функцію і алгоритм підпису. Режим fips-186-3 гарантує, що процедури генерації підписів відповідають вимогам стандарту щодо безпеки щодо використання випадкових нонсів(nonce).

```
def sign_message(public_keys):
    print("\n--- 2. Key Loading and Signing (FIPS 186-4, SHA256) ---")

    private_key_file = input("Enter the filename of the encrypted private key to use:
")
    decryption_passphrase_str = getpass.getpass(f"Enter passphrase to decrypt
{private_key_file}: ")

    decryption_passphrase_bytes = decryption_passphrase_str.encode('utf-8')

    # ISO 8859-1 (Latin-1) message encoding
    message_content = input("Enter the content of the message to be signed (will
use ISO 8859-1 encoding): ")
    message = message_content.encode('iso-8859-1')

    loaded_key = None
    # PKCS#8/PKCS#5 Standard (Import)
    try:
        with open(private_key_file, "r") as f:
            encrypted_key_data_str = f.read()

        encrypted_key_data_bytes = encrypted_key_data_str.encode('ascii')

        loaded_key = ECC.import_key(encrypted_key_data_bytes,
passphrase=decryption_passphrase_bytes)
        public_key_object = loaded_key.public_key()
        print("Encrypted private key loaded and decrypted successfully.")
    except FileNotFoundError:
        print(f"Error: Private key file {private_key_file} not found.")
        return
    except ValueError as e:
        print(f"Error: Failed to decrypt key. Incorrect passphrase or unsupported
protection. Details: {e}")
        return
    except Exception as e:
        print(f"Error loading key: {e}")
        return

    # FIPS 180-4 (SHA-2 Standard)
    message_hash = SHA256.new(message)

    # FIPS 186-4/FIPS 186-3 (DSS Standard Mode)
    try:
```

```
        signer = DSS.new(loaded_key, 'fips-186-3')
        digital_signature = signer.sign(message_hash)

        print("Message successfully signed.")
        print(f"Signature (hex): {digital_signature.hex()[:60]}...")

        public_keys[private_key_file] = (public_key_object, message,
digital_signature)

        export_choice = input("Do you want to export the message and signature to
files? (y/n): ").strip().lower()
        if export_choice == 'y':
            base_filename = input("Enter a base filename for message/signature (e.g.,
doc_a): ")

            message_file = f"{base_filename}.msg"
            with open(message_file, "wb") as f:
                f.write(message)
            print(f"Message saved to {message_file}.")

            signature_file = f"{base_filename}.sig"
            with open(signature_file, "wb") as f:
                f.write(digital_signature)
            print(f"Signature saved to {signature_file}.")

    except Exception as e:
        print(f"Error during signing: {e}")
```

```
Select an option (1-4): 2

--- 2. Key Loading and Signing (FIPS 186-4, SHA256) ---
Enter the filename of the encrypted private key to use: lab4_private.pem
Enter passphrase to decrypt lab4_private.pem:
Enter the content of the message to be signed (will use ISO 8859-1 encoding): Kitty should only like kitties.
Encrypted private key loaded and decrypted successfully.
Message successfully signed.
Signature (hex): 8841ccbdeda861c2916e79092ae171f76364ed969cbf22760590b67ecc55...
Do you want to export the message and signature to files? (y/n): y
Enter a base filename for message/signature (e.g., doc_a): kitty_notes
Message saved to kitty_notes.msg.
Signature saved to kitty_notes.sig.
```

```
user@debian:~/Labs/MRKM/Lab4$ cat kitty_notes.msg | xxd
00000000: 4b69 7474 7920 7368 6f75 6c64 206f 6e6c  Kitty should onl
00000010: 7920 6c69 6b65 206b 6974 7469 6573 2e    y like kitties.
user@debian:~/Labs/MRKM/Lab4$ cat kitty_notes.sig | xxd
00000000: 8841 ccbd eda8 61c2 916e 7909 2ae1 71f7  .A....a..ny.*.q.
00000010: 6364 ed96 9cbf 2276 0590 b67e cc55 2b2b  cd...."v...~.U++
00000020: c89d 0121 0789 d575 422e 3159 9da2 e9fd  ...!...uB.1Y....
00000030: 1506 4893 55af 3cba 736a 40c1 6828 0662  ..H.U.<.sj@.h(.b
```

Перевірка (цілісність та автентичність)

Верифікація необхідна для підтвердження, що сигнатура для повідомлення була створена відповідним публічним ключем.

```
public_key_file = input("Enter filename of the public key (.pub file): ")
            message_file = input("Enter filename of the signed message (.msg file):
")
            signature_file = input("Enter filename of the signature (.sig file): ")

            with open(public_key_file, "r") as f:
                public_key_data_str = f.read()
            public_key = ECC.import_key(public_key_data_str)
            print("Public Key loaded.")

            with open(message_file, "rb") as f:
                message_to_check = f.read()

            with open(signature_file, "rb") as f:
                digital_signature = f.read()

            # FIPS 186-4 Verification Procedure
            verifier_hash = SHA256.new(message_to_check)

            verifier = DSS.new(public_key, 'fips-186-3')

            try:
                verifier.verify(verifier_hash, digital_signature)
                print("\nRESULT: Signature is **VALID**. Message authenticity and
integrity confirmed.")
            except ValueError:
                print("\nRESULT: Signature is **INVALID**. The message, signature,
or key is corrupted/incorrect.")
```

```
Select an option (1-4): 3

--- 3. Signature Verification ---
Enter filename of the public key (.pub file): lab4_private.pub
Enter filename of the signed message (.msg file): kitty_notes.msg
Enter filename of the signature (.sig file): kitty_notes.sig
Public Key loaded.

RESULT: Signature is **VALID**. Message authenticity and integrity confirmed.
```

Модифікуємо повідомлення

```
user@debian:~/Labs/MRKM/Lab4$ xxd kitty_notes.msg
00000000: 4b69 7474 7920 7368 6f75 6c64 206f 6e6c  Kitty should onl
00000010: 7920 6c69 6b65 2070 7570 7069 6573 2e0a  y like puppies..
```

```
Select an option (1-4): 3

--- 3. Signature Verification ---
Enter filename of the public key (.pub file): lab4_private.pub
Enter filename of the signed message (.msg file): kitty_notes.msg
Enter filename of the signature (.sig file): kitty_notes.sig
Public Key loaded.

RESULT: Signature is **INVALID**. The message, signature, or key is corrupted/incorrect.
```

Захищеність від атак, що використовують недосконалості ОС

Захищеність стандартних криптопровайдерів від атак, що використовують недосконаліть механізмів захисту операційної системи дуже складна тема, оскільки залучає багато Side-Channel векторів, що включають використання фізичного обладнання для отримання інформації. Прикладом таких атак є Electromagnetic Analysis, що передбачає моніторинг EM випромінення від CPU та GPU, останнє, на приклад, дозволяє частково відновити зображення на екрані пристрою. Більшою ефективністю володіють Timing атаки, особливо при недосконалому механізмі випадковості у криптоалгоритмі або використанні системного часу для генерації випадкових значень без дотримання стандартів.

Більш актуальними є Cold Boot та Memory Exploits атаки, оскільки після використання секретного ключа він, ще деякий час, знаходиться у оперативній пам’яті пристрою, із якої його можна вилучити. Memory Exploits, які обходять права доступу до пам’яті процесів, можуть використовуватися як для викрадення так і для підміни ключа, що становить найбільшу загрозу для криптоалгоритмів та є постійною небезпекою зі сторони ОС.

Висновки

Реалізовано впровадження протоколу криптосистеми ECDSA, що забезпечило функціональну та відповідну стандартам систему цифрового підпису. Було використано необзідні криптографічні стандарти, включаючи NIST P-256 для генерації кривих, FIPS 180-4 (SHA-256) для хешування та FIPS 186-4 для режимів підписання та перевірки стандарту цифрового підпису (DSS). Безпечне управління ключами було забезпечено за допомогою інкапсуляції PKCS#8 з

використанням PBKDF2 з шифруванням AES-256. Кінцева система забезпечує практичний, реалістичний робочий процес, дозволяючи експортувати похідний відкритий ключ, підписане повідомлення та підпис в окремі файли, тим самим успішно перевіряючи критичні компоненти автентичності та цілісності повідомлення на етапі перевірки на основі файлів.