Collegium Da Vinci w Poznaniu

Studia stacjonarne I stopnia – Informatyka

Sprawozdanie

z wykonania projektu

Przedmiot: **Bezpieczeństwo systemów informatycznych**

Grupa: **3**

Rok akad.: **2023/2024**

Prowadzący: **mgr inż. Jacek Mielnik**

Projekt:

Temat: **Secure Communicator**

Student: **Kacper Kureń (28128), Antoni Marcinek (28926)**

Data wykonania: **15.06.2024**

1. Cel ćwiczenia

Celem ćwiczenia było stworzenie aplikacji do bezpiecznej komunikacji za pomocą protokołu MQTT oraz implementacja szyfrowania wiadomości przy użyciu kluczy RSA. Aplikacja miała umożliwiać użytkownikom rejestrowanie się, logowanie, a następnie wymianę zaszyfrowanych wiadomości tekstowych z innym użytkownikiem.

2. Stosowane aplikacje/narzędzia

Ćwiczenie wykonano korzystając z następujących aplikacji i narzędzi:

- Python 3
- Biblioteka Paho MQTT Client
- Baza danych do przechowywania kluczy publicznych użytkowników stworzona przy użyciu biblioteki sqlite3

3. Wykonanie ćwiczenia oraz zrzuty ekranu

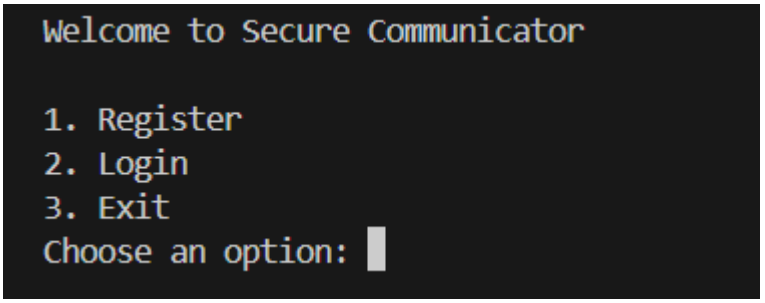
1. **Instalacja wymaganych bibliotek:**
    - paho-mqtt: do komunikacji MQTT
      Instalacja wykonywana jest komendą:
      `pip install paho-mqtt`
    - Pozostałe biblioteki są wbudowane w standardową bibliotekę Pythona, więc ich instalacja nie jest wymagana.

2. **Obsługa rejestracji i logowania użytkowników**
    - W pierwszej kolejności wykonywana jest funkcja, która tworzy dwie tabele - jedną z danymi użytkowników oraz drugą z ich kluczami publicznymi.

```
def create_tables():
    conn = connect_to_db()
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS users
                 (id INTEGER PRIMARY KEY, username TEXT UNIQUE, password TEXT)''')
    c.execute('''CREATE TABLE IF NOT EXISTS keys
                 (username TEXT UNIQUE, public_key INTEGER, n INTEGER)''')
    conn.commit()
    conn.close()
```

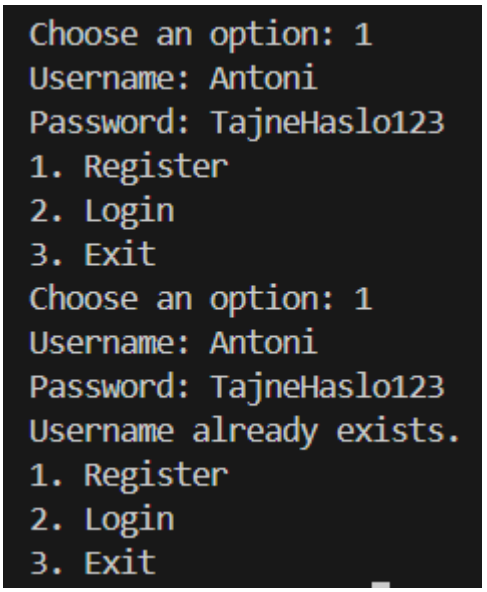
Obraz 2.1 - funkcja *create_tables()* do tworzenia tabel

    - Następnie wyświetlane jest menu, z którego użytkownik wybiera jedną z trzech opcji: rejestracja, logowanie oraz wyjście.



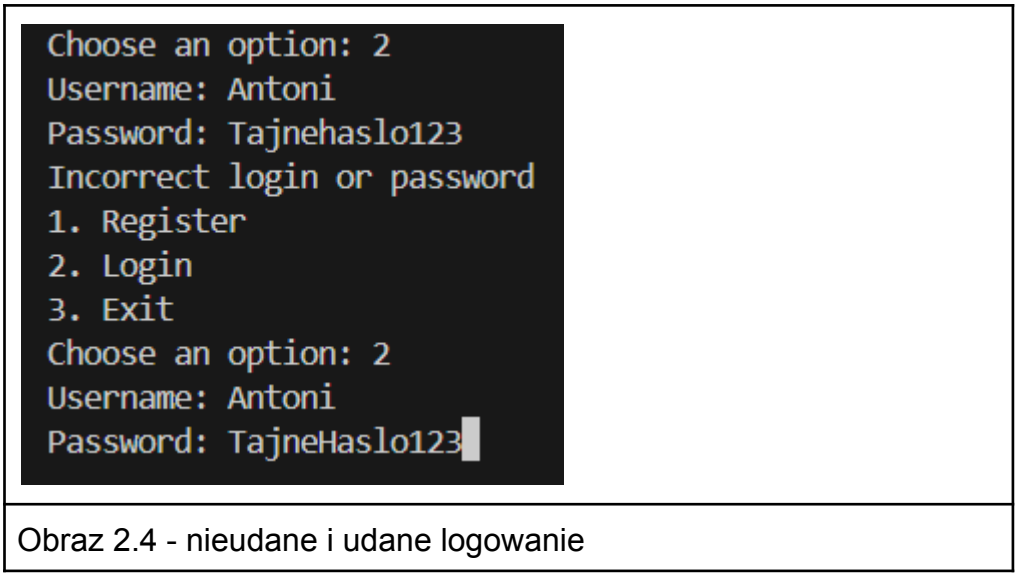
Obraz 2.2 - menu wyboru

- Przy rejestracji użytkownik podaje login oraz hasło. Następnie program łączy się z bazą danych i dodaje użytkownika. W przypadku istnienia użytkownika o takim loginie wyświetlany jest komunikat i użytkownik przenoszony jest ponownie do menu.



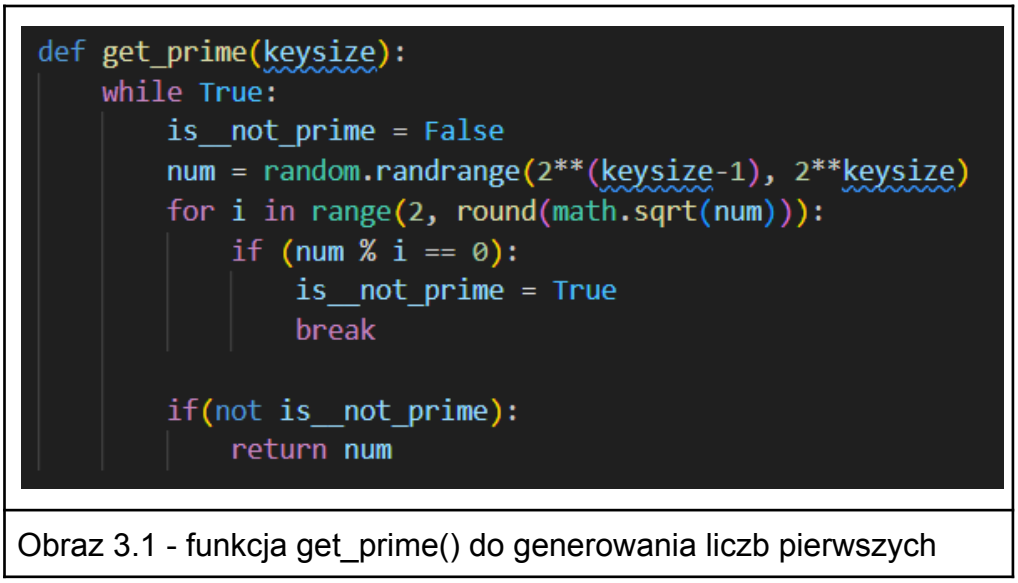
Obraz 2.3 - rejestracja udana i nieudana

- Podczas logowania użytkownik proszony jest o login oraz hasło. Następnie program łączy się z bazą danych. Jeżeli zwraca ona rekord, generowane są klucze. W przeciwnym razie wyświetlony jest komunikat i użytkownik przenoszony jest do menu.



Obraz 2.4 - nieudane i udane logowanie

3. **Generowanie kluczy RSA:**
   - Generowane klucze mają rozmiar 16 bitów, aby ich generowanie było jak najszybsze.
   - Program na początku generuje dwie liczby pierwsze p i q, każda o rozmiarze połowy rozmiaru klucza (w tym przypadku 8 bitów, ponieważ key_size wynosi 16).



```python
def get_prime(keysize):
    while True:
        is_not_prime = False
        num = random.randrange(2**(keysize-1), 2**keysize)
        for i in range(2, round(math.sqrt(num))):
            if (num % i == 0):
                is_not_prime = True
                break

        if(not is_not_prime):
            return num
```

Obraz 3.1 - funkcja get_prime() do generowania liczb pierwszych

   - Następnie wyliczana jest zmienna n, która jest iloczynem zmiennych p i q. Obliczana jest również liczba Eulera jako jako (p-1)*(q-1).
   - Kolejnym krokiem jest generowanie liczby e, która stanowi część klucza publicznego oraz liczby d stanowiącej część klucza prywatnego. Liczba e, czyli wykładnik publiczny, ma być względnie pierwszy z liczbą Eulera. Liczba d, czyli wykładnik prywatny, ma być odwrotnością modulo Eulera liczby e.

```python
def get_e(euler):
    # NWD(e, euler) = 1
    e = 2
    while True:
        if (math.gcd(e, euler) == 1):
            return e
        e += 1
```

```python
def get_d(e, euler):
    d = 2
    while True:
        if((d*e)%euler == 1):
            return d
        d += 1
```

Obraz 3.2, 3.3 - funkcje do generowania liczb e i d

- Po wygenerowaniu par liczb stanowiących klucz publiczny (n, e) są one zapisywane w bazie danych. Każdy użytkownik może posiadać tylko jedną taką parę kluczy, która generowana jest przed każdą konwersacją. Klucze są usuwane z bazy danych kiedy użytkownik kończy konwersacje.

4. **Implementacja szyfrowania i deszyfrowania:**
   - Po stworzeniu klienta MQTT i połączeniu z brokerem, użytkownik wpisuje wiadomość. Wiadomość ta jest szyfrowana kluczem publicznym adresata wiadomości.
   - Szyfrowanie rozpoczyna się od zmiany każdego znaku w wiadomości na jego kod ASCII.

```python
def to_number(message):
    to_ascii = []
    for letter in message:
        letter_code = ord(letter)
        to_ascii.append(letter_code)
    encrypted_message = encrypt(to_ascii)
    return encrypted_message
```

Obraz 4.1 - funkcja to_number() zmieniająca znak na kod ASCII

   - Następnie na liczbach wykonywana jest operacja szyfrowania ($c = t^e$ **mod** n, t - szyfrowana liczba). Kolejne znaki oddzielone są od siebie przecinkiem w celu ułatwienia dekodowania. Tak zakodowana wiadomość jest wysyłana.

```python
def encrypt(ascii):
    global recipient_public_key, recipient_n
    encrypted_message_list = []

    for char in ascii:
        encrypted = pow(char, recipient_public_key, recipient_n)
        encrypted_message_list.append(str(encrypted))

    encrypted_message = ','.join(encrypted_message_list)
    return encrypted_message
```

Obraz 4.2 - funkcja encrypt() do szyfrowania

- Deszyfrowanie następuje natomiast w momencie otrzymania wiadomości przy użyciu klucza prywatnego.
- Deszyfrowanie zaczyna się od podzielenia zaszyfrowanej wiadomości na listę zaszyfrowanych wartości, używając przecinków jako separatorów.

```python
def decoder(message):
    message_tab = message.split(",")
    message = ''
    for num in message_tab:
        message += chr(decode(num))
    return message
```

Obraz 4.3 - funkcja decoder() do dzielenia wiadomości

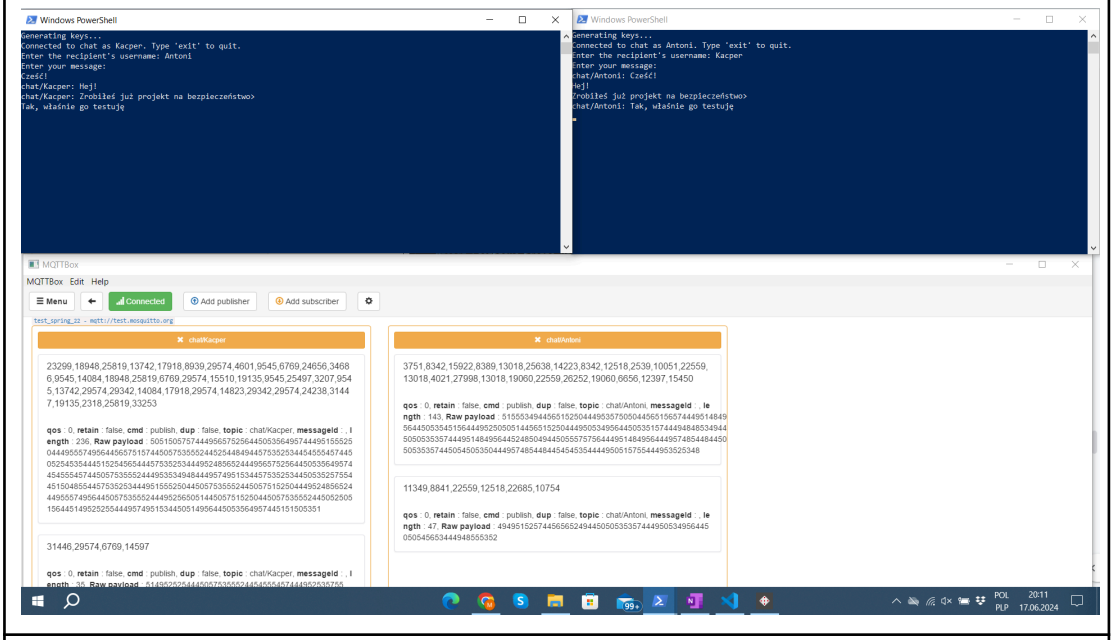
- Następnie każda wartość przekształcana jest na pierwotną wartość stosując wzór $t = c^d$ **mod** n, gdzie c to zaszyfrowana wartość.

```python
def decode(num):
    global private_key, n
    d = private_key
    num = int(num)
    decrypted = pow(num, d, n)
    return decrypted
```

Obraz 4.4 - funkcja decode() do deszyfrowania

- Zdeszyfrowana wiadomość wyświetlana jest użytkownikowi.

5. **Działanie programu**
   - Program uruchamia się komendą
     python .\chat.py



Obraz 5.1 - działanie programu

4. Wnioski z wykonanego projektu

Projekt wykazał, że możliwe jest stworzenie aplikacji do bezpiecznej komunikacji wykorzystującej szyfrowanie RSA i protokół MQTT. Kluczowe wnioski to:

1. **Bezpieczeństwo komunikacji**:
   - Szyfrowanie RSA zapewnia wysoki poziom bezpieczeństwa, ponieważ wiadomości są szyfrowane kluczem publicznym adresata i mogą być odszyfrowane tylko przy użyciu jego klucza prywatnego.
   - Protokół MQTT jest lekki i wydajny, co sprawia, że jest odpowiedni do aplikacji czasu rzeczywistego, takich jak czat. W połączeniu z szyfrowaniem, zapewnia bezpieczną wymianę wiadomości.

2. **Zarządzanie kluczami**:

- Implementacja wymaga skutecznego zarządzania kluczami publicznymi i prywatnymi użytkowników. Przechowywanie kluczy w bazie danych było kluczowe dla prawidłowego działania aplikacji.
- Konieczność usuwania kluczy z bazy danych po zakończeniu sesji czatu była istotna dla zapewnienia bezpieczeństwa.

3. **Obsługa wyjątków i błędów**:
   - Ważne było zaimplementowanie obsługi wyjątków, zwłaszcza w kontekście połączeń sieciowych i operacji na bazie danych. Problemy z połączeniem MQTT lub błędy podczas szyfrowania/deszyfrowania wiadomości muszą być odpowiednio obsłużone, aby zapewnić stabilność aplikacji.
4. **Skalowalność i wydajność**:
   - MQTT jest odpowiedni do skalowalnych systemów, jednak wymaga odpowiedniej konfiguracji brokera oraz zabezpieczeń w przypadku produkcyjnych wdrożeń.
   - Szyfrowanie RSA, mimo że bezpieczne, jest kosztowne obliczeniowo. Dla dużych wiadomości może być konieczne zastosowanie bardziej efektywnych metod szyfrowania w połączeniu z RSA dla wymiany kluczy.

Podsumowanie

Implementacja aplikacji do bezpiecznej komunikacji wykorzystującej szyfrowanie RSA i protokół MQTT pokazała, że jest to możliwe i efektywne rozwiązanie. Kluczowe aspekty to zapewnienie odpowiedniego zarządzania kluczami, stosowanie dodatkowych zabezpieczeń dla połączeń MQTT oraz efektywna obsługa wyjątków i błędów. Przy odpowiedniej konfiguracji i zarządzaniu, takie rozwiązanie może być skuteczne w wielu zastosowaniach wymagających bezpiecznej komunikacji w czasie rzeczywistym.

5. Literatura

https://pypi.org/project/paho-mqtt/

https://en.wikipedia.org/wiki/RSA_(cryptosystem)

https://eduinf.waw.pl/inf/utils/010_2010/0219.php

https://docs.python.org/3/library/sqlite3.html

https://mqtt.org/

https://github.com/Antasmg/bezpieczenstwo-projekt