Міністерство освіти і науки України
Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"
Фізико-технічний інститут

Проектування, розробка і реалізація криптографічних систем
Лабораторна робота №1

Дослідження реалізацій протоколу SSL

Виконав:
Студенти групи ФІ-42мн
Молдован Дмитро,
Сковрон Роман,
Волинець Сергій

Перевірив:
Байденко П. В.

Київ 2025

1. Мета та цілі

Мета: Дослідження особливостей реалізації криптографічних механізмів протоколу SSL/TLS.

Цілі:

- Створити серверну частину підтримувану ОС Windows (Python + OpenSSL).
- Створити клієнт для Android 11.0 або вище (або ОС iOS 17.0 або вище), який використовує TLS 1.3 з можливістю користувацького контролю.
- Побудувати проксі/логер, який дасть змогу користувачеві відображати всі TLS-пакети (record layer), блокувати перевірку сертифікатів, задавати конкретні значення випадкових змінних, задавати та згодом змінювати криптографічні методи, що використовуються.
- Виконати передачу пакетів клієнт ⇄ проксі ⇄ сервер, збереження і демонстрація пакетів у Wireshark.
- Виконати sslstrip-атаку на дану реалізацію протоколу і обґрунтувати результати.

2. Постановка задачі

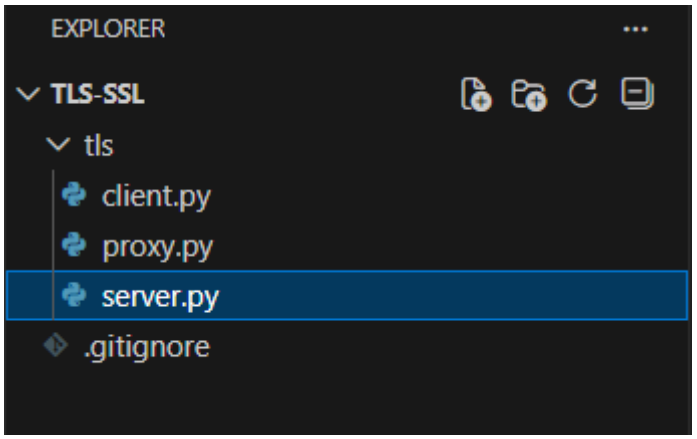
Розробити програмний засіб захисту логічного каналу зв’язку, що використовує протокол TLS (версія 1.3). Програмний засіб повинен мати хмарну архітектуру, в якій клієнт повинен бути орієнтованим на операційні системи та платформи для мобільних телефонів та планшетних комп’ютерів. Дозволяється використання бібліотеки OpenSSL або Crypto++ – реалізації з відкритим кодом.

Клієнт має підтримувати ОС Android 11.0 або вище, або ОС iOS 17.0 або вище. Серверна частина має підтримувати ОС Windows (версія ядра не менша за 10). Програмний засіб має реалізовувати автентифікацію обох сторін згідно з протоколом TLS. Реалізація протоколу має детально відображати процес формування нового сеансу та відновлення існуючого (має відображатися кожен з типів пакетів, його вміст та всі параметри

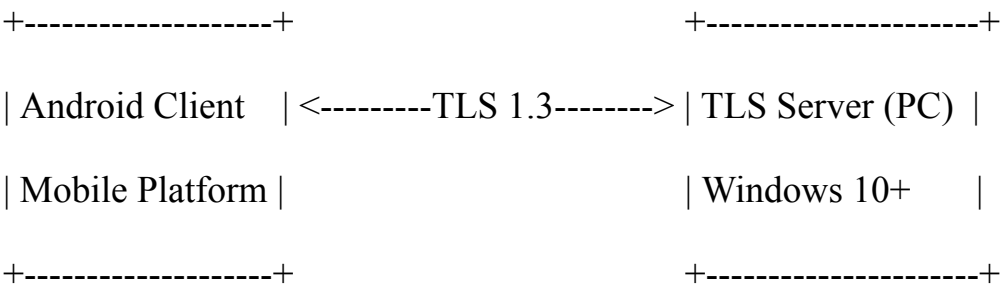
протоколу як на клієнті, так і на серверній частині), включаючи процес перевірки сертифікатів. Програмний засіб має надавати змогу користувачу безпосередньо впливати на цей процес з кожної із сторін, а саме блокувати перевірку сертифікатів, задавати конкретні значення випадкових змінних, задавати та згодом змінювати криптографічні методи, що використовуються. Після встановлення логічного каналу зв'язку програмний засіб повинен дозволяти проводити обмін довільними повідомлення за цим каналом. Коректність реалізації необхідно підтвердити за допомогою використання програми для перехоплення та аналізу пакетів – Wireshark. Також необхідно продемонструвати атаку sslstrip на власну реалізацію протоколу (можна використовувати допоміжні програми) та обґрунтувати результати застосування

3. Хід виконання роботи та опис труднощів

Для початку створимо папку нашого проекту, у якій зберігатимуться відповідні файли для серверу, клієнта та користувача:



Схематично роботу каналу зв'язку можна представили так:



Клієнт зв'язується з сервером по TLS каналу зв'язку, а між ними стоїть проксі сервер, який дозволяє користувачеві блокувати перевірку

сертифікатів, задавати конкретні значення випадкових змінних, задавати та згодом змінювати криптографічні методи, що використовуються.

Перейдемо до створення серверної частини (server.py), розглянемо лише ключові деталі:

```
1  import ssl
2  import socket
```

Підключення пакетів для створення сокету (socket) і обгортання його в TLS (ssl)

Далі визначимо серверу його роль та завантажимо сертифікат (cert.pem) і приватний ключ (key.pem), щоб сервер міг довести свою автентичність.

```
4  context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER) #defining server role
5  context.load_cert_chain("cert.pem", "key.pem")
```

Створимо звичайний сокет TCP, прив'яжемо його до IP та порту, налаштуємо його в режим прослуховування усіх інтерфейсів ("0.0.0.0") та поставимо в режим прослуховування

```
7  sock = socket.socket()
8  sock.bind(("0.0.0.0", 4443))
9  sock.listen()
```

Під кожне нове підключення виділяємо одне TCP з'єднання (conn) та IP (addr), а також виконуємо TLS handshake:

```
11 conn, addr = sock.accept()
12 tls = context.wrap_socket(conn, server_side=True)
```

Далі ми приймаємо зашифровані пакети, OpenSSL автоматично їх розшифровує та повертає вже звичайний текст, який ми виводимо на екран, даємо відповідь клієнту у форматі TLS (шифруємо, додаємо MAC тег) та закриваємо з'єднання:

```
data = tls.recv(1024)
print("Client says:", data.decode())
tls.send(b"hello from tls server")
tls.close()
```

Далі створимо користувацьку частину, яка стоятиме між клієнтом та сервером (proxy.py):

створимо парсер, який з клієнтського пакету витягуватиме необхідні дані для автентифікації та процедури handshake.

```
import argparse
import socket
import ssl
import threading
import struct

def parse_client_hello(data: bytes):
    try:
        if len(data) < 9: return {"error":"too short"}
        if data[0] != 22: return {"error":"not handshake record"}
        hs_type = data[5]
        if hs_type != 1: return {"error":"not ClientHello"}
        ptr = 9
        ptr += 2
        rand = data[ptr:ptr+32]; ptr += 32
        sid_len = data[ptr]; ptr += 1
        sid = data[ptr:ptr+sid_len]; ptr += sid_len
        cs_len = int.from_bytes(data[ptr:ptr+2],"big"); ptr +=2
        ciphers = []
        for i in range(0, cs_len, 2):
            ciphers.append("0x"+data[ptr:ptr+2].hex()); ptr +=2
        return {"random":rand.hex(),"session_id":sid.hex(),"cipher_suites":ciphers}
    except Exception as e:
        return {"error":str(e)}
```

надамо можливість проксі серверу переглядати пакети від кристувача, не витягаючи їх з TLS каналу. Таким чином користувач зможе переглядати ClientHello, розпарсити його, проаналізувати вибрані cipher suites / TLS

версію / extensions, але не розривати handshake:

```
def handle(conn, addr, args):
    print(f"[+] client {addr}")
    try:
        # peek the ClientHello bytes without consuming
        peek = conn.recv(4096, socket.MSG_PEEK)
        print("[>] Peeked bytes:", hexdump(peek,120))
        parsed = parse_client_hello(peek)
        print("[>] Parsed ClientHello:", parsed)

        # create server-side ctx (for client)
        srv_ctx = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        srv_ctx.load_cert_chain("certs/proxy.crt","certs/proxy.key")
        if args.client_ciphers:
            try: srv_ctx.set_ciphers(args.client_ciphers)
            except: pass

        client_tls = srv_ctx.wrap_socket(conn, server_side=True)
        print("[*] Handshake with client done.", client_tls.version(), client_tls.cipher())
```

Далі проксі створює з’єднання з клієнтом та виступаю для нього в ролі сервера, додамо змогу користувачу вимикати перевірку сертифікатів та вказувати свої шифри:

```
        # connect to upstream (real server)
        up_ctx = ssl.create_default_context()
        if not args.verify_up:
            up_ctx.check_hostname = False
            up_ctx.verify_mode = ssl.CERT_NONE
            print("[!] Upstream verification DISABLED")
        if args.up_ciphers:
            try: up_ctx.set_ciphers(args.up_ciphers)
            except: pass

        raw = socket.create_connection((args.up_host, args.up_port))
        upstream = up_ctx.wrap_socket(raw, server_hostname=args.up_host)
        print("[*] Connected to upstream:", upstream.version(), upstream.cipher())
```

Створимо з’єднання проксі з сервером (друга пара handshake)

```
        raw = socket.create_connection((args.up_host, args.up_port))
        upstream = up_ctx.wrap_socket(raw, server_hostname=args.up_host)
        print("[*] Connected to upstream:", upstream.version(), upstream.cipher())
```

Для того щоб об’єднати два канали зв’язку (client->server; server->client), створимо два потоки та скористаэмось функцією pipe для передачі даних в поток з відповідним логування і шифровкою, дешифровкою:

```
        # simple piping with logging
        def pipe(src,dst, label):
            try:
                while True:
                    b = src.recv(4096)
                    if not b: break
                    print(f"[{label}] {len(b)} bytes")
                    dst.sendall(b)
            except Exception as e:
                print(f"[{label}] error {e}")
            finally:
                try: dst.shutdown(socket.SHUT_RDWR)
                except: pass

        t1 = threading.Thread(target=pipe, args=(client_tls, upstream, "C->S"))
        t2 = threading.Thread(target=pipe, args=(upstream, client_tls, "S->C"))
        t1.start(); t2.start()
        t1.join(); t2.join()
    except Exception as e:
        print("[!] handle error", e)
    finally:
        try: conn.close()
        except: pass
        print("[*] closed", addr)
```

клієнтська частина створена за принципом серверу та функцій з proxy.py:

```
def run_once(host,port,verify,ciphers):
    ctx = ssl.create_default_context()
    ctx.minimum_version = ssl.TLSVersion.TLSv1_2
    if not verify:
        ctx.check_hostname = False
        ctx.verify_mode = ssl.CERT_NONE
```

```
def main():
    p = argparse.ArgumentParser()
    p.add_argument("--host", default="127.0.0.1")
    p.add_argument("--port", type=int, default=8444)
    p.add_argument("--verify", type=lambda s: s.lower() in ("1","true","yes"), default=True)
    p.add_argument("--ciphers", default=None)
    p.add_argument("--rounds", type=int, default=1)
    args = p.parse_args()

    for i in range(args.rounds):
        print(f"\n=== round {i+1} ===")
        run_once(args.host,args.port,args.verify,args.ciphers)
        time.sleep(1)

if __name__ == "__main__":
    main()
```

Перед початком роботи програми варто створити кореневі сертифікати, сертифікати для сервера, проксі та ключі:

```
mkdir -p certs
```

```
openssl genpkey -algorithm RSA -out certs/ca.key -pkeyopt rsa_keygen_bits:2048
```

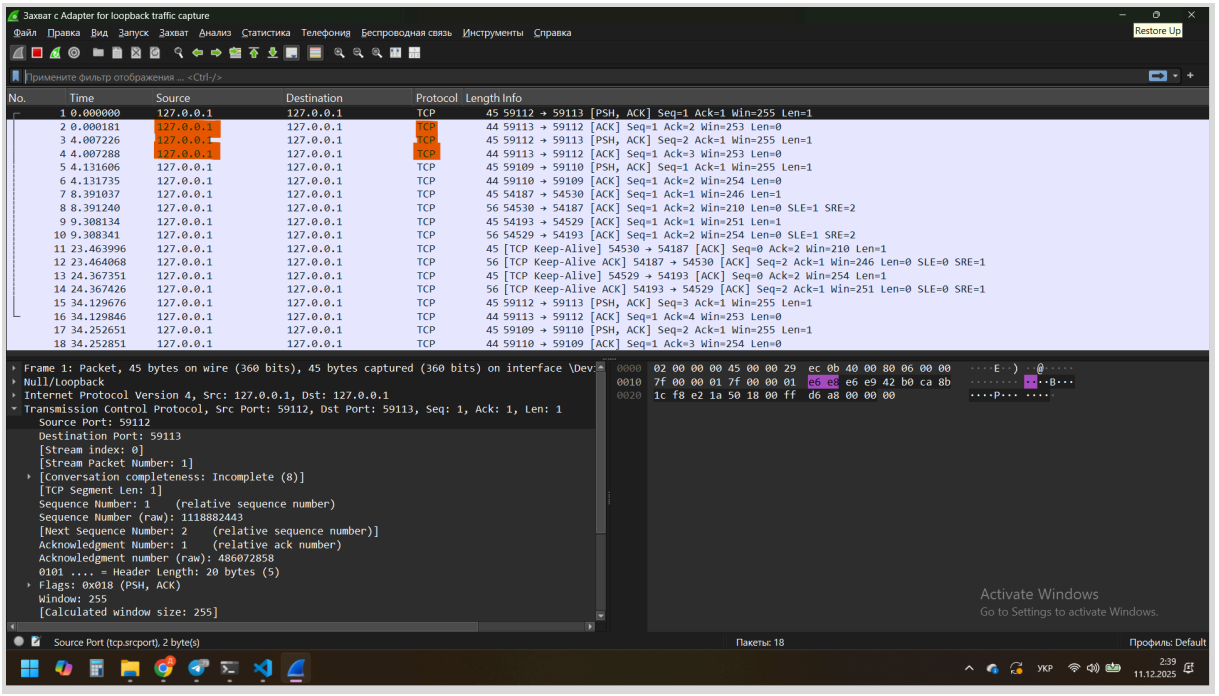
```
openssl req -x509 -new -nodes -key certs/ca.key -sha256 -days 3650 -out certs/ca.crt -subj "/CN=LabRootCA"
```

```
# server

openssl genpkey -algorithm RSA -out certs/server.key -pkeyopt rsa_keygen_bits:2048

openssl req -new -key certs/server.key -out certs/server.csr -subj "/CN=real.server.local"

openssl x509 -req -in certs/server.csr -CA certs/ca.crt -CAkey certs/ca.key -CAcreateserial -out certs/server.crt -days 365


# proxy cert (CA)

openssl genpkey -algorithm RSA -out certs/proxy.key -pkeyopt rsa_keygen_bits:2048

openssl req -new -key certs/proxy.key -out certs/proxy.csr -subj "/CN=proxy.local"

openssl x509 -req -in certs/proxy.csr -CA certs/ca.crt -CAkey certs/ca.key -CAcreateserial -out certs/proxy.crt -days 365
```

Приклад запуску сервера через командний рядок (порт:4443, шифрування: AES256, GCM:

```
python server.py --port 4443 --ciphers "TLS_AES_256_GCM_SHA384"
```

Результат перевірки з’єднання через Wireshark та вміст пакетів:



4. Структуру протоколів SSL та TLS та їх підпротоколів з визначенням функцій

Протоколи SSL та TLS забезпечують захищений обмін даними між клієнтом і сервером поверх TCP. Їхня архітектура складається з базового рівня Record Layer та кількох підпротоколів, які відповідають за встановлення і підтримку захищеного сеансу.

1. Record Layer

Базовий рівень, який фрагментує дані, шифрує їх, забезпечує цілісність (MAC/AEAD) і передає у вигляді записів. Використовується під час хендшейку та передачі даних.

2. Handshake Protocol

Підпротокол, що встановлює захищений сеанс: узгоджує версію TLS, криптографічні алгоритми, виконує автентифікацію (сервер, опційно клієнт) і обмін ключами, після чого формуються робочі ключі шифрування.

3. Change Cipher Spec

Службове повідомлення, яке сигналізує про перехід на узгоджені ключі та алгоритми шифрування. У TLS 1.3 майже не використовується.

4. Alert Protocol

Протокол повідомлень про помилки або завершення сесії (warning/fatal alerts, close_notify).

5. Application Data Protocol

Протокол передачі корисних даних, що працює після встановлення шифрованого каналу. Забезпечує конфіденційність і цілісність даних.

5. Порівняльний аналіз версій протоколів SSL 1.0, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3.

Характеристика	SSL	TLS
значення	SSL — *Secure Sockets Layer* (рівень захищених сокетів).	TLS — *Transport Layer Security* (протокол безпеки транспортного рівня).
Історія версій	SSL замінено на TLS. Існували версії SSL 1.0, 2.0, 3.0.	TLS — оновлена версія SSL. Має версії 1.0, 1.1, 1.2, 1.3.
Актуальність	Усі версії SSL застарілі та більше не використовуються.	Активно застосовуються TLS 1.2 та TLS 1.3.

Оповіщення (Alerts)	Є лише два типи оповіщень; вони *не шифруються*.	Оповіщення зашифровані та мають більшу різноманітність.
Аутентифікація повідомлень	Використовується MAC.	Використовується HMAC.
Набори шифрів (Cipher Suites)	Підтримуються старі алгоритми з відомими вразливостями.	Використовуються сучасні та безпечні алгоритми шифрування.
Рукостискання (Handshake)	Складне й повільне, містить більше кроків.	Простіше та швидше, оптимізоване за кількістю етапів.

6. Структуру сертифікатів (CA, CRL, OSCP)

Структура сертифікатів базується на стандартній ієрархії PKI. CA-сертифікат (Certificate Authority) — це кореневий або проміжний сертифікат, який підписує інші сертифікати і використовується для перевірки їх довіри. CRL (Certificate Revocation List) — це список відкликаних сертифікатів, який публікує CA, щоб клієнти могли визначити, чи не був сертифікат анульований. OCSP (Online Certificate Status Protocol) — це онлайн-механізм перевірки статусу сертифіката в реальному часі, який дозволяє дізнатися, чи дійсний сертифікат, без завантаження всього CRL. Разом ці компоненти забезпечують перевірку автентичності, цілісності та актуальності сертифікатів у TLS-з’єднанні.

7. Криптографічні методи, що використовуються в протоколах SSL та TLS

У даній реалізації: **ECDHE, AES-256-GCM, SHA-256, AEAD**
Загалом також підтримуються: RSA Key Exchange DH, DHE, ECDH, ECDHE, PSK (Pre-Shared Key), SRP, RSA, DSA (застарілий), ECDSA, Ed25519 / Ed448 (TLS 1.3), AES-GCM (рекомендовано), AES-CBC (вразливий, застарілий), ChaCha20-Poly1305 (сучасний, швидкий на мобільних), 3DES (застарілий), RC4 (небезпечний, заборонений), HMAC-SHA1 (застарілий), HMAC-SHA256 / SHA384, MD5 (небезпечно), HKDF-SHA256 (TLS 1.3)

8. Опис та класифікацію відомих вразливостей

Відомі вразливості SSL/TLS поділяють на кілька груп: криптографічні слабкості протоколів (BEAST, CRIME, BREACH, Lucky 13, POODLE — пов'язані зі старими режимами шифрування та компресією), атаки даунґрейду (FREAK, Logjam — змушують сторони перейти на слабкі ключі або алгоритми), проблеми переузгодження (renegotiation attack), атаки реалізації (Heartbleed, BERserk — помилки у конкретних бібліотеках), а також атаки на канал (sslstrip — перехоплення та примус роботи без HTTPS). Усі вони виникають через старі версії SSL, слабкі алгоритми або неправильні реалізації, тому сучасний TLS 1.3 вимикає застарілі механізми та значно зменшує площу атак.

9. Тексти всіх програм

Знаходяться у вигляді репозиторію за посиланням:
https://github.com/Dmytro-Mld/TLS-SSL

10. Детальний опис особливостей реалізації.

Для реалізації даної роботи обрано бібліотеку OpenSSL та мову програмування Python. У межах даної бібліотеки, канал зв'язку TLS зроблений на основі звичайного сокету TCP, з додаванням функцій шифрування, дешифрування, автентифікації сторін та перевірки сертифікатів.

На серверній стороні зберігаються сертифікати, ключі, cipher suites, правила перевірки, параметри протоколу.

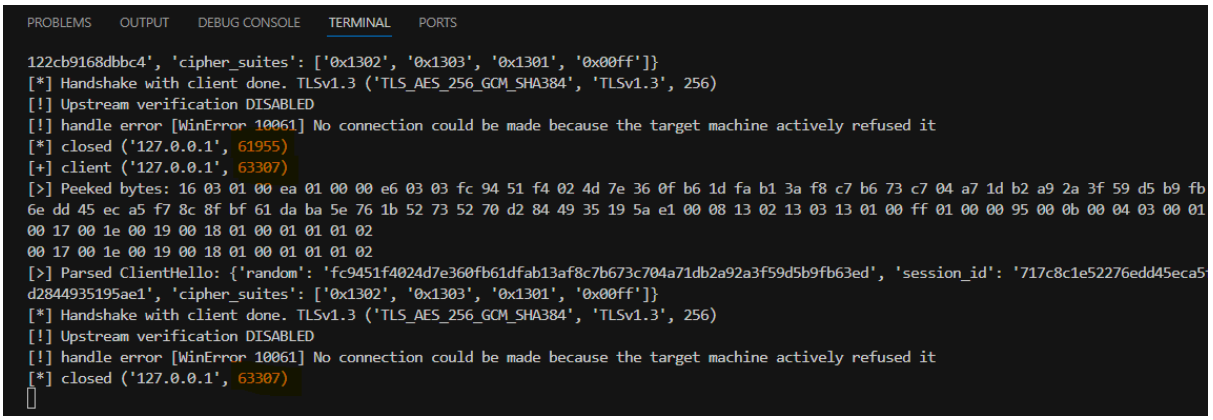
Сервер повинен мати **сертифікат** та **приватний ключ** інакше клієнт не зможе перевірити автентичність сервера.

Саме ця частина (серверний certificate + key), будуть показані клієнту під час TLS handshake.

Поверх вже створеного TCP з'єднання відбувається обгортка в зашифроване TLS з'єднання, а саме - TLS handshake: ClientHello -> ServerHello -> обмін ключами -> відправлення сертифіката -> перевірка автентичності -> встановлення сесійних ключів AES-GCM. Після цього tls вже є повністю захищеним каналом.

TLS канал створений за принципом MITM-TLS, що дає змогу користувачеві переглядати вміст пакетів, блокувати перевірку сертифікатів, переглядати вміст пакетів.

Кожного разу після встановлення нової сесії клієнта та користувача, виділяється новий порт:



Для відновлення сесії (коли не була виконана команда ssl.close()) порт зберігається.

11. Висновки до роботи.

Під час виконання даної лабораторної роботи нами було досліджено структуру каналів захищеної передачі пакетів типу TLS\SSL, а також реалізовано даний канал у межах локально розгорнутої мережі. Також було досліджено теоретичні властивості та особливості роботи даного типу передачі даних. Для аналізу роботи реалізованого каналу зв'язку було використано інструмент аналізу мережевих з'єднань Wireshark.