Bachelor of Science with Honours in Cyber Security

COURSEWORK No. 2

Module Name	:	**Cryptography**
Module Code	:	**PSB500IT**
Weighting	:	**60%**
Due Date	:	**28 February 2026**

Submitted By

Name	:	**Don Loo Boon Kiat**
CU Index No.	:	**16907748**
Word Count	:	**1349**

Contents

Abstract 3
Introduction 3
Technical Background 3
Mathematical Modelling 4
Terminal Demonstration 5
Wireshark Capture 5
GUI Incorporation 6
Secure Implementation 7
Workflow 8
Challenges encountered 9
Conclusion 10

Abstract

This report describes a Python based peer-to-peer chat application with end-to-end encryption to ensure direct TCP connection security. The security mechanisms used includes ephemeral key exchange with X25519, confidentiality with AES-256-CBC and authentication with HMAC-SHA256. This ensures the implementation of perfect forward secrecy (PFS), security against man-in-the-middle and replay attacks. A plain text-based approach is also presented to show the possibility of security risks. Wireshark is used to corroborate the encryption. Propositional logic models are used to ensure the implementation of fundamental cybersecurity principles.

Introduction

This project seeks to mitigate the risks that are associated with insecure plain text messaging. In this case, the messaging service is vulnerable to eavesdropping and data tampering. The project focuses on developing an alternative that is secure using standard libraries that are available for use. The project focuses on peer-to-peer TCP conversations. The project's objectives are stated as follows: using block ciphers, public key exchange and mitigating threats by using nonces/IVs and ephemeral keys.

Technical Background

Current secure messaging protocols use a combination of asymmetric and symmetric cryptography, dubbed hybrid cryptography and to enable efficiency in the process.

- X25519: This is an implementation of the elliptic curve Diffie-Hellman key exchange protocol which is employed to facilitate forward secrecy. This is done to ensure that previously conducted secure messages are not compromised even if the long-term keys are accessed.

- HKDF-SHA256 is a instance of HMAC-based key derivation function that uses SHA-256 as its underlying hash function.

- AES-256-CBC: This is employed to provide data integrity and authentication. The verification is done before the decryption process.

- HMAC-SHA256 is a keyed-hash message authentication code that combines a secret key with the SHA-256 cryptographic hash function to verify both data integrity and authenticity of a message.

- Threat Modelling employs propositional logic to systematically identify and mitigate key risks in P2P messaging, preserving the CIA triad.

Mathematical Modelling

Modelling the P2P Chat App with the given task of propositional logic, sets, functions and relations. Refer to the table below for the defined propositions.

Label	Logic
E	Encrypt
D	Decrypt
V	HMAC Verify
KX	Key Exchange

Following the table above, refer to the table below for the logic.

Label	Logic
$E \land \neg KX \rightarrow MITM$	Eavesdrop no exchange attack
$R \land \neg Fresh(IV) \rightarrow Replay$	Replay no nonce accept
$C \land Ephemeral \rightarrow \neg PFS$	Compromise + temp keys past safe

Next, setting of the representative system components are set. Refer to the table below for the defined sets.

Label	Logic
M	$\{plaintext\} \cup \{IV \mid\mid C \mid\mid MAC\}$
Keys	$AES_{256} \cup HMAC_{SHA256}(\ \mid (key \mid= 32B)$

Lastly, functions define computation. With everything aligned and ready for the P2P Chat App to run, refer to the table below for the functions.

Component	Notation	Description
Key Exchange	$KX: Priv \backslash times\ Pub \backslash to\ Shared \backslash quad\ (X25519)$	Maps private and public keys to a shared secret using X25519 ECDH.
Encryption Function	$f(m) = IV \backslash parallel\ AES_\{CBC\}(Pad(m)) \backslash parallel\ HMAC(IV \backslash parallel\ C)$	Constructs payloads: IV concatenated with padded AES-CBC ciphertext, then HMAC over IV and ciphertext.
Verification & Decrypt	$V(payload) \backslash to\ D(C) = m \backslash quad \backslash text\{or\ Reject\}$	HMAC verifies payload; if valid, decrypts ciphertext to original message or rejects.
Security Trust	$\backslash text\{Trust:\}\ (KX \backslash land\ Fresh \backslash land\ Ephemeral) \backslash to\ Secure$	Propositional logic: secure if key exchange is fresh and uses ephemeral keys.

Terminal Demonstration

The plain_chat.py file provides basic implementation of TCP sockets to facilitate two-way communication, but it does not encrypt the messages sent over the network.

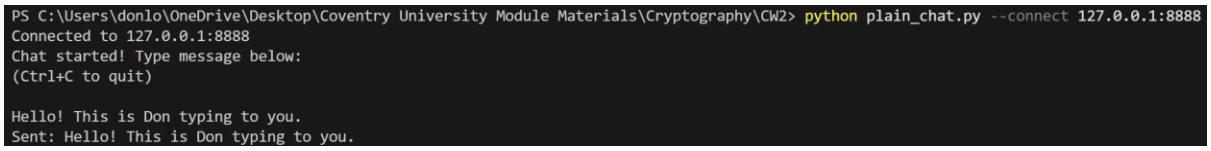


Figure 1 - Plain text chatting with no encryption [Sender]

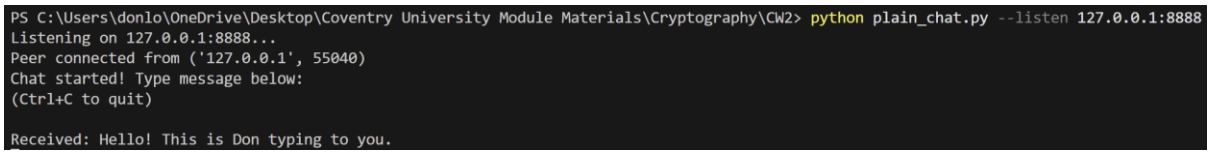


Figure 2 - Plain text chatting with no encryption [Receiver]

On the other hand, the secure_chat.py adds encryption to the process before threaded send/receive loops.

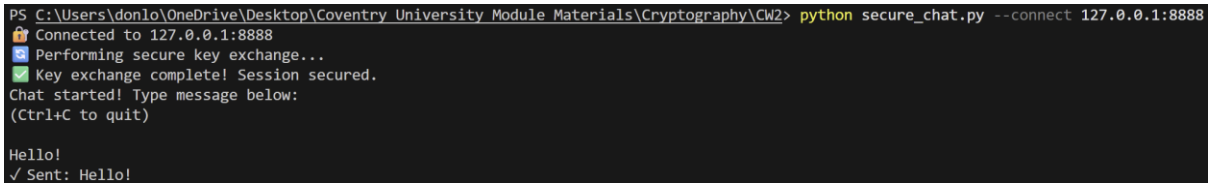


Figure 3 - Secure text chatting with encryption [Sender]

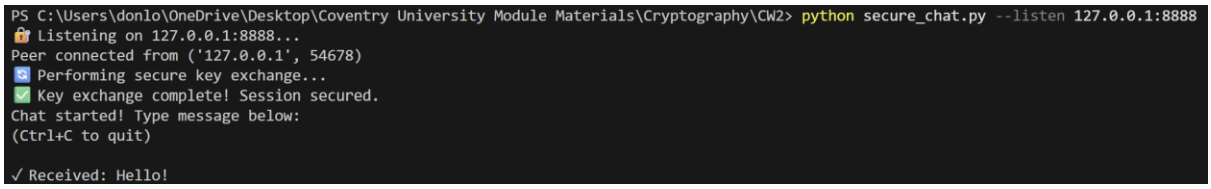


Figure 4 - Secure text chatting with encryption [Receiver]

Wireshark Capture

Using Wireshark to monitor the communications it is evident that when the user is using plain_chat.py, the message is able to be captured by Wireshark.

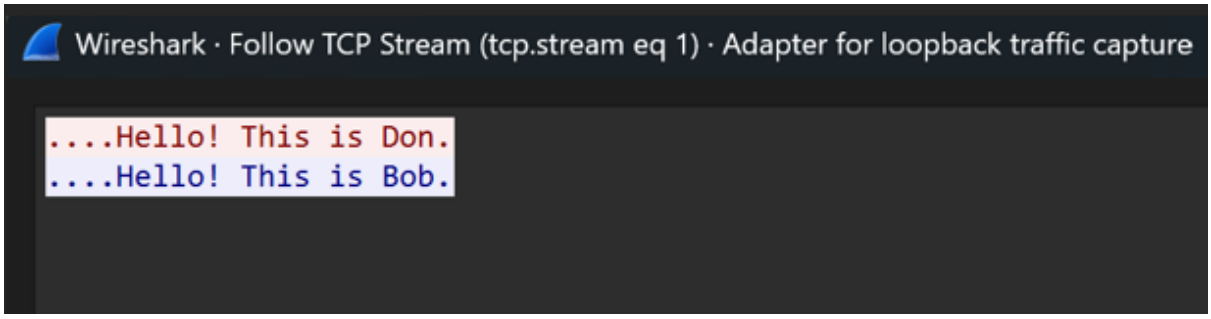


Figure 5 - Wireshark capturing chat [plain_chat.py]

However, when monitoring the communication traffic for secure_chat.py, the text shows up as encrypted gibberish.

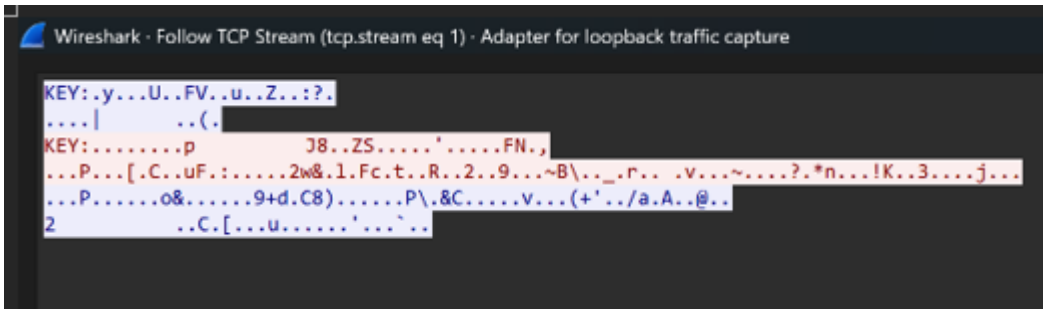


Figure 6 – Wireshark capturing chat [secure_chat.py]

GUI Incorporation

To reduce the limitation with only using the terminal, a modified version of the code in the program "secure_plain_gui.py" was implemented by using the Tkinter GUI package. This implementation combined the features of both the "plain chat" and "secure chat" modes in a unified interface.

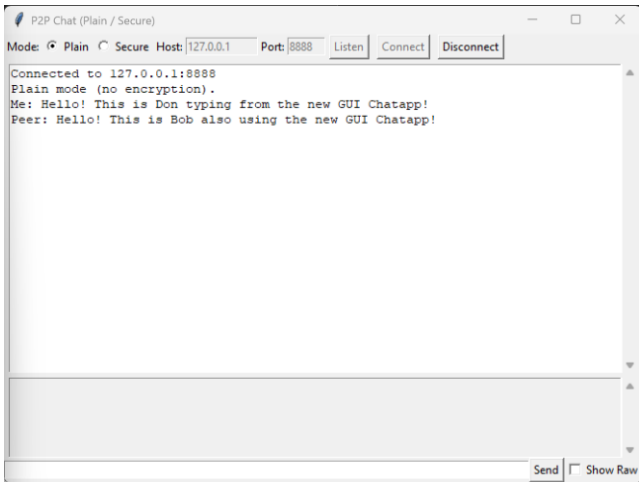


Figure 7 - GUI Chat App [Sender 's Plain Chat]

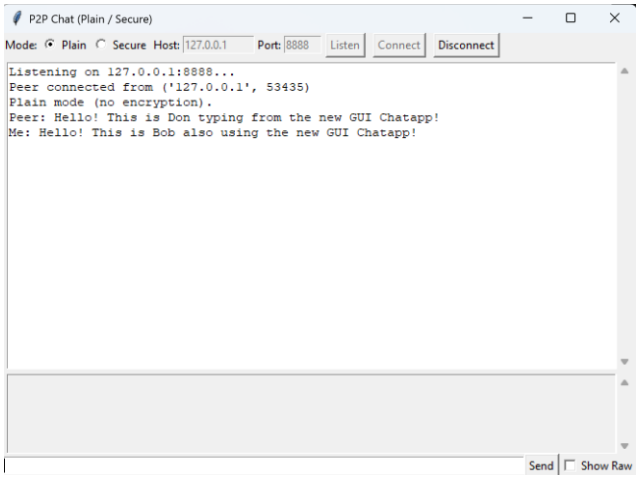


Figure 8 - GUI Chat App [Receiver's Plain Chat]

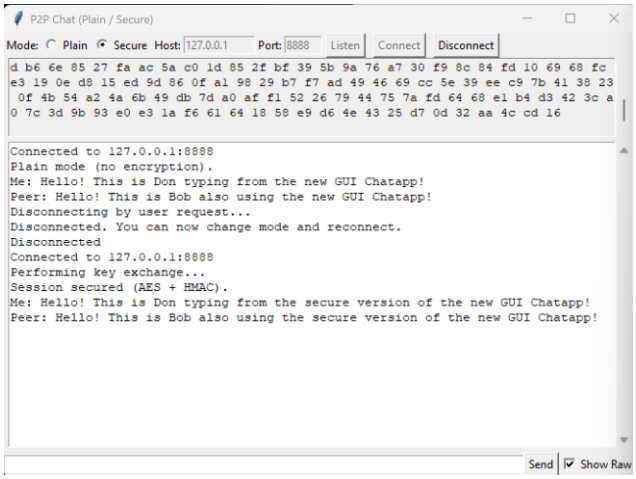


Figure 9 - GUI Chat App [Sender's Secure Chat]

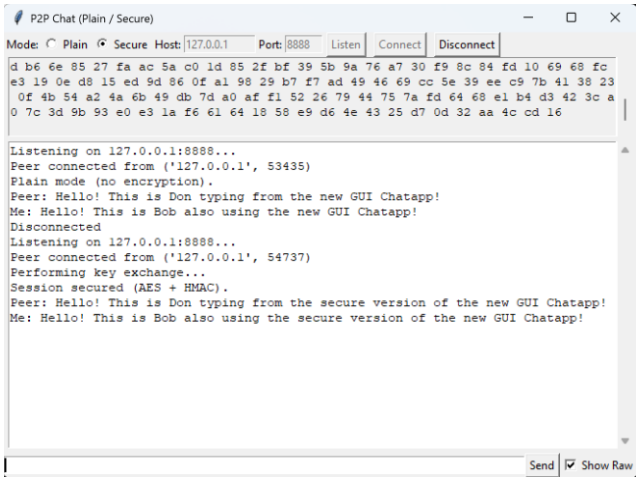


Figure 10 - GUI Chat App [Receiver's Secure Chat]

Also included in the GUI Chat app is the ability to inspect the raw text via the button on the bottom right of the GUI app saving the user time from using external apps like Wireshark. In the images above, the text is converted into readable ASCII bytes which directly matches Wireshark captures.

Secure Implementation

In the secure_chat.py script, the key exchange phase is included before the send and receive operations are performed in the threads. In this phase, key generation employs X25519 ECDH with peers exchanging 32-byte public keys over TCP, it derives a shared secret which is then sent to HKDF-SHA256. It expands it into 32-byte AES-256 and 32-byte HMAC-SHA256 keys for per session encryption and authentication with perfect forward secrecy.

```python
def key_exchange(sock):
    private_key = x25519.X25519PrivateKey.generate()
    public_key = private_key.public_key()

    our_pub_bytes = public_key.public_bytes(
        encoding=serialization.Encoding.Raw,
        format=serialization.PublicFormat.Raw
    )
    sock.sendall(b'KEY:' + our_pub_bytes)

    peer_msg = sock.recv(36)  # 'KEY:' + 32 bytes
    if not peer_msg.startswith(b'KEY:'):
        raise ValueError("Invalid peer key message")
    peer_pub_bytes = peer_msg[4:]
    peer_public_key = x25519.X25519PublicKey.from_public_bytes(peer_pub_bytes)

    shared_secret = private_key.exchange(peer_public_key)

    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=64,
        salt=None,
        info=b'secure_chat_session',
    )
    derived_keys = hkdf.derive(shared_secret)
    aes_key = derived_keys[:32]
    hmac_key = derived_keys[32:]

    return aes_key, hmac_key
```

Figure 11 - Key Exchange Function [Image]

Workflow

In plain mode, the sender turns text into bytes and then adds a 4-bytes length which is then sent over the network.

```python
def send_loop(sock, running):
    while running[0]:
        try:
            message = input().strip()
            if message:
                msg_bytes = message.encode('utf-8')
                sock.sendall(len(msg_bytes).to_bytes(4, 'big') + msg_bytes)
                print(f"Sent: {message}")
        except EOFError:
            break
        except KeyboardInterrupt:
            break
    running[0] = False
```

Figure 12 - Plain Chat send_loop [Image]

The receiver reads the length and fetches the bytes, turning them back into text and displaying it.

```python
def recv_loop(sock, running):
    while running[0]:
        try:
            length_bytes = sock.recv(4)
            if len(length_bytes) < 4:
                break
            length = int.from_bytes(length_bytes, 'big')
            if length > 0:
                message = b""
                while len(message) < length:
                    chunk = sock.recv(length - len(message))
                    if not chunk:
                        break
                    message += chunk
                print(f"Received: {message.decode('utf-8')}")
        except:
            break
    running[0] = False
```

Figure 13 - Plain Chat recv_loop [Image]

In secure mode, it includes the addition of PKCS7-pads, it then scrambles the message via AES encryption. Afterwards, it uses HMAC to authenticate tag sizes and lastly it sends it.

```python
def encrypt_message(message_bytes, aes_key, hmac_key):
    """AES-256-CBC encrypt + HMAC-SHA256"""
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv))
    encryptor = cipher.encryptor()

    # PKCS7 padding
    pad_len = 16 - (len(message_bytes) % 16)
    padded = message_bytes + bytes([pad_len] * pad_len)

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

    h = crypto_hmac.HMAC(hmac_key, hashes.SHA256())
    h.update(iv + ciphertext)
    message_mac = h.finalize()

    return iv + ciphertext + message_mac
```

Figure 14 - Secure Chat encryption [Image]

The receiver checks the proof first then verifies it via HMAC which then decrypts it and unpads the PKCS7. Decodes UTF-8 and lastly displays it.

```python
def decrypt_message(payload, aes_key, hmac_key):
    """Verify HMAC, AES-256-CBC decrypt, unpad"""
    iv = payload[:16]
    ciphertext = payload[16:-32]
    received_mac = payload[-32:]

    h = crypto_hmac.HMAC(hmac_key, hashes.SHA256())
    h.update(iv + ciphertext)
    h.verify(received_mac)

    # Decrypt
    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    padded = decryptor.update(ciphertext) + decryptor.finalize()

    # Unpad PKCS7
    pad_len = padded[-1]
    if pad_len < 1 or pad_len > 16:
        raise ValueError("Invalid padding")
    plaintext = padded[:-pad_len]
    return plaintext
```

Figure 15 - Secure Chat decryption [Image]

Challenges encountered

The project architecture only supports one-to-one connections between peers and does not support the concept of multicasting. If the application is to be scaled up for group messaging, it would require more bandwidth, separate IDs for the participants and monitoring the connections to handle the problem of dropouts from the multicast session.

While the design is for an intranet scenario for the application on the internet it would require the addition of defensive strategies such as the use of honeypots, monitoring the traffic and the use of deception-based traps to control the flow of malicious traffic

and protect the application from the threat of DDoS attacks. In addition, the application could benefit from the inclusion of vulnerability scan tools.

Conclusion

This project showcases the secure implementation of a peer-to-peer chat application that can transform a vulnerable plaintext message exchange into a secure end-to-end encrypted message exchange using the X25519 key exchange algorithm, AES-256-CBC encryption and HMAC-SHA256 authentication. The project eliminates major security risks such as the man-in-the-middle attack, replay attack and eavesdropping using the perfect forward secrecy and initialization vector techniques. The project is validated using the Wireshark protocol and terminal screenshots showing the obfuscation of the payload data, previously sent as plaintext messages.

Propositional threat modelling techniques are employed to make crucial project decisions, ensuring the preservation of the CIA triad in the real-world application of the project. Future directions of the project can be the use of authenticated symmetric key block cipher modes such as AES-GCM, the use of digital signatures, and the expansion of the project to multi-party UDP-based group chats, thereby connecting the realms of academic cryptography with real-world secure communication applications.