

## **21AD71- DATA SECURITY**

### **ASSIGNMENT - 3**

#### **1. Case Study:**

##### **SIGNAL APP**

##### **Overview**

Signal is a widely used secure messaging platform recognized for its commitment to privacy and security. By employing a robust suite of cryptographic techniques, Signal ensures that users' communications are protected from unauthorized access and tampering. This case study explores how Signal utilizes key exchange protocols, hash functions, message authentication codes (MACs), digital signatures, and concepts inspired by email security to safeguard user data.

#### **1. Key Exchange (Double Ratchet Algorithm)**

##### **Description**

The key exchange process in Signal is central to its security model. Signal implements the Double Ratchet Algorithm, which combines several cryptographic techniques, notably the Diffie-Hellman (DH) key exchange and symmetric encryption (AES). This algorithm enables secure communication between users even if previous keys are compromised.

##### **Process**

- **Initiation:** When two users wish to start a secure conversation, they perform an X3DH (Extended Triple Diffie-Hellman) handshake. This process involves exchanging public keys, which includes long-term identity keys and temporary ephemeral keys.
- **Key Generation:** Using these keys, a shared secret is generated. This shared secret is used to derive session keys for encrypting messages during the conversation.
- **Forward Secrecy:** Each message generates a new key for encryption, meaning that if a key is compromised, it only affects past messages and not future ones.

##### **Benefits**

- **Forward Secrecy:** Ensures that even if an attacker gains access to a session key, they cannot decrypt past messages.
- **Post-Compromise Security:** Future communications remain secure even if a key is compromised, protecting users from long-term vulnerabilities.

#### **2. Hash Functions**

##### **Description**

Hash functions are a vital part of ensuring data integrity within Signal. The app employs cryptographic hash functions, such as SHA-256, to create unique identifiers for messages and validate data integrity.

## **Process**

- **Key Agreement:** During the key exchange process, hash functions are used to create a hash of the exchanged keys, ensuring that both parties have arrived at the same key.
- **Message Integrity:** Each message is hashed before transmission, allowing the recipient to verify that the message has not been altered during transit.

## **Benefits**

- **Data Integrity:** Hash functions provide a way to verify that the content of the message has not been tampered with, as any modification would result in a different hash.
- **Authentication:** By hashing the keys involved in the exchange, Signal ensures that the keys are authentic and have not been modified.

## **3. Message Authentication Codes (MACs)**

### **Description**

Signal uses HMAC (Hash-based Message Authentication Code) to ensure the integrity and authenticity of messages sent between users. This cryptographic mechanism prevents unauthorized parties from modifying messages.

### **Process**

- **Creation of HMAC:** Each message is accompanied by an HMAC, which is generated using a shared cryptographic key between the sender and the recipient.
- **Verification:** When a message is received, the recipient computes the HMAC for the received message and compares it to the provided HMAC. If they match, the message is considered authentic and intact.

### **Benefits**

- **Message Integrity:** HMACs ensure that messages are received in their original form, preventing tampering or forgery.
- **Authentication:** HMACs authenticate the sender, providing assurance that the message is genuinely from the claimed sender.

## **4. Digital Signatures**

### **Description**

Digital signatures play a crucial role in authenticating users and verifying the integrity of communications within Signal. Signal utilizes the Elliptic Curve Digital Signature Algorithm (ECDSA) for signing and verifying messages.

### **Process**

- **User Authentication:** Each user generates a pair of cryptographic keys: a private key (kept secret) and a public key (shared). When a user sends a message, they sign it with their private key.
- **Verification:** The recipient can verify the signature using the sender's public key, confirming the identity of the sender and the integrity of the message.

## **Benefits**

- **Non-repudiation:** Digital signatures provide assurance that the sender cannot deny having sent the message, establishing accountability.
- **User Authentication:** Validates the identity of the sender, ensuring that users are communicating with the intended parties.

## **5. Email Security (Inspired Principles)**

### **Description**

While Signal is primarily a messaging app, it adopts principles from email security protocols such as PGP (Pretty Good Privacy) and S/MIME (Secure/Multipurpose Internet Mail Extensions) to enhance the privacy of user communications.

### **Process**

- **End-to-End Encryption:** Similar to encrypted email, messages in Signal are encrypted on the sender's device before being transmitted. Only the intended recipient can decrypt the messages using their private key.
- **No Third-Party Access:** Signal's design ensures that even the service provider cannot access the content of the messages, preserving user privacy.

## **Benefits**

- **Eavesdropping Prevention:** End-to-end encryption protects messages from interception, ensuring that only the intended recipient can read them.
- **User Privacy:** Users can communicate securely without fear of unauthorized access, maintaining confidentiality in their conversations.

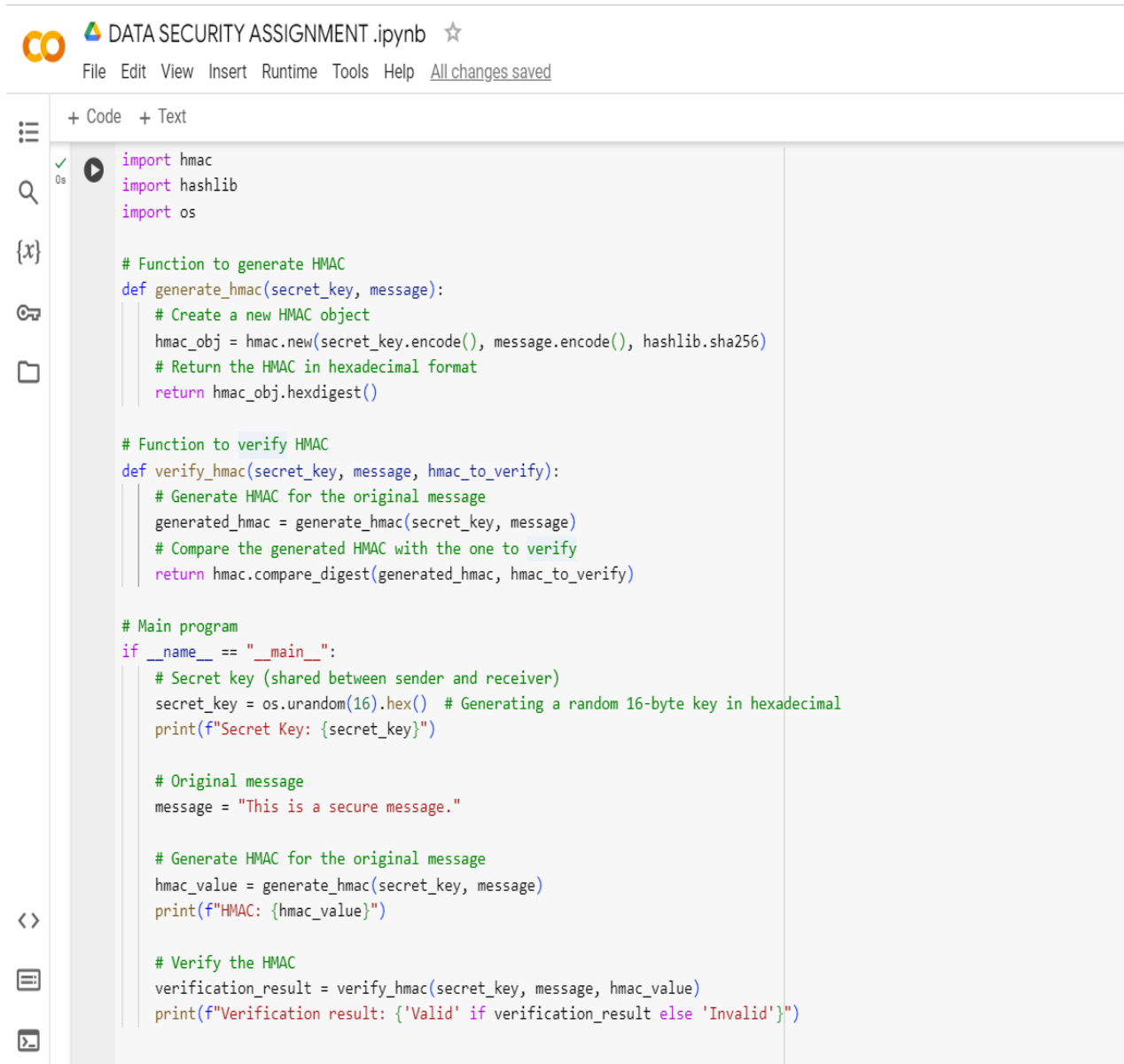
## **Conclusion**

Signal's commitment to privacy and security is reflected in its comprehensive use of advanced cryptographic techniques. By implementing a robust framework involving key exchange, hash functions, MACs, digital signatures, and principles inspired by email security, Signal provides a secure communication platform that protects users from eavesdropping, data tampering, and unauthorized access. As privacy concerns continue to rise in the digital age, Signal sets a benchmark for secure messaging applications, demonstrating how effective cryptographic practices can foster user trust and safety.

## 2. Practical Task:

### HMAC generation and verification using Python

#### Code:



The image shows a Jupyter Notebook interface with a file named "DATA SECURITY ASSIGNMENT .ipynb". The code is written in Python and demonstrates HMAC generation and verification using the hmac and hashlib libraries. The code includes comments for each step, from importing libraries to generating a random secret key, creating an HMAC object, generating the HMAC, and finally verifying it. The main program section shows the generation of a random 16-byte secret key, the original message "This is a secure message.", the generation of the HMAC, and the verification result, which is "Valid".

```
import hmac
import hashlib
import os

# Function to generate HMAC
def generate_hmac(secret_key, message):
    # Create a new HMAC object
    hmac_obj = hmac.new(secret_key.encode(), message.encode(), hashlib.sha256)
    # Return the HMAC in hexadecimal format
    return hmac_obj.hexdigest()

# Function to verify HMAC
def verify_hmac(secret_key, message, hmac_to_verify):
    # Generate HMAC for the original message
    generated_hmac = generate_hmac(secret_key, message)
    # Compare the generated HMAC with the one to verify
    return hmac.compare_digest(generated_hmac, hmac_to_verify)

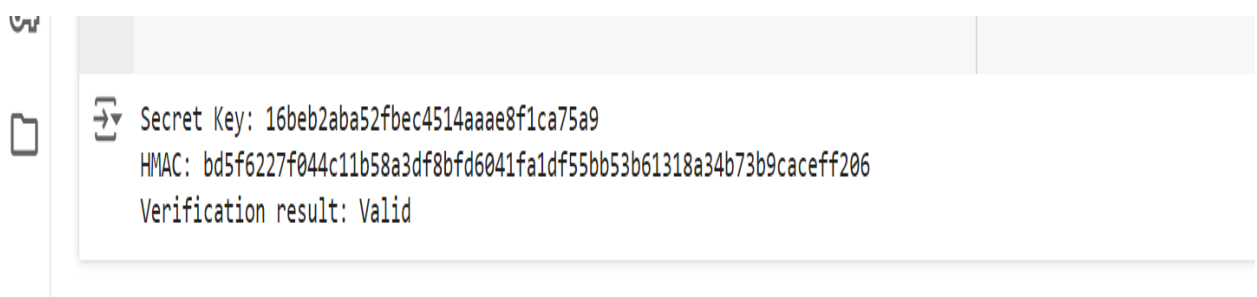
# Main program
if __name__ == "__main__":
    # Secret key (shared between sender and receiver)
    secret_key = os.urandom(16).hex() # Generating a random 16-byte key in hexadecimal
    print(f"Secret Key: {secret_key}")

    # Original message
    message = "This is a secure message."

    # Generate HMAC for the original message
    hmac_value = generate_hmac(secret_key, message)
    print(f"HMAC: {hmac_value}")

    # Verify the HMAC
    verification_result = verify_hmac(secret_key, message, hmac_value)
    print(f"Verification result: {'Valid' if verification_result else 'Invalid'}")
```

#### Output:



The output of the code execution is displayed in a Jupyter Notebook cell. It shows the generated secret key, the HMAC value, and the verification result.

```
Secret Key: 16beb2aba52fbec4514aaae8f1ca75a9
HMAC: bd5f6227f044c11b58a3df8bfd6041fa1df55bb53b61318a34b73b9caceff206
Verification result: Valid
```

## Cryptographic Techniques applied in this implementation:

- ✚ **HMAC:** Combines a secret key with a cryptographic hash function for message integrity and authenticity.
- ✚ **Secret Key:** A randomly generated 16-byte key used for HMAC creation and verification.
- ✚ **Hash Function:** Uses SHA-256 to generate a fixed-size hash value from the message and key.
- ✚ **HMAC Generation:** Encodes the key and message, creates an HMAC object, and returns the hexadecimal representation.
- ✚ **HMAC Verification:** Generates a new HMAC from the original message and compares it with the provided HMAC using secure comparison.
- ✚ **Secure Random Key:** The key is generated using `os.urandom()`, ensuring cryptographic strength.
- ✚ **Integrity Check:** Any changes to the message after HMAC generation will result in a different HMAC during verification.
- ✚ **Collision Resistance:** SHA-256 is designed to resist collisions, making it difficult to find two inputs with the same hash.
- ✚ **Timing Attack Prevention:** Uses `hmac.compare_digest()` for secure HMAC comparison to avoid timing attacks.
- ✚ **Authentication Assurance:** Ensures that only parties with the secret key can generate and verify valid HMACs for a message.