MESP-SDR-Crypto

Simplon Crypto 01

Explication du script Python

Ce document explique comment le script Python fourni répond aux exigences des différentes questions. Nous allons analyser les différentes étapes telles qu'elles sont réalisées par le script.

1. Génération d'une clé de chiffrement AES256 et des IV avec le destinataire

Le script permet de générer une clé AES en utilisant la fonction suivante :

```
def generate_aes_key(self):
    aes_key = os.urandom(32)
    with open("aes.key", "wb") as aes_file:
        aes_file.write(aes_key)
    messagebox.showinfo("Info", "AES Key generated and saved as aes.key")
```

Cette méthode génère une clé AES de 256 bits (32 octets), ce qui correspond à la taille nécessaire pour le chiffrement AES256. La clé est générée de manière sûre en utilisant `os.urandom()`, qui est une source de génération de nombres aléatoires cryptographiquement sécurisés. La clé est ensuite sauvegardée dans un fichier.

Les IV

Les IV (vecteurs d'initialisation) sont générés dynamiquement au moment du chiffrement dans cette partie du code :

```
iv = os.urandom(16)
cipher = AES.new(aes_key, AES.MODE_CFB, iv)
```

À chaque opération de chiffrement, un IV de 16 octets est généré de manière aléatoire pour être utilisé avec le chiffrement AES en mode CFB. Cela garantit que les IV ne sont pas réutilisés, ce qui est essentiel pour éviter les vulnérabilités dans le schéma de chiffrement.

Pourquoi est-il risqué d'avoir des IV constants ?

Si les IV étaient toujours les mêmes, cela entraînerait un risque important, car un attaquant pourrait remarquer des schémas répétitifs dans les messages chiffrés. Cela affaiblit la sécurité du chiffrement en permettant d'éventuellement révéler des informations sur le message original.

2. Partage sécurisé de la clé de chiffrement avec le destinataire

Le script permet de chiffrer la clé AES en utilisant la clé publique RSA du destinataire pour garantir un partage sécurisé.

La fonction suivante réalise cette tâche :

```
def encrypt_aes_key(self):
    pub_key_path = filedialog.askopenfilename(title="Select RSA Public Key
File")
    with open(pub_key_path, "rb") as pub_file:
        public_key = RSA.import_key(pub_file.read())
    cipher_rsa = PKCS1_OAEP.new(public_key)
    encrypted_aes_key = cipher_rsa.encrypt(aes_key)
    with open("encrypted_aes.key", "wb") as enc_file:
        enc_file.write(encrypted_aes_key)
```

Ici, la clé AES est chiffrée à l'aide de l'algorithme RSA et la méthode `PKCS1_OAEP`, qui est un schéma de chiffrement RSA sécurisé et résistant aux attaques par oracle. La clé publique RSA est utilisée pour chiffrer la clé AES, garantissant que seul le destinataire, qui possède la clé privée RSA, peut la déchiffrer.

3. Chiffrer un message

Le script fournit une fonctionnalité pour chiffrer un message en utilisant la clé AES et l'IV généré aléatoirement, comme illustré ci-dessous :

```
def encrypt_message(self):
    with open(key_path, "rb") as key_file:
        aes_key = key_file.read()
    iv = os.urandom(16)
    cipher = AES.new(aes_key, AES.MODE_CFB, iv)
    plaintext = self.message_entry.get().encode()
    ciphertext = iv + cipher.encrypt(plaintext)
    with open(enc_message_path, "wb") as enc_file:
        enc_file.write(ciphertext)
```

Le message est chiffré en mode CFB (Cipher Feedback), qui permet de chiffrer des messages de longueur variable. L'IV et le message chiffré sont ensuite sauvegardés dans un fichier.

4. Recevoir et déchiffrer un message

Le script inclut également une fonction pour déchiffrer un message, en utilisant la clé AES et l'IV récupérés depuis le message chiffré :

```
def decrypt_message(self):
    with open(key_path, "rb") as key_file:
        aes_key = key_file.read()
    with open(enc_message_path, "rb") as enc_file:
        ciphertext = enc_file.read()
```

```
    iv = ciphertext[:16]
    cipher = AES.new(aes_key, AES.MODE_CFB, iv)
    plaintext = cipher.decrypt(ciphertext[16:]).decode()
```

Le déchiffrement se fait en récupérant l'IV et en déchiffrant le reste du message à l'aide de la clé AES. Le message est ensuite affiché en clair.

5. Authentification du destinataire et intégrité du message

Pour garantir l'authenticité du destinataire et l'intégrité du message, nous pourrions utiliser une signature numérique RSA sur le message chiffré. Une signature garantit que le message n'a pas été altéré et provient du bon expéditeur. Cela peut être fait en ajoutant une fonction supplémentaire qui signe le message avant de l'envoyer.

Voici un exemple de code qui pourrait être ajouté au script pour signer et vérifier le message :

```
from Cryptodome.Signature import pkcs1_15
from Cryptodome.Hash import SHA256

def sign_message(self, message):
    priv_key_path = filedialog.askopenfilename(title="Select RSA Private
Key File")
    with open(priv_key_path, "rb") as priv_file:
        private_key = RSA.import_key(priv_file.read())
    h = SHA256.new(message)
    signature = pkcs1_15.new(private_key).sign(h)
    return signature

def verify_signature(self, message, signature):
    pub_key_path = filedialog.askopenfilename(title="Select RSA Public Key
File")
    with open(pub_key_path, "rb") as pub_file:
        public_key = RSA.import_key(pub_file.read())
    h = SHA256.new(message)
    try:
        pkcs1_15.new(public_key).verify(h, signature)
        return True
    except (ValueError, TypeError):
        return False
```

Cette fonctionnalité utilise l'algorithme RSA et SHA256 pour créer une signature numérique, qui peut ensuite être vérifiée par le destinataire pour assurer l'intégrité et l'authenticité du message.

Conclusion QUestion 1 à 5

Le script Python fourni répond aux exigences des questions initiales en permettant de :

- Générer une clé AES et des IV de manière sécurisée.
- Chiffrer et déchiffrer la clé AES avec RSA pour un partage sécurisé.

- Chiffrer et déchiffrer un message à l'aide d'AES.
- Ajouter des fonctionnalités pour vérifier l'authenticité du destinataire et l'intégrité du message via des signatures numériques.

L'utilisation d'un algorithme postquantique n'est pas abordée dans ce script, mais pourrait être ajoutée dans une version future.

7. Décryptage Caesar

=> "prggr grpuavdhr f'nccryyr yr puvsserzrag qr prnfre, vy a'rfg cyhf hgvyvft nhwbheq'uhv, pne crh ftphevft" => https://www.dcode.fr/identification-chiffrement => https://www.dcode.fr/chiffre-rot-13 => "cette technique s'appelle le chiffrement de ceaser, il n'est plus utilisg aujourd'hui, car peu sgcurisg"

8. Analyse des anomalies dans les messages chiffrés d'Alice : Réutilisation d'IV et incohérences de taille

Les messages d'Alice contiennent plusieurs fois le même texte chiffré '\xde@=\x1ed\xc0Qe\x0fK=\x1c\xb3$\xd9\xcb'. Ce texte devrait être différent à chaque chiffrement avec un IV différent. La réutilisation du même bloc chiffré avec la même clé et IV peut indiquer une réutilisation de la clé ou de l'IV, ce qui est un signe de problème potentiel. En outre, certains messages semblent être de longueur différente, ce qui pourrait être dû par exemple à une erreur de padding.

9. Brute force AES

b'12345678bien' => 12 octets sur les 16 => 4 caractères a décrypter

on vérifie que tous les caractères contenus dans le message déchiffrés sont bien dans la table ascii

```
def is_ascii(s):
    """Check if all bytes in the string are within the ASCII range (0-
127)"""
    return all(0 <= byte < 128 for byte in s)
```

on itère sur les 10 caractères numériques et les 26 caractères minuscule sur 4 octets

```
def brute_force(ciphered, partial_key, num_octets, alphabet):
    missing_key_length = num_octets - len(partial_key)

    # Create an SQLite database to store the results
    conn = sqlite3.connect('bruteforce_results.db')
    c = conn.cursor()

    # Drop the table if it exists, ensuring it's emptied each time the
script is run
    c.execute("DROP TABLE IF EXISTS results")

    # Dynamically create a table with fields char0...charN, IS_ASCII, and
UNCIPHERED
    fields = ', '.join([f'char{i} TEXT' for i in
```

```
range(missing_key_length)]) + ', IS_ASCII BOOLEAN, UNCIPHERED BLOB'
    c.execute(f"CREATE TABLE results ({fields})")

    # Generate all possible combinations for the missing key characters
    total_combinations = len(alphabet) ** missing_key_length
    with tqdm(total=total_combinations, desc="Brute Forcing", ncols=100) as
pbar:
        for combo in itertools.product(alphabet,
repeat=missing_key_length):
            key_guess = partial_key + ''.join(combo).encode('utf-8')

            try:
                # Try to decrypt with the current key guess
                decrypted_message = des_decrypt(key_guess, ciphered)

                # Check if all characters in the decrypted message are
within the ASCII range
                ascii_check = is_ascii(decrypted_message)

                # Insert into database: combination of characters, the
ASCII check, and the raw unciphered message
                values = ', '.join([f"'{char}'" for char in combo] +
[str(int(ascii_check)), "?"])
                c.execute(f"INSERT INTO results VALUES ({values})",
(decrypted_message,))

            except Exception as e:
                # If there's an error (likely wrong key), skip this
combination
                pass

            # Update progress bar
            pbar.update(1)

    # Commit and close database connection
    conn.commit()
    conn.close()
```

résultat =>

Field	Value
char0	j
char1	o
char2	u
char3	e
IS_ASCII	1
UNCIPHERED	DES n'est plus sur de nos jours!