LEPRAT Florian
Oubeid Allah JABALLAH

Introduction to Cryptography ESISAR (2024-2025)TP1

Questions de cours

Quelles sont les deux grandes familles de la cryptographie moderne ?

- La cryptographie à clé symétrique
- La cryptographie à clé asymétrique

Quelles propriétés doivent vérifier les fonctions de hachage ?

1. Déterminisme : Le même entrée donne toujours le même résultat.
2. Résistance à la préimage : Il doit être difficile de retrouver l'entrée à partir de la sortie.
3. Résistance à la seconde préimage : Il doit être difficile de trouver une autre entrée qui produit le même hachage qu'une entrée donnée.
4. Sortie fixe : La longueur du résultat est fixe et prédéfinie.

Quels sont les deux types de chiffrement symétrique ?

1. Le chiffrement par bloc :
    - Chiffre les données en blocs de taille fixe.
    - L'AES est un exemple d'algorithme de chiffrement par bloc.
2. Le chiffrement par flux :
    - Chiffre les données en continu sans découper le message en blocs.
    - RC4 est un exemple de chiffrement par flux.

Qu'est-ce que la malléabilité et comment s'en protéger?

La malléabilité fait référence à la capacité d'un message chiffré à être modifié de manière à produire un message déchiffré valide, sans connaître la clé de déchiffrement.

Pour s'en protéger :

- Utilisez des modrique ?es de chiffrement intégrant des mécanismes d'authentification.
- Utilisez l'authentification par code d'authentification de message (MAC/TAG).

- Utilisez des signatures numériques.

Challenges

#Challenge #1 Hash - SHA-2

Analyse du hash

> Le hash fourni est :
> 96719db60d8e3f498c98d94155e1296aac105ck4923290c89eeeb3ba26d3eef92

Observation importante

Un élément dans le hash n'est pas un caractère hexadécimal valide. Cela suggère une corruption.

Solution

1. Enlever le caractère suspect ("k") du hash.
2. Le hash modifié est un SHA-256 standard (64 caractères hexadécimaux).
3. Rechercher ce hash dans une base de données comme CrackStation.net.

Résultat

Après recherche, le hash correspond au mot de passe : 4dM1n

Le sha-1 du mot de passe est le suivant:
a7c9d5a37201c08c5b7b156173bea5ec2063edf9

#Challenge #2 Clair connu XOR

> Cette image au format BMP a été chiffrée par inadvertance.

On a d'abord inspecté les premières 20 octets et on sait que le header peut etre facilement devinable: https://en.wikipedia.org/wiki/BMP_file_format.

```
0x42, # BM in ASCII
0x4D,

0xF6, # file size: 495606
0x8F,
0x07,
0x00,

0x00, #Reserved most probably 0
0x00,
```

```
    0x00, #Reserved most probably 0
    0x00
```

En trouvant le debut de la clef nous retrouvons notre pattern:

```
# key: 102
# key: 97
# key: 108
# key: 108
# key: 101
# key: 110
# ==== REAPEAT ===
# key: 102
# key: 97
# key: 108
# key: 108
```

Donc la clef est tres probablement seulement: [102,97,108,108,101,110] => “fallen”

Après cela, nous avons simplement écrit ce petit programme pour décrypter.

```
image = open("TP1_challenge.bmp","rb").read()

def keys():
    for key in range(256):
        yield key

KNOWN=[
    0x42, # BM in ASCII
    0x4D,

    0xF6, # file size: 495606
    0x8F,
    0x07,
    0x00,

    0x00, #Reserved most probably 0
    0x00,

    0x00, #Reserved most probably 0
    0x00,
]
for part_img,part_known in zip(image,KNOWN):
    for key in keys():
```

```
        if part_img^key==part_known:
            print("key:",key)
# key: 102
# key: 97
# key: 108
# key: 108
# key: 101
# key: 110
# ==== REAPEAT ===
# key: 102
# key: 97
# key: 108
# key: 108
KEY=[102,97,108,108,101,110]

output=open("out.bmp","wb")
for i in range(len(image)):
    res = image[i] ^ KEY[i%len(KEY)]
    output.write(res.to_bytes(1, 'little'))
```

On a trouvé l'image suivante.

SYMMETRIC

En partant d'une implémentation quelconque de chiffrement AES, implémenter l'algorithme GCM (chiffrement et déchiffrement).

```
from Crypto.Cipher import AES
from Crypto.Util import Counter
```

```
from Cryptodome.Random import get_random_bytes
from Crypto.Util.number import long_to_bytes, bytes_to_long
from Cryptodome.Cipher import AES


def gf128_mul(a, b):
    res = 0
    for bit_position in range(127, -1, -1):
        if (b >> bit_position) & 1:
            res ^= a
        a = (a >> 1) ^ ((a & 1) * 0xE1 * (16 ** 30))
    return res

def precompute_table(auth_key):
    table = []
    for i in range(16):
        row = []
        for j in range(256):
            row.append(gf128_mul(auth_key, j << (8 * i)))
        table.append(tuple(row))
    return tuple(table)

class AES_WITH_GCM:
    def __init__(self, master_key):
        self.master_key = master_key
        self.aes = AES.new(self.master_key, AES.MODE_ECB)
        self.auth_key = bytes_to_long(self.aes.encrypt(b'\x00' *
16))
        self.pre_table = precompute_table(self.auth_key)

    def calc_auth_key(self, val):
        res = 0
        for i in range(16):
            res ^= self.pre_table[i][val & 0xFF]
            val >>= 8
        return res

    def _pad_data(self, data):
        pad_len = 16 - (len(data) % 16)
        return data if pad_len == 16 else data + b'\x00' *
pad_len

    def ghash(self, aad, txt):
        data = self._pad_data(aad) + self._pad_data(txt)
```

```
        tag = 0
        for i in range(len(data) // 16):
            tag ^= bytes_to_long(data[i * 16: (i + 1) * 16])
            tag = self.calc_auth_key(tag)
        tag ^= ((8 * len(aad)) << 64) | (8 * len(txt))
        return self.calc_auth_key(tag)

    def encrypt(self, init_value, plaintext, auth_data=b""):
        init_value = bytes_to_long(init_value)
        counter = Counter.new(
            nbits=32,
            prefix=long_to_bytes(init_value, 12),
            initial_value=2,
            allow_wraparound=False,
        )
        aes_ctr = AES.new(self.master_key, AES.MODE_CTR,
counter=counter)
        padded_plaintext = self._pad_data(plaintext)
        ciphertext = aes_ctr.encrypt(padded_plaintext)[:
len(plaintext)]
        auth_tag = self.ghash(auth_data, ciphertext)
        auth_tag ^= bytes_to_long(
            self.aes.encrypt(long_to_bytes((init_value << 32) |
1, 16))
        )
        return ciphertext, auth_tag

    def decrypt(self, init_value, ciphertext, auth_tag,
auth_data=b""):
        init_value = bytes_to_long(init_value)
        auth_tag = bytes_to_long(auth_tag)
        ghash = self.ghash(auth_data, ciphertext) ^
bytes_to_long(
            self.aes.encrypt(long_to_bytes((init_value << 32) |
1, 16))
        )
        if auth_tag != ghash:
            raise Exception(f"Authentication failed!
{hex(auth_tag)} != {hex(ghash)}")
        counter = Counter.new(
            nbits=32,
            prefix=long_to_bytes(init_value, 12),
            initial_value=2,
            allow_wraparound=True,
```

```
        )
        aes_ctr = AES.new(self.master_key, AES.MODE_CTR,
counter=counter)
        padded_ciphertext = self._pad_data(ciphertext)
        plaintext = aes_ctr.decrypt(padded_ciphertext)[:
len(ciphertext)]
        return plaintext


if __name__ == "__main__":
    key = get_random_bytes(16)
    plaintext = b"Hello, World!"
    aad = b"authentication data"
    iv = get_random_bytes(12)

    # Test Decrypt
    cipher = AES.new(key, AES.MODE_GCM,iv)
    cipher.update(aad)
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)
    nonce = cipher.nonce

    print("Cryptodome Encryption:")
    print(f"Nonce: {nonce.hex()}")
    print(f"Ciphertext: {ciphertext.hex()}")
    print(f"Tag: {tag.hex()}")

    try:
        cipher = AES_WITH_GCM(key)
        decrypted = cipher.decrypt(nonce, ciphertext, tag, aad)
        print("\nCustom Decryption:")
        print(f"Decrypted: {decrypted}")
        print(f"Decryption successful: {decrypted == plaintext}")
    except ValueError as e:
        print(f"Custom Decryption failed: {str(e)}")

    # Test Encrypt
    cipher = AES_WITH_GCM(key)
    custom_ciphertext, custom_tag = cipher.encrypt(iv, plaintext,
aad)

    print("\nCustom GCM Encryption:")
    print(f"Nonce: {iv.hex()}")
    print(f"Ciphertext: {custom_ciphertext.hex()}")
    print(f"Tag: {custom_tag}")
```

```
    try:
        cipher = AES.new(key, AES.MODE_GCM, nonce=iv)
        cipher.update(aad)
        decrypted = cipher.decrypt_and_verify(custom_ciphertext,
custom_tag.to_bytes(16, 'big'))
        print("\nCryptodome Decryption of Custom Encryption:")
        print(f"Decrypted: {decrypted}")
        print(f"Decryption successful: {decrypted == plaintext}")
    except ValueError as e:
        print(f"Cryptodome Decryption failed: {str(e)}")
```

Protocole

> Alice et Bob souhaitent sécuriser leur canal de communication. Ils vous engagent afin de leur proposer une solution. Décrivez la solution en apportant les justifications nécessaires et implémentez les fonctions des deux côtés pour la partie chiffrement / déchiffrement. Pensez également à une solution pour le partage de clés !

Justification de la solution

Pour sécuriser la communication entre Alice et Bob, nous allons utiliser le système de chiffrement asymétrique RSA (Rivest-Shamir-Adleman). Ce choix est justifié par les raisons suivantes :

Sécurité : RSA est largement utilisé et considéré comme sûr. Asymétrie : Il permet le chiffrement avec une clé publique et le déchiffrement avec une clé privée.

Génération des clés

Chaque partie (Alice et Bob) génère indépendamment sa paire de clés :

1. Choisir deux grands nombres premiers distincts, p et q.
2. Calculer n = p * q (le module).
3. Calculer φ(n) = (p-1) * (q-1) (l'indicatrice d'Euler).
4. Choisir un exposant public e, premier avec φ(n).
5. Calculer l'exposant privé d tel que d * e ≡ 1 (mod φ(n)).

La clé publique est (e, n), la clé privée est (d, n).

Chiffrement et Déchiffrement

Chiffrement : Pour chaque caractère m du message, calculer c = m^e mod n
Déchiffrement : Pour chaque caractère chiffré c, calculer m = c^d mod n

Partage de clés

Alice et Bob génèrent chacun leur paire de clés. Ils échangent leurs clés publiques respectives via un canal non sécurisé. Pour envoyer un message à Bob, Alice utilise la clé publique de Bob pour chiffrer. Bob utilise sa clé privée pour déchiffrer le message d'Alice.

```
import random

def miller_rabin(n, k=5):  # number of tests = k
    if n == 2 or n == 3:
        return True
    if n <= 1 or n % 2 == 0:
        return False

    # Write (n - 1) as 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

def generate_prime(bits):
    while True:
        num = random.getrandbits(bits)
        if num % 2 != 0 and miller_rabin(num):
            return num

def create_key_pair(bits):
    p = generate_prime(bits)
    q = generate_prime(bits)
```

```
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537
    d = pow(e, -1, phi)
    return ((e, n), (d, n))

def encrypt(public_key, message):
    e, n = public_key
    return [pow(ord(char), e, n) for char in message]

def decrypt(private_key, encrypted_message):
    d, n = private_key
    return ''.join([chr(pow(char, d, n)) for char in
encrypted_message])

# Génération des clés
bits = 1024
alice_public, alice_private = create_key_pair(bits)
bob_public, bob_private = create_key_pair(bits)

print("Clés d'Alice :")
print(f"Public : {alice_public}")
print(f"Privée : {alice_private}")

print("\nClés de Bob :")
print(f"Public : {bob_public}")
print(f"Privée : {bob_private}")

# Communication Alice -> Bob
alice_message = "Bonjour Bob?"
print("\nCommunication Alice -> Bob :")
print(f"Message original : {alice_message}")
encrypted_message = encrypt(bob_public, alice_message)
print(f"Message chiffré : {encrypted_message}")
decrypted_message = decrypt(bob_private, encrypted_message)
print(f"Message déchiffré : {decrypted_message}")

# Communication Bob -> Alice
bob_message = "Bonjour Alice"
print("\nCommunication Bob -> Alice :")
print(f"Message original : {bob_message}")
encrypted_message = encrypt(alice_public, bob_message)
print(f"Message chiffré : {encrypted_message}")
```

```
decrypted_message = decrypt(alice_private, encrypted_message)
print(f"Message déchiffré : {decrypted_message}")
```