Laboratorio di Sicurezza dell'Informazione M

Bumma Giuseppe

Contents

1 Pyhton Cryptography

- Cryptography is a Python package that provides cryptographic recipes and primitives to developers
- It includes both high-level recipes and low-level interfaces to common cryptographic algorithms
- You can install cryptography with:

```shell
pip install cryptography
```

N.B. on kali linux it should be preinstalled.

1.1 Fernet - The Recipes Layer

- It includes safe cryptographic recipes that require minimum choices
- Developers don’t make many decisions
- Implementation of symmetric authenticated cryptography
- It uses **AES** in **CBC** mode with 128-bit key for encryption and **PKCS7** padding
- It employs **HMAC** using **SHA256** for authentication
- Initialization vectors are generated using `os.urandom()`

1.2 The Fernet Class

This is how the Fernet Class should be used

Fernet usage Python

```
>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> f = Fernet(key)
>>> token = f.encrypt(b"my deep dark secret")
>>> token
b'...'
>>> f.decrypt(token)
b'my deep dark secret'
```

The key parameter must be kept safe since the encrypted message contains the current time when it was generated, and so the time message will be visible to possible attackers.
Indeed the current time can be extraced with `key.extract_timestamp(token)`.

1.2.1 Time-Based Security

With Fernet is possible to set a time expiration for the token created:

- `encrypt_at_time(token, current_time)` - Encrypts data with a specific timestamp
- `decrypt_at_time(token, ttl, current_time)` - Decrypts only if token hasn't exceeded its TTL (Time To Live)

Here's an example:

Time-Based Security - Generating token Python

```
from cryptography.fernet import Fernet
from datetime import datetime, timedelta

# Generate encryption key
key = Fernet.generate_key()
f = Fernet(key)

# Security code sent at 9:00 AM
send_time = datetime(2025, 5, 28, 9, 0, 0)  # 9:00 AM
secret_message = b"Your verification code is: 847291. Use within 5 minutes."

# Encrypt message with timestamp
encrypted_token = f.encrypt_at_time(
    secret_message,
    int(send_time.timestamp())
)

print(f"🔒 Message encrypted at: (send_time}")
print(f"📱 Encrypted token: (encrypted_token[:50]}...")
```

Time-Based Security - User decrypt the message Python

```python
# User tries to decrypt at 9:03 AM (3 minutes later)
user_access_time = datetime(2025, 5, 28, 9, 3, 0)  # 9:03 AM
ttl = 300  # 5 minutes = 300 seconds

try:
    # Decrypt the message
    decrypted_message = f.decrypt_at_time(
        encrypted_token,
        ttl=ttl,
        current_time=int(user_access_time.timestamp())
    )

    print(f"✅ SUCCESS at {user_access_time.strftime('%I:%M %p')}")
    print(f"📄 Message: {decrypted_message.decode()}")
    print(f"⏰ Message age: 3 minutes (within 5-minute limit)")

except Exception as e:
    print(f"❌ FAILED: {e}")
```

1.3 The MultiFernet Class

This is how MultiFernet should be used:

MultiFernet usage Python

```
>>> from cryptography.fernet import Fernet, MultiFernet
>>> key1 = Fernet(Fernet.generate_key())
>>> key2 = Fernet(Fernet.generate_key())
>>> f = MultiFernet([key1, key2])
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
```

The `MultiFernet` class extends `Fernet` allowing the management of multiple Fernet keys. This is paramount because, in any cryptographic system, keys should not be immortal. The principle of key rotation - periodically changing the keys used to encrypt data - is a cornerstone of good security practice. It serves to limit the potential damage if a key is compromised and to increase the difficulty of certain attacks.

This brings us to the `rotate` method. Imagine you have a corpus of data, all encrypted with a particular Fernet key. Perhaps this key is old, or worse, you suspect it may have been exposed – an employee departure, for instance, is a common trigger for such concerns. Simply starting to encrypt new data with a new key is insufficient; the old, potentially vulnerable data remains encrypted with the compromised key.

The `rotate` function elegantly addresses this. When you instantiate `MultiFernet`, you provide it with a list of Fernet objects (each initialized with a specific key). The key at the head of this list (the first one) is considered the primary key and is used for all new encryption operations.

Rotate usage Python

```
>>> from cryptography.fernet import Fernet, MultiFernet
>>> key1 = Fernet(Fernet.generate_key())
>>> key2 = Fernet(Fernet.generate_key())
>>> f = MultiFernet([key1, key2])
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
>>> key3 = Fernet(Fernet.generate_key())
>>> f2 = MultiFernet([key3, key1, key2])
>>> rotated = f2.rotate(token)
>>> f2.decrypt(rotated)
b'Secret message!'
```

When you call `rotate(token)` on a `MultiFernet` instance, the following occurs:

1. **Decryption:** `MultiFernet` will iterate through its list of keys and attempt to decrypt the `token`. It will use the key that successfully decrypts the token.
2. **Re-encryption:** Once decrypted, the plaintext data is then immediately re-encrypted using the primary key.
3. **Timestamp Preservation:** The original timestamp embedded within the Fernet token is preserved during this re-encryption process. This is important for maintaining the integrity of the token's age information.

The `rotate` method, therefore, allows you to systematically re-encrypt your existing data under a new, secure key without ever exposing the plaintext data during the transition.

1.4 Using Password with Fernet

To use passwords securely with Fernet, the password needs to be transformed into a strong cryptographic key. Doing this require the use of a Key Derivation Function (KDF).

A KDF takes a password (and other parameters) and derives a cryptographically strong key from it. This process is also known as *"key stretching"*.

Password management Python

```
>>> import base64
>>> import os
>>> from cryptography.fernet import Fernet
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
>>> password = b"password"
>>> salt = os.urandom(16)
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=1_200_000,
... )
>>> key = base64.urlsafe_b64encode(kdf.derive(password))
>>> f = Fernet(key)
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
```

This "script" shows the use of `PBKDF2HMAC` (Password-Based Key Derivation Function 2 with HMAC). Other common KDFs include Argon2id and Scrypt.

- **PBKDF2HMAC:** Applies a pseudorandom function (like HMAC-SHA256) to the password and salt repeatedly.
- **Argon2id:** The winner of the Password Hashing Competition (2015), designed to be resistant to various attacks, including those using GPUs. It's often recommended for new applications.
- **Scrypt:** Designed to be memory-hard, making it expensive for attackers to perform large-scale custom hardware attacks.

The `salt` need to be **retrievable** because he must be used again with the password to re-derive the same key for decryption.

The `iterations` variable represents the number of times the KDF repeatedly applies its internal hashing function. A higher iteration count implies that more computational resources will be required by an ideal hacker performing a brute force attack.

1.5 Symmetric Encryption

`Cipher` objects combine an algorithm such as AES with a mode like CBC or CTR. A simple example of encrypting and then decrypting content with AES is:

Example of Encrypting and Decrypting Python

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> key = os.urandom(32)
>>> iv = os.urandom(16)
>>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message") + encryptor.finalize()
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct) + decryptor.finalize()
b'a secret message'
```

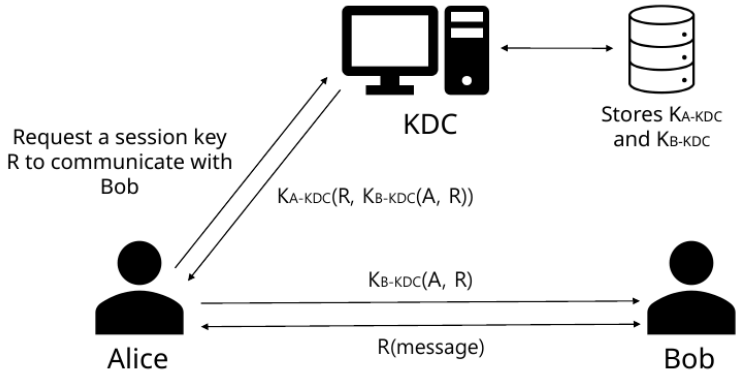
- `iv = os.urandom(16)`
  - This line generates a cryptographically secure random **Initialization Vector (IV)**.
  - 16 bytes are generated, which corresponds to a 128-bit IV.
  - This number is the same size of AES block size, which is 128 bit.
  - CBC mode requires an IV to ensure that encrypting the same plaintext block multiple times (even with the same key) results in different ciphertext blocks. This prevents patterns in the plaintext from being visible in the ciphertext.
- `ct = encryptor.update(b"a secret message") + encryptor.finalize()`
  - This line encrypts the plaintext message.
  - `encryptor.update(data)`: This method processes the data and returns any encrypted bytes that are ready.
  - `encryptor.finalize()`: This method finalizes the encryption process. It handles any remaining data in the buffer (e.g., by applying padding if necessary for modes like CBC) and returns the last part of the ciphertext.
  - The results of `update()` and `finalize()` are concatenated to form the complete ciphertext `ct`.

1.6 Exercise 1

- A Key Distribution Center (KDC) in cryptography is a system responsible for

providing keys to users in a network that shares sensitive or private data It is usually implemented as a server that shares symmetric keys with all registered users

- Suppose that Alice and Bob are two registered users, whose keys are respectively K A-KDC and KB-KDC
- Implement the key exchange protocol that allows Alice and Bob to securely communicate with each other
- Once the symmetric key has been retrieved, use it to securely exchange messages



This is how it should work:

1. Alice asks a key session to the KDC
2. The KDC replies with $K_{A-KDC}(R, K_{B-KDC}(A, R))$, where R is the session key
3. Alice decrypts the message, memorize the session key, then sends $K_{B-KDC}(A, R)$ to Bob
4. Bob decrypts the message and memorize the session key and the sender
5. Alice and Bob can securely exchange messages by encrypting them with the session key

1.6.1 App.py

User Creation and Registration Python

```python
# Generate secret keys for Alice and Bob
alice_key = Fernet.generate_key()
bob_key = Fernet.generate_key()

# Create user objects
alice = User("alice", alice_key)
bob = User("bob", bob_key)

# Register users with KDC
kdc.add_user_key(alice.name, alice_key)
kdc.add_user_key(bob.name, bob_key)
```

- Each user gets a unique secret key K_{A-KDC} and K_{B-KDC}
- These keys are registered with the KDC for future authentication

Session Key Request Python

```python
session_key, receiver_message = kdc.get_session_key(alice.name, bob.name)
```

The KDC's `get_session_key` method:

- Generates a fresh session key R
- Encrypts R with Bob's secret key: $K_{B-KDC}(R)$
- Encrypts R with Alice's secret key:$K_{A-KDC}(R)$
- Encrypts the message for Bob with Alice's key: $K_{A-KDC}(K_{B-KDC}(R))$
- Returns both encrypted session keys

Session Key Distribution

```python
alice.add_session_key(session_key, bob.name)
receiver_message = alice.decrypt(receiver_message)
bob.obtain_session_key(alice.name, receiver_message)
```

- Alice decrypts her session key using K_{A-KDC}
- Alice decrypts Bob's message and forwards it to Bob
- Bob decrypts his session key using K_{B-KDC}
- Both now share the same session key R

1.6.2 `kdc_server.py`

Let's break down the `get_session_key` method.

1.6.2.1 Generate Fresh Session Key

```python
session_key = Fernet.generate_key()
```

- **What:** Creates a new random symmetric key R
- **Why:** Each communication session gets a unique key for forward secrecy
- **Security:** Even if one session key is compromised, other sessions remain secure

1.6.2.2 Prepare Encryption Objects

```python
f_r = Fernet(self.registered_users[receiver])  # Bob's secret key
f_s = Fernet(self.registered_users[sender])    # Alice's secret key
```

- **What:** Creates Fernet cipher objects using each user's pre-shared secret key
- **Why:** The KDC needs to encrypt data that only specific users can decrypt

1.6.2.3 Create Message for Receiver (Bob)

```python
receiver_message = f_r.encrypt(session_key)
```

- **What:** Encrypts the session key with Bob's secret key $\longrightarrow K_{B-KDC}(R)$
- **Why:** Only Bob can decrypt this to get the session key
- **Important:** Alice cannot read this even though she'll receive it

1.6.2.4 Return Encrypted Data for Sender (Alice)

```python
return f_s.encrypt(session_key), f_s.encrypt(receiver_message)
```

This returns **two encrypted messages** for Alice:

- **First Return Value:** $K_{A-KDC}(R)$

  ```python
  f_s.encrypt(session_key)
  ```

  - **What:** Session key encrypted with Alice's secret key
  - **Purpose:** Allows Alice to decrypt and obtain the session key R
  - **Usage:** Alice uses this to get her copy of the session key
- **Second Return Value:** $K_{A-KDC}(K_B - KDC(R))$

  ```python
  f_s.encrypt(receiver_message)
  ```

- **What:** Nested encryption - Bob's encrypted session key, encrypted again with Alice's key
- **Purpose:** Secure delivery mechanism for Bob's session key
- **Critical Point:** Alice can decrypt the outer layer but cannot read the inner content

1.6.2.5 Why This Double Encryption?

1. **Authenticated Delivery:**
   - Bob knows the message came from the KDC (via Alice) because only Alice could have decrypted the outer layer
   - Prevents man-in-the-middle attacks where an attacker tries to send fake session keys to Bob
2. **Confidentiality from Alice:**
   - Alice cannot read Bob's session key even though she handles the delivery
   - The inner encryption $K_{B-KDC}(R)$ protects against Alice being malicious