

Contents

1	Pyhton Cryptography	1
1.1	Fernet - The Recipes Layer	1
1.2	The Fernet Class	1
1.2.1	Time-Based Security	1
1.3	The MultiFernet Class	2
1.4	Using Password with Fernet	4
1.5	Symmetric Encryption	4

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:

```
1 pip install cryptography
```

\$ Shell

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

```
1 >>> from cryptography.fernet import Fernet
2 >>> key = Fernet.generate_key()
3 >>> f = Fernet(key)
4 >>> token = f.encrypt(b"my deep dark secret")
5 >>> token
6 b'...'
7 >>> f.decrypt(token)
8 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

```
1  from cryptography.fernet import Fernet
2  from datetime import datetime, timedelta
3
4  # Generate encryption key
5  key = Fernet.generate_key()
6  f = Fernet(key)
7
8  # Security code sent at 9:00 AM
9  send_time = datetime(2025, 5, 28, 9, 0, 0) # 9:00 AM
10 secret_message = b"Your verification code is: 847291. Use within 5 minutes."
11
12 # Encrypt message with timestamp
13 encrypted_token = f.encrypt_at_time(
14     secret_message,
15     int(send_time.timestamp()))
16 )
17
18 print(f"🔒 Message encrypted at: {send_time}")
19 print(f"📱 Encrypted token: {encrypted_token[:50]}...")
```

Time-Based Security - User decrypt the message


 Python

```
1  # User tries to decrypt at 9:03 AM (3 minutes later)
2  user_access_time = datetime(2025, 5, 28, 9, 3, 0) # 9:03 AM
3  ttl = 300 # 5 minutes = 300 seconds
4
5  try:
6      # Decrypt the message
7      decrypted_message = f.decrypt_at_time(
8          encrypted_token,
9          ttl=ttl,
10         current_time=int(user_access_time.timestamp()))
11 )
12
13 print(f"✅ SUCCESS at {user_access_time.strftime('%I:%M %p')}")
14 print(f"📄 Message: {decrypted_message.decode()}")
15 print(f"🕒 Message age: 3 minutes (within 5-minute limit)")
16
17 except Exception as e:
18     print(f"❌ FAILED: {e}")
```

1.3 The MultiFernet Class

This is how MultiFernet should be used:

MultiFernet usage

 Python


```
1 >>> from cryptography.fernet import Fernet, MultiFernet
2 >>> key1 = Fernet(Fernet.generate_key())
3 >>> key2 = Fernet(Fernet.generate_key())
4 >>> f = MultiFernet([key1, key2])
5 >>> token = f.encrypt(b"Secret message!")
6 >>> token
7 b'...'
8 >>> f.decrypt(token)
9 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

```
1 >>> from cryptography.fernet import Fernet, MultiFernet
2 >>> key1 = Fernet(Fernet.generate_key())
3 >>> key2 = Fernet(Fernet.generate_key())
4 >>> f = MultiFernet([key1, key2])
5 >>> token = f.encrypt(b"Secret message!")
6 >>> token
7 b'...'
8 >>> f.decrypt(token)
9 b'Secret message!'
10 >>> key3 = Fernet(Fernet.generate_key())
11 >>> f2 = MultiFernet([key3, key1, key2])
12 >>> rotated = f2.rotate(token)
13 >>> f2.decrypt(rotated)
14 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 requires 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
1	>>> import base64	
2	>>> import os	
3	>>> from cryptography.fernet import Fernet	
4	>>> from cryptography.hazmat.primitives import hashes	
5	>>> from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC	
6	>>> password = b"password"	
7	>>> salt = os.urandom(16)	
8	>>> kdf = PBKDF2HMAC(
9	... algorithm=hashes.SHA256(),	
10	... length=32,	
11	... salt=salt,	
12	... iterations=1_200_000,	
13	...)	
14	>>> key = base64.urlsafe_b64encode(kdf.derive(password))	
15	>>> f = Fernet(key)	
16	>>> token = f.encrypt(b"Secret message!")	
17	>>> token	
18	b'...'	
19	>>> f.decrypt(token)	
20	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.

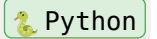
The salt needs to be **retrievable** because it must be used again with the password to re-derive the same key for decryption.

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



```
1  >>> import os
2  >>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
3  >>> key = os.urandom(32)
4  >>> iv = os.urandom(16)
5  >>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
6  >>> encryptor = cipher.encryptor()
7  >>> ct = encryptor.update(b"a secret message") + encryptor.finalize()
8  >>> decryptor = cipher.decryptor()
9  >>> decryptor.update(ct) + decryptor.finalize()
10 b'a secret message'
```