# Cryptographie

Avant de mettre en place des mécanismes de protection, il est essentiel de comprendre les objectifs fondamentaux qu'elle poursuit:
ils sont les suivants:

- Confidentialité : empêcher un tiers de lire les données.
- Authenticité : garantir l'identité de l'expéditeur.
- Intégrité : vérifier que les données n'ont pas été modifiées.
- Non-répudiation : prouver qu'un message a bien été envoyé par une personne.

Commençons par un peu de théorie...

## 1- Le chiffrement symétrique

Le chiffrement symétrique repose sur une clé secrète partagée entre l'expéditeur et le destinataire.

1. L'émetteur choisit une clé secrète.
2. Il chiffre le message à l'aide d'un algorithme et de cette clé.
3. Il transmet le message chiffré (chiffre).
4. Le récepteur, qui connaît la même clé, déchiffre le message.

La clé doit être échangée de manière sécurisée, sinon un attaquant peut intercepter le message et la clé.

Exemple avec Fernet (AES + clé générée automatiquement) :

```python
from cryptography.fernet import Fernet

# Génération et partage de la clé
cle = Fernet.generate_key()
cipher = Fernet(cle)

# Chiffrement
message = "Bonjour".encode()
message_chiffre = cipher.encrypt(message)
print("Chiffré :", message_chiffre)

# Déchiffrement
message_dechiffre = cipher.decrypt(message_chiffre)
print("Déchiffré :", message_dechiffre.decode())
```

## 2- Le chiffrement asymétrique

Le chiffrement asymétrique (ou cryptographie à clé publique) repose sur deux clés différentes :

- une clé publique (connue de tous),
- une clé privée (gardée secrète).

Pour chiffrer un message on utilise la clé publique du destinataire et pour déchiffrer le message on utilise la clé privée du destinataire.
Ce système est asymétrique car les clés sont différentes mais liées mathématiquement.

1 Bob crée
- une clé publique
- une clé privé

2 Bob publie sa clé publique qui est maintenant accessible à tous

3 Alice récupère la clé publique de Bob car elle veut lui envoyer un message

4 Alice chiffre son message avec la clé publique de Bob

5 Alice envoie son message chiffré à Bob

6 Bob déchiffre le message avec sa clé privé

```python
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes

# Génération des clés
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()

# Chiffrement avec la clé publique
message = b"Message secret"
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)

# Déchiffrement avec la clé privée
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
)

print("Déchiffré :", plaintext.decode())

```

## 3- Le CRC

Le CRC est une méthode de détection d'erreurs utilisée lors de la transmission de données.
Il ne chiffre pas, ne protège pas contre l'espionnage, mais permet de vérifier que les données n'ont pas été altérées (par une erreur de transmission ou un défaut de stockage).

Principe:

1. Le message est traité comme une longue suite de bits.
2. On le divise (modulo 2) par un polynôme générateur (représenté lui aussi par une suite de bits).
3. Le reste de cette division (appelé le CRC) est ajouté à la fin du message.
4. Le récepteur refait le calcul :
    - Si le nouveau reste est zéro, alors le message est valide (a priori non corrompu).
    - Sinon, il y a eu une erreur de transmission.

```python
import zlib

message = b"Bonjour"
crc = zlib.crc32(message)
print("CRC :", hex(crc))

```

Le CRC ne garantit pas la sécurité des données (pas de clé, pas de chiffrement).
Il permet uniquement de détecter certaines erreurs accidentelles (changement d'un ou plusieurs bits).
Il est utilisé dans :
les protocoles réseau (Ethernet, TCP/IP),
les fichiers compressés (ZIP),
les CD/DVD, disques durs, etc.

## 3- Fonctions de hachage

Une fonction de hachage transforme une donnée en une empreinte numérique (ou condensat) de taille fixe. Elle possède plusieurs propriétés essentielles :

- Non-réversibilité : il est impossible de retrouver le message original à partir de l'empreinte.
- Unicité : deux messages différents ont très peu de chances d'avoir la même empreinte (collision).
- Sensibilité aux modifications : une petite modification du message entraîne une empreinte complètement différente.

Concrètement, comment cela fonctionne ?
Lorsqu'on applique une fonction de hachage à un message, elle parcourt le contenu (souvent octet par octet) et applique une série d'opérations mathématiques et logiques pour produire une empreinte. Cette empreinte est unique pour chaque message (avec une probabilité très élevée).
Elle est utilisée pour :

- Vérifier l'intégrité (ex : comparer deux hachages).
- Stocker des mots de passe sous forme sécurisée (seule l'empreinte est enregistrée).
- Signatures numériques (hachage du message signé).

```python
import hashlib

message = b"Bonjour"
hash_obj = hashlib.sha256(message)
print("SHA-256 :", hash_obj.hexdigest())

```

## 4. L'intégrité avec HMAC

HMAC est une méthode utilisée pour garantir l'intégrité et l'authenticité d'un message.
Il ne chiffre pas les données comme AES, mais il génère un code d'authentification qui permet de vérifier qu'un message n'a pas été altéré.

**Principe de fonctionnement  HMAC ?**
HMAC utilise une fonction de hachage cryptographique (comme SHA-256) combinée à une clé secrète pour produire une signature unique du message.

1. **Préparation de la clé**
   - Si la clé est trop longue, elle est d'abord hachée.
   - Si elle est trop courte, elle est complétée avec des zéros.
2. **Concaténation et hachage**
   - La clé est mélangée avec un premier masque (opad) et combinée avec le message.
   - Le tout est ensuite haché.
   - Le résultat est mélangé avec un second masque (ipad) et re-haché.
3. **Génération du code HMAC**
   - Le résultat final est un code court et unique qui permet de vérifier l'authenticité du message.

**Vérification d'un HMAC**
Pour vérifier qu'un message n'a pas été modifié :

- Le destinataire reproduit le calcul HMAC avec la même clé.
- Si le HMAC généré correspond à celui reçu, le message est authentique.

```
5   import hmac
6   import hashlib
7
8   message = b"Message secret"
9   cle = b"ma_cle"
0   mac = hmac.new(cle, message, hashlib.sha256)
1   print("HMAC :", mac.hexdigest())
2
```

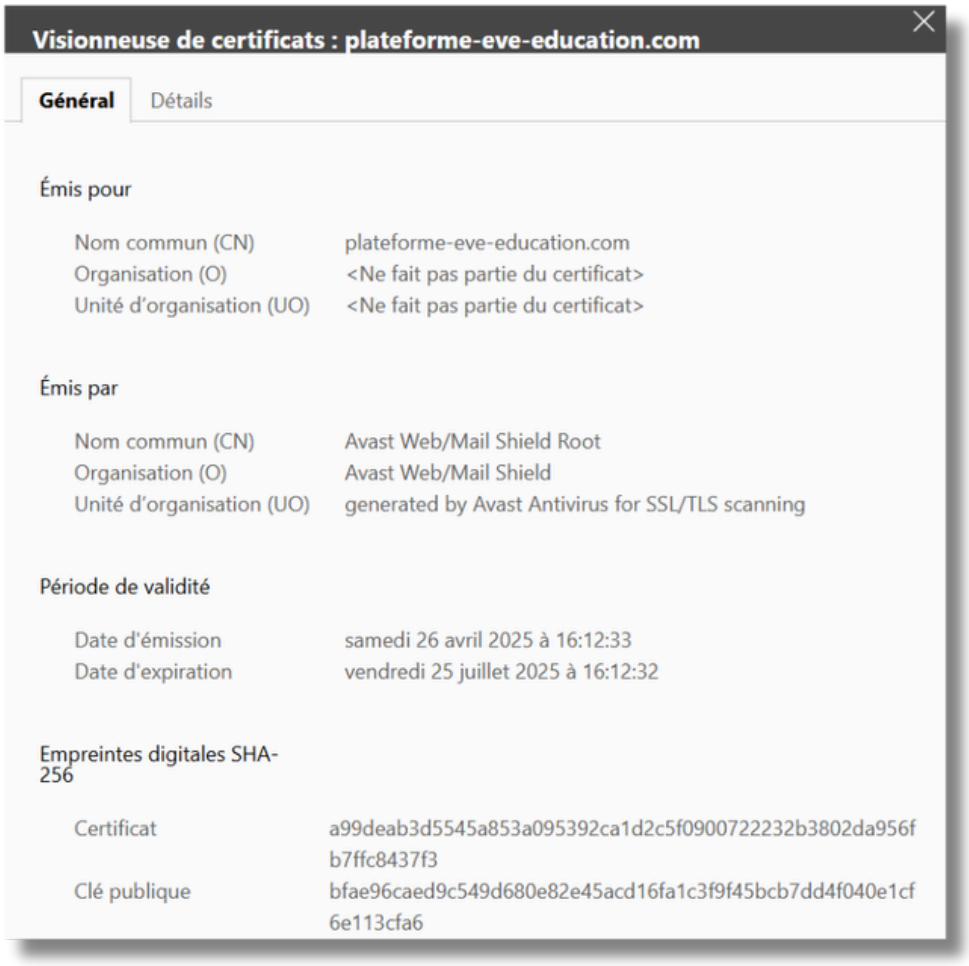
## 5. HTTPS

HTTPS (HyperText Transfer Protocol Secure) est une version sécurisée du protocole HTTP.
Il repose sur le protocole TLS (Transport Layer Security) pour chiffrer les communications entre le navigateur et le serveur web.

**Principes de fonctionnement:**
Le serveur web possède une paire de clés (publique/privée) et un certificat numérique (souvent signé par une autorité de certification).
Lorsqu'un client se connecte via HTTPS, il reçoit ce certificat.
Le navigateur vérifie la validité du certificat (chaîne de confiance).
Ensuite, le client chiffre une clé de session avec la clé publique du serveur.
Cette clé est utilisée pour le chiffrement symétrique des échanges (plus rapide).
Ainsi, HTTPS garantit :

- La confidentialité (données chiffrées)
- L'authenticité du serveur (certificat)
- L'intégrité des données transmises

Tu peux visualiser le certificat en cliquant sur le cadenas dans l'url.



## 6. Simulation

### Chiffrement symétrique (XOR)

Ouvre le fichier crypto.py dans ton IDE et analysons le:

```
##### fonction qui récupère le code ASCII #####
def get_utf8(texte) :
    """
    Renvoie le tableau des codes ASCII/UTF8 de chaque caractère du message
    Entrée : texte (type str)
    Sortie : un tableau (type list) d'entiers
    """
    liste_ascii=[]
    for i in texte:
        liste_ascii.append(ord(i))
    return liste_ascii

assert get_utf8("NSI") == [78, 83, 73]
```

La fonction *get_utf8()* prend une chaine de caractères en argument et renvoie un tableau contenant les codes ASCII de chaque caractère. Elle utilise pour ça la fonction *ord()* qui renvoie le code ASCII d'un caractère.

```
##### fonction qui récupère les caractères #####
def get_string(tab_utf8) :
    """
    Renvoie la chaine de caractères dont les codes ASCII/UTF8 sont les valeurs de tab_utf8
    Entrée : un tableau (type list) d'entiers
    Sortie : une chaine de caractères (type str)
    """
    chaine=""
    for i in tab_utf8:
        chaine+=chr(i)
    return chaine

assert get_string([78, 83, 73]) == "NSI"
```

La fonction *get_string()* fait l'opération inverse. Elle reçoit en argument une liste d'entier et renvoie les caractères dont les codes ASCII correspondent à ces entiers. Elle utilise pour ça la fonction chr*()* qui renvoie le caractère dont le code ASCII est donné.

```
##### fonction qui réalise un XOR entre 2 tableaux #####
def chiffre_XOR(texte, clef) :
    """ Renvoie le tableau d'entiers obtenu par chiffrement XOR de texte avec clef comme clé de  chiffrement
    Entrées : texte et clef sont de type str
    Sortie : un tableau (type list) d'entiers
    """
    tab_utf8_texte = get_utf8(texte)
    tab_utf8_clef = get_utf8(clef)
    tab_xor = []
    for k in range(len(texte)) :
        nb_xor = tab_utf8_texte[k] ^ tab_utf8_clef[k%len(clef)]
        tab_xor.append(nb_xor)
    return tab_xor

assert chiffre_XOR("UN MESSAGE TRÈS SECRET", "NSI") == \
       [27, 29, 105, 3, 22, 26, 29, 18, 14, 11, 115, 29, 28, 155, 26, 110, 0, 12, 13, 1, 12, 26]

print(chiffre_XOR("UN MESSAGE TRÈS SECRET", "NSI"))
```

La fonction chiffre_XOR chiffre un message à l'aide d'une clé en utilisant le chiffrement XOR:

- Elle prend en argument un texte et une clé (tous les 2 de type string)
- Elle convertit ces 2 String en nombre entier en utilisant la fonction get_utf8()
- Elle effectue l'opération XOR (opérateur ^)  entre chaque entier du message et chaque entier de la clé ( la clé est répétée si elle est plus petite que le message (k%len(clef))
- Le résultat de l'opération XOR est stocké dans un tableau: c'est notre message chiffré

```
def dechiffre_XOR(tab_crypte, clef) :
    """ Renvoie le texte obtenu par déchiffrement du tableau tab_crypté avec clef comme clé de  chiffrement
    Entrées :
        tab_crypte est un tableau d'entiers
        clef est de type str
    Sortie : une chaine de caractères
    """
    chaine=""
    tab_utf8_clef = get_utf8(clef)
    tab_xor = []
    for k in range(len(tab_crypte)) :
        nb_xor = tab_crypte[k] ^ tab_utf8_clef[k%len(clef)]
        tab_xor.append(nb_xor)
    chaine=get_string(tab_xor)
    return chaine

assert dechiffre_XOR([27, 29, 105, 3, 22, 26, 29, 18, 14, 11, 115, 29, 28, 155, 26, 110, 0, 12, 13, 1, 12, 26], "NSI") \
       == "UN MESSAGE TRÈS SECRET"


print(dechiffre_XOR([27, 29, 105, 3, 22, 26, 29, 18, 14, 11, 115, 29, 28, 155, 26, 110, 0, 12, 13, 1, 12, 26], "NSI"))
```

La fonction dechiffre_XOR déchiffre un message à l'aide d'une clé en utilisant le chiffrement XOR:

- Elle prend en argument un tableau contenant notre message chiffré et une clé 2 de type string
- Elle convertit la clé en nombre entier en utilisant la fonction get_utf8()
- Elle effectue l'opération XOR (opérateur ^)  entre chaque entier du message et chaque entier de la clé ( la clé est répétée si elle est plus petite que le message (k%len(clef))
- Le résultat de l'opération XOR est convertit en string à l'aide de la fonction get_string(): c'est notre message déchiffré

**Chiffrement asymétrique**

Notre situation est la suivante:

- Alice génère une clé privée et une clé publique
- Bob utilise la clé publique d'Alice pour crypter son message.
- Bob envoie son message crypté à Alice.
- Alice décrypte le message reçu avec sa clé privée

*crypto_alice.py*

```python
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes

# Génération de la clé privée
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=4096
)
# Génération de la clé publique
public_key = private_key.public_key()

# Sérialisation de la clé privée
private_pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
)

# Sérialisation de la clé publique
public_pem = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Affichage des clés
print("Clé privée :\n", private_pem.decode())
print("\nClé publique :\n", public_pem.decode())

# écriture du fichier contenant la clé publique
with open("public_key.pem", "wb") as g:
    g.write(public_pem)

# écriture du fichier contenant la clé privée
with open("private_key.pem", "wb") as g:
    g.write(private_pem)
```

La méthode *generate_private_key()* de la bibliothèque *rsa*, permet de générer une clé privée.
En peut en déduire une clé publique en appelant la méthode *public_key()* sur cette clé privée.
On doit ensuite sérialiser ces clés pour les stocker dans les fichiers *public_key.pem* et *private_key.pem.*

*crypto_bob.py*

```python
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
import sys

# Message à chiffrer
message = b"Je suis Bob, comment ca va ?"

# Récupération de la clé publique
with open("public_key.pem", "rb") as g:
    key = serialization.load_pem_public_key(g.read())

# Chiffrement avec la clé publique
ciphertext = key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Affichage du message chiffré
print("\nMessage chiffré :", ciphertext.hex())

# Ecriture du message chiffré dans un fichier
with open("datatext.bin", "wb") as f:
    f.write(ciphertext)
```

La méthode *load_pem_public_key()* permet de récupérer la clé publique dans le fichier *public_key.pem*.
La méthode *encrypt()* permet ensuite de chiffrer le message en *SHA_256*.
Le message chiffré est enfin stocké dans un fichier pour être transmis.

*crypto-alice2.py*

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes

# Récupération clé privée dans le fichier
with open("private_key.pem", "rb") as f:
    key = serialization.load_pem_private_key(
        f.read(),
        password=None  # Mettre un mot de passe si la clé est protégée
    )

# Récupération du message chiffré
with open("datatext.bin", "rb") as f:
    ciphertext = f.read()

# Déchiffrement du message avec la clé privée
decrypted_message = key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Affichage du message déchiffré
print("\nMessage déchiffré :", decrypted_message.decode())
```

Tout d'abord , la clé privée est récupérée dans le fichier  private_key.pem.
On récupère ensuite le message chiffré dans le fichier datatext.bin.
On déchiffre enfin le message avec la méthode decrypt()