# TD4 - Chiffrement par Blocs

Responsable : M. Bombar

## 1 Attaque par Vaudenay sur le padding dans le mode CBC

### 1.1 Introduction

Un système de chiffrement par blocs $(E_K, D_K)$ opère sur des blocs de taille fixée $\ell$ (par exemple 128 bits, ou encore 16 octets pour l'AES). Ainsi, le message à chiffrer doit nécessairement être un multiple de $\ell$. Rappelons le fonctionnement du mode CBC (pour Cipher Block Chaining).

**Mode CBC** Un texte clair de taille $n\ell$ bits est découpé en $n$ blocks de $\ell$ bits $(p_1, \ldots, p_n)$, et le chiffré correspondant va être de la forme $(v, c_1, \ldots, c_n)$ où

- $v \in \mathbb{F}_2^\ell$ est un vecteur d'initialisation, tiré uniformément.
- $c_1 = E_K(p_1 + v)$ (où l'addition est bien sûr celle de $\mathbb{F}_2^\ell$)
- $c_i = E_K(p_i + c_{i-1})$.

(Q1) Rappelez comment fonctionne le déchiffrement en mode CBC.

En pratique, les messages pouvant être de taille arbitraire, on va alors utiliser un système de *padding* pour transformer tout message en un message de longueur multiple de $\ell$. Plus précisément, un texte clair va être découpé en blocs de taille $\ell$, sauf éventuellement le dernier bloc qui sera de taille $i \leq \ell$, et on complète ce dernier bloc selon un processus standardisé afin de le rendre de taille $\ell$. Il existe beaucoup de systèmes de paddings possibles, mais un système couramment utilisé en pratique est le padding **PKCS#7** (pour **Public Key Cryptography Standard**). C'est par exemple le système utilisé dans la suite SSL/TLS depuis SSLv3 and TLS1.0. Voici son fonctionnement dans le cas de blocs de taille $\ell = 16$ octets (ce qu'on va toujours supposer par la suite) :

**Padding PKCS#7** Notons $(\mu_1, \ldots, \mu_i)$ les $i$ octets du dernier bloc.[1] Ce processus de padding consiste alors à concaténer $16 - i$ octets représentant l'entier $16 - i$. Par exemple,

1. En pratique, un bloc est forcément un multiple de 8 puisqu'en général l'octet est la plus petite unité de mémoire addressable, sur la plupart des architectures matérielles.

si le dernier bloc du texte clair est formé de 12 octets $(\mu_1, \ldots, \mu_{12})$, le bloc à chiffrer sera alors $(\mu_1, \ldots, \mu_{12}, 0\text{x}04, 0\text{x}04, 0\text{x}04, 0\text{x}04)$. [2]

Si la longueur originale du message était déjà un multiple de 16 octets, alors le dernier bloc n'a pas besoin d'être complété et on ajoute simplement un bloc supplémentaire contenant 16 fois l'octet 0x10 (qui représente donc l'entier 16).

Pour retirer le padding sur le message une fois déchiffré, il suffit donc de lire le dernier octet $\mu$ comme un entier entre 0 et 16, et de supprimer le nombre d'octets correspondant.

**Vulnérabilité de ce processus de padding** Ce processus de padding est très simple. C'est pourquoi il a été très vite adopté aux débuts du chiffrement sur internet (notamment dans les protocoles SSLv3 puis TLS1.0). Cependant, en 2002, Serge Vaudenay a montré qu'il était vulnérable à une attaque extrêment simple lorsqu'il est utilisé dans le mode CBC [Vau02]. L'objectif de cet exercice est de vous faire découvrir et implémenter l'attaque. Il s'agit de l'une des seules attaques par **canaux auxiliaires** que nous verront dans ce cours. Plus précisément, on suppose que l'attaquant a accès à un oracle qui prend en entrée un chiffré, et détermine sur le message clair associé a été formé via ce processus de padding. Cela pourrait sembler fort, mais en réalité de nombreux protocoles vont renvoyer un message d'erreur (ou terminer la communication) si le message n'est pas correctement formaté, et donc obtenir un tel oracle en pratique est possible. Aujourd'hui, dans les versions TLS1.2 puis TLS1.3, on utilise plutôt le mode GCM (Galois Counter Mode) afin de résister à cette attaque.

## 1.2 Mise en place

Pour vous montrer que cette attaque est extrêmement réaliste, on va utiliser une véritable implémentation du chiffrement AES, celle de la bibliothèque `PyCryptodome` qu'on a déjà utilisée au premier TD [3].

**Chaîne d'octets en Python** La bibliothèque `PyCryptodome` manipule des objets sous forme de chaînes d'octets (de type `bytes`), et pas de chaînes de caractères (`str`). En Python, si `maChaine` est de type `str`, vous pouvez obtenir la chaîne d'octets correspondante par `maChaine.encode()`. Réciproquement, si `myByteString` est une chaine d'octets interprétables comme des caractères `UTF-8`, alors `myByteString.decode()` renvoie la `str` correspondante.

```
In [1]: maChaine = "bonjour"

In [2]: type(maChaine)
Out[2]: str
```

2. Si vous n'avez jamais rencontré cette notation très courante en informatique, le préfixe 0x signifie que le nombre qui suit est écrit en hexadécimal : $0\text{x}00 = 0, 0\text{x}01 = 1, \ldots, 0\text{x}0\text{A} = 10, \ldots, 0\text{x}0\text{F} = 15, \ldots, 0\text{xFF} = 255$.

3. Vous trouverez la documentation de cette bibliothèque sur `https://www.pycryptodome.org/`.

```
In [3]: maChaine.encode()
Out[3]: b'bonjour'

In [4]: myByteString = b'Hello'

In [5]: type(myByteString)
Out[5]: bytes

In [6]: myByteString.decode()
Out[6]: 'Hello'

In [7]: myByteString[0] = b'a'
TypeError: 'bytes' object does not support item assignment

In [8]: myByteArray = bytearray(myByteString)
```

Un objet de type `bytes` se manipule de la même façon qu'une `str`. En particulier, on peut facilement accéder à un octet précis, mais elle n'est pas modifiable. On doit pour cela passer par un objet de type `bytearray`.

```
In [7]: myByteString[0]
Out[7]: 72 # C'est un entier entre 0 et 255.

In[8]: myByteString[0] = 98
TypeError: 'bytes' object does not support item assignment

In [9]: myByteArray = bytearray(myByteString)

In [10]: myByteArray[0] = 98

In [11]: myByteArray
Out[11]: bytearray(b'bello')

In [12]: myByteString = b'' + myByteArray

In [13]: myByteString
Out[13]: b'bello'
```

(Q2) À l'aide de la documentation de `PyCryptodome`, implémentez en Python une fonction `aes_cbc_encrypt(plaintext, key)` qui prend en entrées un texte clair (de type `bytes`) et une clé, et renvoie le chiffré correspondant, en mode CBC, avec le padding PKCS7 défini plus haut, et en utilisant un vecteur d'initialisation aléatoire qu'on pourra générer à l'aide de la commande `os.urandom(BLOCK_SIZE)` (où `BLOCK_SIZE` est la taille, en octets, d'un bloc (donc ici 16)).

(Q3) De même, implémentez une fonction `aes_cbc_decrypt(ciphertext, key)`, qui prend en entrées un chiffré obtenu avec AES-CBC et une clé secrète, et renvoie le texte clair correspondant, ou bien lève une exception `ValueError` si le padding est incorrect.

(Q4) En déduire une façon d'implémenter l'oracle à l'aide d'une fonction `paddingOracle(ciphertext, key)` qui renvoie `False` si le message clair associé a un padding incorrect, et `True` dans le cas contraire.

**Remarque 1.** *Bien entendu, pour le moment cet oracle prend en entrée la clé secrète, mais sera extrêmement utile pour tester votre attaque localement.*

## 1.3 L'attaque de Vaudenay

On suppose que le chiffré est de la forme $\mathbf{c} \overset{\text{def}}{=} (c_0, c_1, \ldots, c_n) \in (\mathbb{F}_2^{128})^{n+1}$, où chacun des blocs est formé de 16 octets et $c_0 \in \mathbb{F}_2^{128}$ est le vecteur d'initialisation, aléatoire. On note de même $\mathbf{p} \overset{\text{def}}{=} (p_1, \ldots, p_n) \in (\mathbb{F}_2^{128})^n$ le clair correspondant (le padding a déjà été effectué, donc chacun des $p_i$ contient aussi 16 octets).

Soit $s \in \mathbb{F}_2^{128}$ un vecteur de 16 octets, et considérons le chiffré altéré $\widetilde{\mathbf{c}} \in (\mathbb{F}_2^{128})^n$ tel que

$$\widetilde{c}_i = \begin{cases} c_i + s & \text{si } i = n-1 \\ c_i & \text{sinon.} \end{cases}$$

(Q5) Montrez qu'en appliquant l'algorithme de déchiffrement à $\widetilde{\mathbf{c}}$ on obtient un texte clair de la forme $\widetilde{\mathbf{p}} = (p_1, \ldots, p_{n-1}, p_n + s)$.

**Remarque 2.** *Cette propriété est inhérente au mode d'opération CBC, et pas au chiffrement par blocs sous-jacent utilisé. En particulier, on n'utilise pas du tout la structure du chiffrement `AES`.*

(Q6) En déduire un algorithme qui altère le chiffré, et permet à l'aide de l'oracle de retrouver la valeur utilisée dans le padding.
*Indice :* Faites un dessin !

(Q7) Implémentez une fonction `getPaddingValue(ciphertext, key)` qui renvoie la position du premier octet correspondant au padding dans le dernier bloc. Par exemple, si le dernier bloc du texte clair est de la forme

$$(\mu_1, \ldots, \mu_{11}, 0\text{x}05, 0\text{x}05, 0\text{x}05, 0\text{x}05, 0\text{x}05),$$

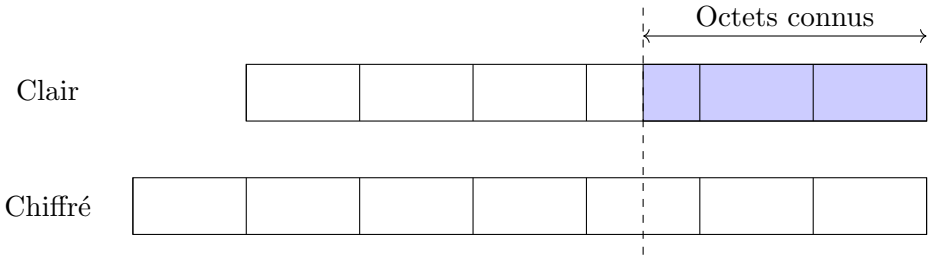
alors `getPaddingValue(ciphertext, key)` renvoie 5.

À présent qu'on connaît la valeur $a$ du padding (et donc la position du dernier octet de clair pertinent), on va de nouveau altérer le chiffré pour que le padding corresponde à la valeur $a + 1$, et on va en déduire la valeur du dernier octet du texte clair. Dans l'exemple précédent, on veut déterminer la valeur de $\mu_{11}$. On va donc faire en sorte que le dernier bloc du chiffré corresponde à un message de la forme

$$(\mu_1, \ldots, \mu_{10}, 0\text{x}06, 0\text{x}06, 0\text{x}06, 0\text{x}06, 0\text{x}06, 0\text{x}06)$$

(Q8) En déduire un algorithme pour déterminer le dernier octet du message clair.

Afin de déterminer le message clair complet, on va procéder de proche en proche en remontant tous les blocs petit à petit. Dans le cas général, on connaît les bits les plus à droite dans le texte clair, et on cherche à déterminer un bit de plus.




(Q9) Implémentez l'attaque complète afin de retrouver le message clair à partir du chiffré, sans utiliser la clé secrète autrement que pour faire des appels à l'oracle. Faites des tests pour vérifier que votre attaque fonctionne.

## 2 Challenge

Le but de ce challenge est de mettre en oeuvre l'attaque de Vaudenay en conditions réelles. J'ai implémenté un petit serveur qui écoute sur le port 5000 de la machine Hagrid du CREMI. Vous pouvez effectuer deux types de requêtes HTTP en POST :

— Sur l'endpoint `/get_ciphertext` vous allez pouvoir récupérer votre chiffré.

— Sur l'endpoint `/is_padding_ok` vous allez pouvoir tester si le padding est correct ou non (dans le cadre de l'attaque de Vaudenay).

Pour vous faciliter la tâche, je vous ai aussi implémenté un `client.py` qui fait lui même les requêtes au serveur à travers les deux fonctions suivantes :

— `get_ciphertext(name)` qui prend en entrées un nom de la forme `"Prenom_Nom"` et contacte le serveur pour vous attribuer un chiffré personalisé.

— `is_padding_ok(ciphertext, name)` qui prend en entrées une chaîne d'octets et implémente l'oracle de padding correspondant à votre chiffré personnalisé en contactant le serveur sur l'endpoint précité..

**Remarque 3.** *Vous pouvez choisir n'importe quel pseudo, nom ou autre tant qu'il est de la forme `"Votre_Pseudo"`. Simplement, gardez le même nom pour générer le chiffré et pour le padding.*

(Q10) Implémentez l'attaque complète dans les conditions du réel, pour retrouver votre message clair attribué, et en faisant appel à mon oracle de padding. Si votre clair est un objet de type `bytes`, rappelez-vous que vous pouvez retrouver une chaîne de caractère en utilisant la méthode `'.decode()'`.

(Q11) Lorsque vous m'enverrez la solution, mettez la dans un fichier qui contient deux lignes : `"Votre_Pseudo"` et le flag. Vous m'enverrez aussi le fichier de l'attaque.

## Références

[Vau02] Serge Vaudenay. « Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... » In : *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*. Sous la dir. de Lars R. Knudsen. T. 2332. Lecture Notes in Computer Science. https://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf. Springer, 2002, p. 534-546.