

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 ℓ (par exemple 128 bits, ou encore 16 octets pour l'AES). Ainsi, le message à chiffrer doit nécessairement être un multiple de ℓ . Rappelons le fonctionnement du mode CBC (pour Cipher Block Chaining).

Mode CBC Un texte clair de taille $n\ell$ bits est découpé en n blocs de ℓ bits (p_1, \dots, p_n) , et le chiffré correspondant va être de la forme (v, c_1, \dots, 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^ℓ)
- $c_i = E_K(p_i + c_{i-1})$.

(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 ℓ . Plus précisément, un texte clair va être découpé en blocs de taille ℓ , 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 ℓ . 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 et 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 (μ_1, \dots, μ_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 adressable, sur la plupart des architectures matérielles.

si le dernier bloc du texte clair est formé de 12 octets (μ_1, \dots, μ_{12}) , le bloc à chiffrer sera alors $(\mu_1, \dots, \mu_{12}, 0x04, 0x04, 0x04, 0x04)$.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 μ 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 extrêmement 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, mais pour vous éviter de passer du temps à l'implémenter vous-mêmes en Python, nous allons utiliser l'implémentation disponible dans la bibliothèque PyCryptodome. Celle-ci est disponible librement sur Github : <https://github.com/Legrandin/pycryptodome/tree/master> et activement maintenue. Vous trouverez la documentation de cette bibliothèque sur <https://www.pycryptodome.org/>.

Au CREMI Sur les machines du CREMI, PyCryptodome est normalement déjà installé. Vous pouvez le vérifier en ouvrant un shell Python ou iPython et en exécutant la commande suivante. Si elle fonctionne sans erreur, c'est que la bibliothèque est bien installée.

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 : $0x00 = 0, 0x01 = 1, \dots, 0x0A = 10, \dots, 0x0F = 15, \dots, 0xFF = 255$.

```
mbombar@hagrid $ ipython3
Python 3.11.2 (main, May 2 2024, 11:59:08) [GCC 12.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from Cryptodome.Cipher import AES
```

Installation en locale Si vous souhaitez l'installer sur votre machine personnelle, vous pouvez installer la version de votre distribution, ou bien l'installer avec le gestionnaire de paquet Python pip. Dans ce dernier cas, je vous incite très fortement à utiliser un environnement virtuel afin de bien séparer les dépendances liées à ce TP. Par exemple sur ma machine, en exécutant les commandes suivantes, je crée un dossier venv qui va contenir les dépendances de ce TP sans affecter le reste de mon système. Je vous renvoie à la documentation pour plus d'information <https://docs.python.org/3/library/venv.html>.

```
(21:32:14) (~/.Teaching/cryptanalysis/TD/TD4/Sage/
- -> python3 -m venv venv
(21:32:20) (~/.Teaching/cryptanalysis/TD/TD4/Sage/
- -> source venv/bin/activate
(21:32:22) (~/.Teaching/cryptanalysis/TD/TD4/Sage/
- (venv:TestAttack)--> pip install pycryptodome
Collecting pycryptodome
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.manylinux_2_17_aarch64.manylinux2014_aarch64.whl (3.4 kB)
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_aarch64.manylinux2014_aarch64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.4 kB)
Successfully installed pycryptodome-3.20.0
(21:32:25) (~/.Teaching/cryptanalysis/TD/TD4/Sage/
- (venv:TestAttack)--> ls
attack.py paddingOracle.py venv
```

Attention, si vous l'installez avec pip, cette bibliothèque s'appelle alors Crypto :

```
Python 3.12.6 (main, Sep 7 2024, 14:20:15) [GCC 14.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.20.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from Crypto.Cipher import AES
```

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 chaîne d'octets interprétable comme des caractères UTF-8, alors `myByteString.decode()` renvoie la `str` correspondante.

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

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

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 $\mathbf{c} \stackrel{\text{def}}{=} (c_0, c_1, \dots, c_n) \in (\mathbb{F}_2^{128})^{n+1}$, 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 $\mathbf{p} \stackrel{\text{def}}{=} (p_1, \dots, p_n) \in (\mathbb{F}_2^{128})^n$ 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 $\tilde{\mathbf{c}} \in (\mathbb{F}_2^{128})^n$ tel que

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

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

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

$$(\mu_1, \dots, \mu_{11}, 0x05, 0x05, 0x05, 0x05, 0x05),$$

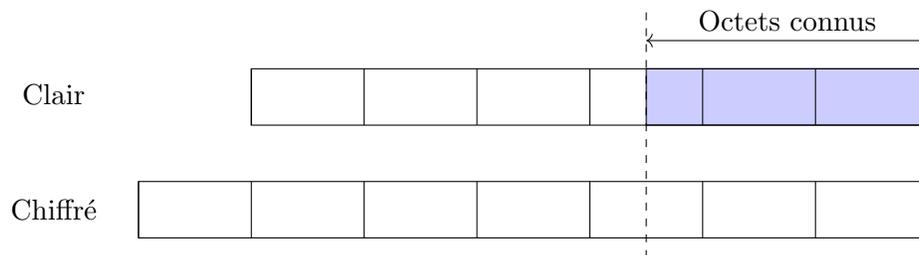
alors `getPaddingValue(ciphertext, key)` renvoie 5.

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 μ_{11} . On va donc faire en sorte que le dernier bloc du chiffré corresponde à un message de la forme

$$(\mu_1, \dots, \mu_{10}, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06)$$

(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. Téléchargez sur ma page web le fichier `paddingOracle.py`. Il s'agit d'un module Python parfaitement fonctionnel, mais volontairement obfusqué. Vous pouvez l'ouvrir, mais ne cherchez pas à en comprendre le sens. Vous pouvez l'importer comme n'importe quel autre fichier Python, et deux fonctions vous sont accessibles :

- `get_ciphertext(name)` qui prend en entrées un nom de la forme `"Prenom_Nom"` et vous attribue un chiffré personnalisé.
- `is_padding_ok(ciphertext, name)` qui prend en entrées une chaîne d'octets et implémente l'oracle de padding correspondant à votre chiffré personnalisé.

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 `"Votre_Pseudo"` et votre texte clair.

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.