TP Cryptographie Padding Oracle Attack
Master CSI

Léo Colisson Palais

Dans ce TP, nous allons implémenter l'attaque "padding oracle attack" décrite dans le cours. Vous trouverez dans le cours Caseine un squelette ainsi que des tests automatiques appelés pour tester la validité de votre code. Nous vous conseillons cependant fortement d'essayer de lancer votre code (idéalement en local) afin de pouvoir le débugger plus facilement.

Il est interdit d'utiliser des LLM pour générer le code demandé, et vous n'utiliserez aucune librairie à part `os.urandom`.

1 Rappels Python

Dans ce TP, nous représentons les messages et les chiffrés avec le type python `bytes` (tableau immutable d'octets). Vous pouvez créer un `bytes` de plusieurs manières:

- `b'mon message'` va créer un nouveau tableau d'octets initialisé aux valeurs ASCII de `mon message`.
- `bytes(42)` va créer un nouveau tableau de 42 octets initialisés à 0.
- `bytes([65,66])` va créer un tableau de 2 octets initialisés respectivement à 65 et 66 (codes ASCII de `A` et `B`).
- Si `a` et `b` sont deux `bytes`, vous pouvez les concaténer avec `a + b`.

Vous pouvez accéder aux différents éléments d'une variable `bytes` comme avec un tableau traditionnel.

Le type `bytes` est cependant immutable, il est donc impossible de modifier un tableau existant. Si vous avez besoin de travailler en place sur un tableau d'octets, vous pouvez utiliser `bytearray`:

- `bytearray(my_bytes_array)` va transformer la variable `my_bytes_array` de type `bytes` en `bytearray`.
- `bytes(my_bytearray_array)` fera l'opération inverse.

Vous pourrez alors modifier et lire les éléments du tableau comme avec un tableau traditionnel.

2 Chiffrement et déchiffrement

Nous allons considérer dans ce TP une attaque contre le mode de chiffrement CBC basé sur le cipher AES, utilisant le padding ANSI X.923. Pour rappel, ce padding ajoute à la fin de chaque message une liste de $n-1$ octets (n étant choisit pour que le message final soit un multiple de la taille de bloc, la taille de bloc d'AES étant 16 octets) contenant la valeur zéro, suivit d'un octet contenant la valeur n.

Commencer par implémenter les fonctions de génération de clé, de chiffrement et de déchiffrement implémenant le mode CBC avec le padding ANSI X.923 et le cipher AES. Plus précisément, compléter dans Caséine la définition des fonctions suivantes:

1. `def gen_key()`, qui génère une clé aléatoire pour le bloc cipher AES avec une taille dé clé de 256 bits. Vous pourrez utiliser `os.urandom(X)` pour générer X *octets* aléatoires.
2. `def cbc_ansix923_enc(cipher, decipher, key, message)`, qui va chiffrer, avec le mode CBC et le padding ANSI X.923, avec la clé `key` (type `bytes`) le message `message` (type `bytes`) avec le block cipher `cipher` (Fonction typiquement instanciée avec AES, prenant en paramètres une clé générée via `gen_key()` et un message sur 128 bits. Je fournis dans Caséine les fonctions `aes_cipher` et `aes_decipher` que vous pouvez utiliser dans vos tests personnels.) `decipher` fonctionne comme `cipher` mais calcule la fonction inverse du block cipher. Votre fonction doit retourner le chiffré (type `bytes`).
3. `def cbc_ansix923_dec(cipher, decipher, key, ciphertext)` qui déchiffre le message `ciphertext` (type `bytes`) (les arguments sont de même type que pour le chiffrement). Si le padding n'est pas correct (forme ANSI X.923), votre fonction devra retourner une erreur.

3 Chiffrement et déchiffrement

Dans la deuxième partie de ce TP, vous implémenterez la fonction `def padding_oracle_attack(oracle, ciphertext)`, où `oracle` est une fonction simulant l'oracle de déchiffrement de l'attaque vue en cours (qui accepte un chiffré, et retourne `True` si après déchiffrement le message a un padding correct, et `False` dans le cas contraire), et `ciphertext` est un chiffré obtenu via `cbc_ansix923_enc` (type `bytes`).

Astuce : commencez d'abord par faire une attaque qui fonctionne lorsque le message rentre sur un seul bloc, avant d'utiliser cette attaque pour déchiffrer des messages plus grands.