Laboratoire 2

Cryptographie

ANNEN Rayane

HEIG VD

17.04.2023

Table des matières

CBC

Quel est l'algorithme (précis) utilisé pour chiffrer ces salaires ? N'oubliez pas la taille de la clef. Le code source vous est donné dans le fichier cbc.py

Le chiffrement utilisé est AES muni du mode opératoire CBC. La taille de la clef utilisée est de 256 bits.

Vous avez trouvé le salaire chiffré (`ct`) ainsi que l'IV correspondant d'un dirigeant d'USB (ils sont dans votre fichier de paramètres). Récupérez le salaire ! Expliquez votre attaque en détails. Le paramètre ID dans votre fichier de paramètres est l'identifiant du dirigeant dont vous souhaitez connaitre le salaire.

La salaire du dirigeant trouvé : **2271 CHF**.

La vulnérabilité trouvée est l'utilisation d'un IV prédictible. En effet, on constate que dans le code un nombre aléatoire est choisi au démarrage du programmme et fait office d'IV. A chaque fois que nous utilisons l'oracle de chiffrement nous constatons que l'IV est incrémenté de 1. Il suffit de calculer une fois un message aléatoire pour obtenir l'IV qui a été utilisé, une fois cela obtenu, on peut simplement prédire la valeur du prochain IV en l'incrémentant de 1.

Pour illustrer l'attaque on rappelle le chiffrement avec CBC :

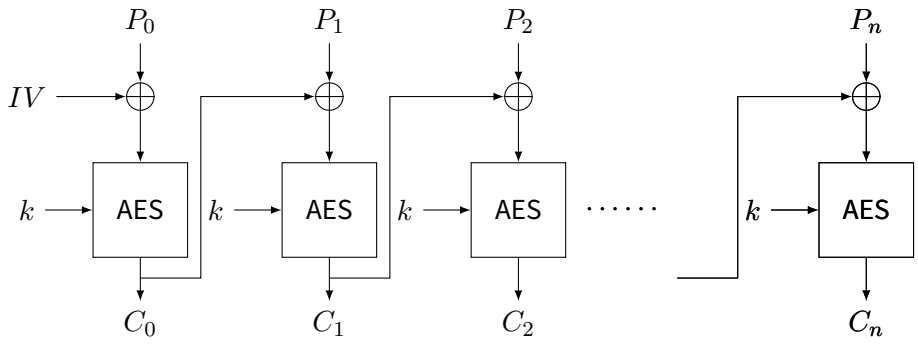


Figure 1: Chiffrement AES-CBC

On remarque que pour le premier bloc, on fait un ou-exclusif entre le premier bloc de texte et le vecteur d'initialisation.

On procède alors maintenant une attaque à texte connu / choisi. Nous connaissons en partie le texte en clair : *Le salaire journalier du dirigeant USB est x CHF* avec $0 \leq x < 3000$, on appelera ce message $\mathcal{M}(x)$. Les blocs de ce messages seront notés de la manière suivante : $\mathcal{M}_i(x)$ avec i le numéro du bloc.

On connaît également le prochain IV qui sera utilisé, en effet, il nous suffit de demander au serveur de chiffrer un message aléatoire pour obtenir le dernier IV utilisé, celui-ci sera appelé $\mathcal{IV}_1$, le prochain utilisé est alors $\mathcal{IV}_1 + 1$.

Nous avons le texte chiffré qu'on appelle $\mathcal{C}$ et son IV qu'on appelle $\mathcal{IV}_0$. Les blocs du texte chiffré sont représentés de manière similaire au message en clair défini ci-dessus.

Ainsi, maintenant que toutes nos variables sont définis, pour un $x \in [\![0, 3000]\!]$, on a son premier bloc qui est calculé de la manière suivante :

$$C_0(x) = E_k((\mathcal{IV}_1 + 1) \oplus \mathcal{M}_0(x))$$

Nous allons jouer sur le fait que l'IV est prédictible et notre connaissance du texte chiffré, une partie du texte clair et de son IV pour trouver notre x. En effet, on va choisir de chiffrer un premier bloc tel qu'il soit de la forme suivante :

$$\mathcal{M}'_0(x) = \mathcal{M}_0(x) \oplus (\mathcal{IV}_1 + 1) \oplus \mathcal{IV}_0$$

Si on calcule le premier bloc :

$$C'_0(x) = E_k(\cancel{(\mathcal{IV}_1 + 1)} \oplus \mathcal{M}_0(x) \oplus \cancel{(\mathcal{IV}_1 + 1)} \oplus \mathcal{IV}_0) = E_k(\mathcal{M}_0(x) \oplus \mathcal{IV}_0)$$

On voit que $C'_0(x)$ est presque égal à C_0 à un x près. En effet, il suffit de trouver le x pour lequel le chiffrement obtenu est le même que le chiffrement que nous avons intercepté. Cela fonctionne car AES-CBC est un chiffrement déterministe et que la clef utilisée est toujours la même.

On a donc juste à faire une recherche exhaustive, cela est raisonnable car il n'y a que 3000 possibilités.

Comment pouvez-vous corriger le problème ? Proposez un code n'ayant pas cette vulnérabilité

On voit clairement que le problème est l'IV prédictible, on ne pourrait pas "annuler" l'IV si on ne le connaissait pas à l'avance. Ce qu'on peut faire c'est de tirer aléatoirement un IV à chaque fois qu'on chiffre au lieu de simplement tirer un IV aléatoire au lancement du programme et l'incrémenter de un à chaque fois.

Le code n'ayant plus la vulnérabilité :

```python
from Crypto.Cipher import AES
from Crypto.Util.strxor import strxor
from Crypto import Random
from Crypto.Util.Padding import pad
from base64 import b64encode
import secrets

ra = Random.new()
```

```python

def oracle(key, plaintext):
    cipher = AES.new(key, mode = AES.MODE_CBC) # IV will be random
    return (cipher.iv, cipher.encrypt(pad(plaintext, AES.block_size)))

if __name__ == "__main__":
    key  = ra.read(32)
    salaire = secrets.randbelow(3000)
    m = b"Le salaire journalier du dirigeant USB est de " + str(salaire
        ).encode() + b" CHF"
    (IV, ct) = oracle(key, m)
    print("IV = %s" % b64encode(IV))
    print("ct = %s" % b64encode(ct))
```

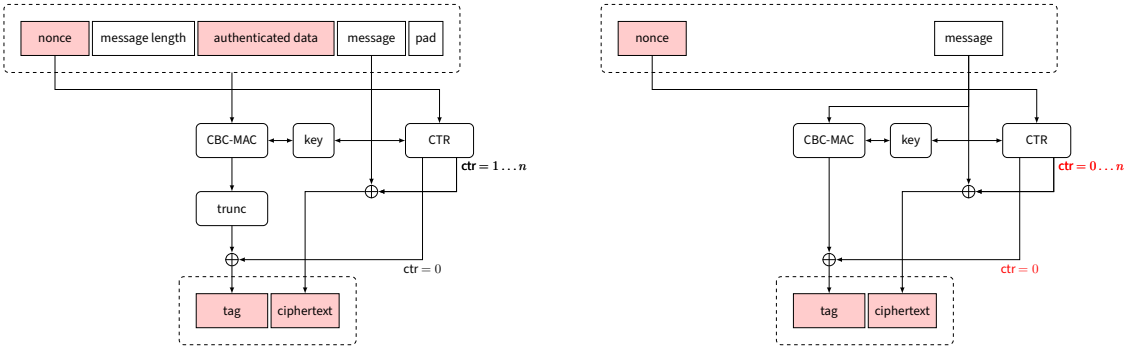
Est-ce que cette attaque s'applique aussi à AES-CTR ?

Non. AES-CTR chiffre l'IV uniquement. On ne peut pas modifier l'entrée de AES-CTR de la manière dont nous l'avons fait avec notre attaque.

CCM

Plusieurs éléments sont modifiés par rapport à CCM. Lesquels ?

Voici le schémas des deux constructions :



(a) AES-CCM original **(b)** AES-CCM modifié

Figure 2: Schémas des constructions AES-CCM

Premièrement, on constate que par rapport à la construction initiale nous utilisons moins d'informations. En particulier, nous ne gérons traitons pas la longueur du message, nous ne gérons donc pas directement le padding. Finalement nous n'utilisons pas de données authentifiées.

Nous constatons également une différence **importante** par rapport à la construction de base, en effet,

nous pouvons voir que lorsque le tag est chiffré avec AES-CTR, nous utilisons bien entendu un compteur initialisé à 0 mais que c'est également le cas quand on chiffre le message.

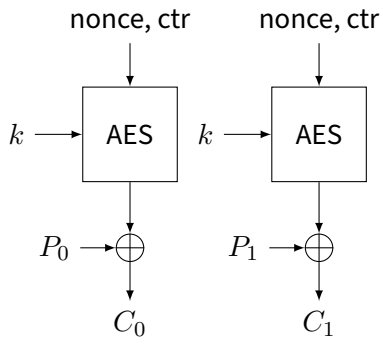
Implémentez le déchiffrement correspondant

```python
def ccm_dec(ciphertext: bytes, tag: bytes, key: bytes, nonce: bytes) -> bytes:
    """Decrypts with AES128-CCM without authenticated data."""
    if len(key) != 16:
        raise Exception("Only AES-128 is supported")

    cipher = AES.new(key, mode = AES.MODE_CTR, nonce = nonce)
    plaintext = cipher.decrypt(ciphertext)

    cipher = AES.new(key, mode = AES.MODE_CTR, nonce = nonce)
    computed_tag = cipher.encrypt(cbcmac(plaintext, key))

    if computed_tag != tag:
        raise Exception("Tampering detected, found: %s, expected: %s" %
            (b64encode(computed_tag), b64encode(tag)))
    return plaintext
```

Vous avez intercepté deux textes clairs (m_1 et m_2) de test ainsi que les textes chiffrés, IVs, et tags correspondants. Utilisez ces informations pour casser la construction. Utilisez vos paramètres pour implémenter votre attaque et donnez le résultat (IV, chiffré, tag) dans votre rapport.

- IV : `7j4I2dF47fg=`
- Chiffré : `AOv2ASSoxGCp4D2q+skZwDGvFNCxr9ucuerphnsX5m8=`
- Tag : `uigrsz4FSBw7z/7U50l5cQ==`

Comme on l'a vu ci-dessus, le tag et le premier bloc du message en clair avec AES-CTR avec le compteur à zéro, c'est la vulnérabilité qui nous intéresse. En effet, si on regarde la construction du chiffrement AES avec le mode opératoire CTR (on s'arrête à deux blocs pour plus de simplicité) :



On voit que si K est le keystream (sortie d'un bloc AES), on a pour un bloc : $K \oplus P = C$ et on retrouve K simplement : $K = C \oplus P$. Si ce K est utilisé aussi pour un tag CBCMAC noté τ, on a que TAG $= K \oplus \tau$

et que $C = K \oplus P$. Ainsi on peut avoir :

$$\begin{aligned} C \oplus \text{TAG} \oplus P &= (\cancel{K} \oplus \cancel{R}) \oplus (\cancel{K} \oplus \tau) \oplus \cancel{R} \\ &= \tau \end{aligned}$$

Comme on retrouve $\tau = \text{CBCMAC}(P)$, on peut forger simplement un tag en formant un message de la manière suivante comme vu dans le cours : $P' = P \parallel \tau \oplus P$.

Maintenant, il faut faire en sorte de rechiffrer le message, cependant contrairement au cas initial, on doit avoir un tag valide pour un message de deux blocs, le second texte chiffré et clair étant sur deux blocs, on peut donc retrouver les deux keystreams utilisés (K_1, K_2) en calculant $P_1 \oplus C_1$ et $P_2 \oplus C_2$.

On peut finalement chiffrer notre message forgé en calculant $C' = P'_1 \oplus K_1 \parallel P'_2 \oplus K_2$

Nous avons désormais notre chiffré C', en le déchiffrant en utilisant la clef et l'IV correspondant à la seconde paire de messages avec l'algorithme affaibli, on constate que la vérification du TAG fonctionnera avec le message forgé, on casse donc bien l'intégrité du message.

Corrigez l'implémentation et faites en sorte que le chiffrement CCM soit correct !

On utilise simplement les fonctionnalités de la librairie `pycryptodome`, il vaut mieux utiliser une solution qui a été réalisée par des experts en cryptographie que de nous même réaliser une solution, on évite ainsi toute erreur.

```python
from Crypto.Cipher import AES
from Crypto.Util import strxor
from Crypto import Random

def ccm(message: bytes, key: bytes) -> tuple:
    """Encrypts with AES128-CCM without authenticated data. """
    cipher = AES.new(key, mode = AES.MODE_CCM, assoc_len=0)
    ciphertext, tag = cipher.encrypt_and_digest(message)
    return (cipher.nonce, ciphertext, tag)

def ccm_dec(ciphertext: bytes, tag: bytes, key: bytes, nonce: bytes) ->
    bytes:
    """Decrypts with AES128-CCM without authenticated data."""
    cipher = AES.new(key, mode = AES.MODE_CCM, nonce=nonce, assoc_len
        =0)
    plaintext = cipher.decrypt_and_verify(ciphertext, tag)
    return plaintext
```

Dans le fichier `ccm_fix.py` il y a également un test montrant que notre attaque ne fonctionne pas.

Bruteforce intelligent

Vous trouverez dans votre fichier de paramètres un texte clair et un texte chiffré. Récupérez les deux clefs secrètes (en base64) et expliquez comment vous avez procédé

Les clefs secrètes :

- $k =$ `iV8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=`
- $k' =$ `WMMAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=`

Pour retrouver les clefs, on va utiliser la relation suivante :

$$c = \mathsf{AES}_k(\mathsf{AES}_{k'}(m))$$
$$\mathsf{AES}_k^{-1}(c) = \mathsf{AES}_{k'}(m) \qquad (\star)$$

Selon la relation intermédiaire $(\star)$ on peut retrouver une paire de clef (k, k') avec la technique d'attaque *Meet-In-The-Middle* connaissant m et c.

En effet, on peut calculer pour toutes les possibilités de clefs k'_i, l'état intermédiaire $\mathsf{AES}_{k'_i}(m)$.

On stocke ensuite ce résultat dans un dictionnaire tel que la clef soit cet état et la valeur la clef k'_i utilisée.

Une fois le dictionnaire construit, pour toutes les possibilités de clefs k_i, on calcule $\mathsf{AES}_{k_i}^{-1}(c)$, ce qui va de nouveau nous donner un état intermédiaire par $(\star)$. On vérifie alors que cet état n'existe pas déjà dans le dictionnaire, si tel est le cas, alors on vient de trouver notre paire de clefs (k, k').

Décrivez votre algorithme. Quelle est la complexité pire-cas de votre attaque ? Comment se compare-t-elle à un bruteforce des deux clefs ?

Algorithme

1. Soient T un dictionnaire, m le message en clair et c le chiffré et ℓ la taille de la clef.
2. Pour toutes les possibilité de clefs k', calculer $I = \mathsf{AES}'_k(m)$ et placer : $T[I] = k'$
3. Pour toutes les possiblité de clefs k, calculer $I' = \mathsf{AES}_k^{-1}(c)$ et vérifier que I' n'est pas déjà dans le dictionnaire. Si c'est le cas on arrête de boucler et nous avons trouver notre clef k et k', autrement on continue.

Analyse de la complexité

L'étape 2 est en $\mathcal{O}(2^\ell)$, c'est également le cas de l'étape 3, la recherche dans le dictionnaire étant faite en temps constant.

La complexité est donc en $\mathcal{O}(2^\ell + 2^\ell) = \mathcal{O}(2^{\ell+1})$ en terme de nombre de chiffrements / déchiffrements dans le pire des cas. La complexité spatiale est quant à elle en $\mathcal{O}(2^\ell)$. On aurait alors pour $\ell = 16$ (la taille dans cet exercice), respectivement des complexités en opérations et spatiales de $\mathcal{O}(2^{17})$ et $\mathcal{O}(2^{16})$.

En comparaison avec une recherche exhaustive, on améliore grandement le calcul, en effet il aurait fallu faire $\mathcal{O}(2^{2\ell})$ opérations dans le pire des cas, toutefois on gagnerait en complexité spatiale puisqu'on serait en $\mathcal{O}(1)$.

En somme, la technique utilisée fait des concessions au niveau mémoire pour gagner en rapidité d'exécution.

Qu'est-ce que votre attaque implique sur la complexité du bruteforce sur 3-key 3DES ?

Triple-DES fonctionne de la manière suivante :

$$\begin{aligned}\mathsf{C} &= E_{k_3}(D_{k_2}(E_{k_1}(\mathsf{P})))\\ \mathsf{P} &= D_{k_1}(E_{k_2}(D_{k_3}(\mathsf{C})))\end{aligned}$$

Une attaque par recherche exhaustive basique pour une taille ℓ est réalisable en $\mathcal{O}(2^{3\ell})$. Si ℓ est fixé à 56 bits (taille des clefs DES). On aurait donc une attaque en $\mathcal{O}(2^{168})$. Toutefois cette méthode est toujours vulnérable aux attaques par *Meet-In-The-Middle*, en effet à partir de la construction on trouve les relations suivantes :

$$\mathsf{C} = E_{k_3}(D_{k_2}(E_{k_1}(\mathsf{P}))) \tag{1}$$

$$D_{k_3}(\mathsf{C}) = D_{k_3}(E_{k_3}(D_{k_2}(E_{k_1}(\mathsf{P})))), \quad \text{mais aussi} \tag{2}$$

$$D_{k_3}(\mathsf{C}) = D_{k_2}(E_{k_1}(\mathsf{P})) \tag{3}$$

Comme on a des clefs $k \in \{0,1\}^{56}$, On va d'abord construire un dictionnaire en $\mathcal{O}(2^{56})$ opérations (2) puis nous allons chercher les clefs candidates en $\mathcal{O}(2^{2\times 56})$ opérations (3) dans le pire des cas, ce qui nous fait au final une recherche exhaustive en $\mathcal{O}(2^{56} + 2^{2\times 56}) \approx \mathcal{O}(2^{112})$.

On a donc une clef avec une sécurité effective uniquement sur 112 bits, malgré sa taille initiale de 168 bits.

Référence : Meet-In-The-Middle Attacks, Stephane Moore, 16 novembre 2010