Write-Up Macaron

Pseudo : xbquo

On notera $\oplus$ l'opération xor, $H_i k_i$ le `hmac` sha256 avec la clé k_i. On notera aussi BB le big block, p_b le block d'avant et n_b celui d'après. Un message taggé sera noté T_i.

1. TL; DR

Ce challenge est issue des épreuves de préselection pour rejoindre l'équipe de France de Cybersécurité lors du challenge européen. Il était évalué à 200 points. L'énoncé du challenge est le suivante :
Le but du challenge est de trouver une contrefaçon sur le code d'authentification de message Macaron. Service : nc challenges1.france-cybersecurity-challenge.fr 2005.
Analysons le code source du fichier joint.

2. Analyse du code source

Nous pouvons tagger un message, ou envoyer un message à vérifier avec son tag associé. Le service découpe notre message M, en différents blocks de 30 octets. Ces blocks suivent un chiffrement décrit ci-après. On va créer un message M, de 60 caractères, par exemple $'0' * 60$, un block de null bytes. D'après cette ligne :

```
m = pad(input, 2 * self.ps)
```

Le block va d'abord être paddé, c'est à dire que si on avait un block qui contient seulement 59 fois zéros, on lui ajoute $1 *$"$\backslash x01$", car il manque un octet à notre block. S'il en manquait 2, on aurait ajouté $2 *$"$\backslash x02$" etc. Ensuite, notre message M paddé, sera découpé en plusieurs blocks de 30 caractères, dans notre exemple on obtient M_0 et M_1 aux valeurs de "0" $* 30$ et "0" $* 29 +$ "01", comme le montre la ligne suivante :

```
nb_blocks = len(m) // self.ps
tag_hash = bytearray(self.ts)
```

On initialise également un block de null byte de 32 caractères, que l'on appellera à ce stade T_0. Avant d'entrer dans la boucle qui nous intéresse, le programme réalise ces opérations :

```
nonce_block = long_to_bytes(self.ctr, self.cs)
prev_block = nonce_block + m[:self.ps]
tag_nonce = nonce_block
self.ctr += 1
```

Il crée un block de 4 octets nulles, que l'on appellera N_0 à ce stade. Aussi, on initialise p_b qui sera égal au block de 4 octets précédents concaténés au premier block de M, on a alors la relation suivante : $p_b = N_0 M_0$. On incrémente ensuite un compteur, qui aura de l'intérêt après. Entrons maintenant dans le coeur de la fonction **tag**, celle qui effectue le chiffrement de nos blocks un a un.
Regardons les lignes suivantes :

```
for i in range(nb_blocks - 1):
    nonce_block = long_to_bytes(self.ctr, self.cs)
    next_block = nonce_block + m[self.ps*(i+1):self.ps*(i+2)]
    big_block = prev_block + next_block
```

On itère à travers le nombre de blocks, moins un. On attribue une nouvelle valeur au nonce, il devient N_1 et contient toujours un block de 4 octets, dont le dernier vaut 01 car on a incrémenté la variable `ctr` avant d'entrer dans la boucle. On crée alors n_b, qui prends la valeur de p_b concaténé au block suivant de notre message. Dans le cas de la première entrée dans la boucle, on considère M_1, il vient alors l'égalité : $n_b = N_1 M_1$. On initialise alors une variable, notre BB (sans mauvais jeu de mot), qui prends pour valeur la concaténation de p_b ainsi que de n_b, on a alors $BB = p_b n_b = N_0 M_0 N_1 M_1$. Etudions à présent la fin de notre boucle :

```
digest = hmac.new(self.k1, big_block, sha256).digest()
tag_hash = bytearray([x ^ y for (x,y) in zip(tag_hash, digest)])
prev_block = next_block
tag_nonce  = tag_nonce + nonce_block
self.ctr += 1
```

C'est ici qu'intervient pour la première fois le hashage du block BB avec la clé k_1, que nous notons $H_1 k_1(BB)$. A la ligne suivante, on réattribue une nouvelle valeur à notre tag, il devient donc T_1 et est égal au xor, caractère par caractère du T_0 et $H_1 k_1(BB)$ dans notre exemple. Il en découle la relation suivante : $T_1 = T_0 \oplus H_1 k_1(BB)$. Ensuite, p_b devient n_b soit $p_b = N_1 M_1$, et le tag nonce est incrémenté avec la valeur de N_i (ou i est l'itérateur dans la boucle). Dans notre exemple qui contient 2 blocks, on sort alors de la boucle, et notre T_1 est une nouvelle fois hashée :

```
tag_hash = hmac.new(self.k2, tag_hash, sha256).digest()
```

Il vient : $T_2 = H_2 k_2(T_1) = H_2 k_2(N_0 M_0 N_1 M_1)$.
En généralisant ce qu'il se passe dans la boucle, si T_n est le tag final hashée avec la clé k_2, on a, grâce à une relation de récurrence :

$$\begin{aligned} T_{n-1} &= T_{n-2} \oplus H_{n-1} k_1(N_{n-2} M_{n-2} N_{n-1} M_{n-1}) \\ &= T_{n-1} \oplus T_{n-2} \oplus H_{n-2} k_1(N_{n-3} M_{n-3} N_{n-2} M_{n-2}) \\ &= ... \end{aligned}$$

Il s'ensuit que T_{n-1} ne dépend que du xor avec T_0, et des $H_i k_1(BB_i)$. Le xor étant une opération symétrique et réversible, ce service est vulnérable et nous allons voir cela dans la prochaine section.

3. Attaquer le service

Etant donné que tous les T_{n-1} ne dépendent que de T_0 et des $H_i k_1(BB_i)$, on va pouvoir forger un message et un nonce à partir d'un message connu. On va d'abord envoyer un message à tagger, puis un second à vérifier. Le premier message sera un block de null byte, M, contenant 59 zéros, qui paddé donnera 59 zéros plus 01. Après avoir été chiffré, on obtient $T_2 = H_{2k2}(T_1 \oplus H_1 k_1(N_0 M_0 N_1 M_1))$. A partir de ça, comme T_0 apparaît dans la relation de récurrence explicitée plus haut, on en déduit :

$$\begin{aligned} \exists\, T_3, \qquad T_3 &= T_2 \oplus H_2 k_1(N_2 M_2 N_3 M_3) \\ &= T_1 \oplus H_1 k_1(N_1 M_1 N_2 M_2) \oplus H_2 k_1(N_2 M_2 N_3 M_3). \end{aligned}$$

On veut donc que $H_1 k_1(N_1 M_1 N_2 M_2) \oplus H_2 k_1(N_2 M_2 N_3 M_3)$ s'annule, pour retrouver uniquement $H_2 k_2(T_1)$, et ainsi avoir deux hash identiques. Or, pour que ces *hmac* soient identiques, il faut que :

$$N_1 M_1 N_2 M_2 = N_2 M_2 N_3 M_3 \iff N_1 = N_2 = N_3,\ M_1 = M_2 = M_3.$$

On connaît les valeurs de M_0 ($30 * "0"$), M_1 ($29 * "0" + "1"$), N_0 (0000) et N_1 (0001). On peut alors créer un message M', tel que $M_1 = M_2 = M_3 = 29 * "0" + "1"$, et $N_1 = N_2 = N_3 = 0001$. On vient donc de créer un message de longueur 90 (octets), qui a le même hash que M, comme le décrit la relation ci-dessus ou les xor s'annulent.

4. Script de résolution

Bon, mon script est un peu laid, mais, il fait le taff et c'est ce qui compte !

```python
from Crypto.Util.number import long_to_bytes
from pwn import *

r = remote("challenges1.france-cybersecurity-challenge.fr", 2005)
M = b"0" * 59

print(r.recv())
r.sendline("t")
print(r.recv())
r.sendline(M.decode())
tag_hash = r.recvline()
tag_nonce = r.recvline()
tag_hash = tag_hash.decode().split(" ")[-1][:-1].encode()
tag_nonce = tag_nonce.decode().split(" ")[-1][:-1].encode()

N0=tag_nonce[:4]
N1=tag_nonce[4:]
N2 = N1
N3 = N2

t_nonce = N0 + N1 + N2 + N3
print(t_nonce)
print(tag_hash)
r.sendline("v")
print(r.recv())
```

```
M_prime = M + b"\x01" + b"0"*29 + b"\x01" + b"0"*29
print(M_prime)
r.sendline(M_prime.decode())
print(r.recv())
r.sendline(tag_hash)
print(r.recv())
r.sendline(t_nonce)
print(r.recv())
print(r.recv())
```