Write up TM1 14 15 PPTI

Challenge: TME #1 - Length-Extension Attack

Write-up : Attaque par Extension de Longueur sur SHA-256

1. Contexte du Challenge

Nous nous trouvons face à une implémentation vulnérable d'un code d'authentification de message (MAC). Le but est de forger un nouveau message valide sans connaître la clé secrète.

Données fournies :

- **MAC :** `SHA256(K || M)`
- **Clé secrète** `K` **:** 128 bits (16 octets), inconnue.
- **Message** `M` **:** `"Hello World!"`
- **Tag valide :** `f51b55788c563203fe7bc29b168030290cff087fd125ba049c2d3be351d18853`

Notre objectif est de générer un nouveau message `M'` contenant le message original suivi d'une extension arbitraire (ici `"flag"`), accompagné de son tag valide.

2. Analyse de la Vulnérabilité

2.1 Identification de la Faille

La construction `MAC = H(K || M)` utilisant une fonction de hachage de la famille Merkle-Damgård (comme SHA-256) est intrinsèquement vulnérable à l'**Attaque par Extension de Longueur (Length Extension Attack)**.

Points clés :

1. SHA-256 traite les données par blocs de 64 octets.
2. L'état interne de la fonction de hachage après le traitement d'un message est directement exposé dans la sortie (le hash).
3. Si nous connaissons la longueur du message haché (`K || M`), nous pouvons reconstruire le padding interne qui a été appliqué.
4. En réinitialisant l'état interne de SHA-256 avec le hash fourni (`tag`), nous pouvons reprendre le calcul et ajouter des données supplémentaires, comme si la clé était toujours présente au début.

2.2 Conditions Requises

Pour que cette attaque fonctionne :

1. ✅ On connaît le hash d'un message valide (`tag`).
2. ✅ On connaît la taille de la clé (16 octets).
3. ✅ On peut prédire le padding qui a été ajouté à `K || M` .

3. Implémentation de l'Exploit

3.1 Étapes de la Solution

Étape 1 : Reconstruire le Padding "Glue"
Le message original `K || M` fait $16 + 12 = 28$ octets. SHA-256 ajoute un padding pour atteindre un multiple de 64 octets (avec la longueur encodée à la fin).
Ce padding (appelé "glue padding") fera partie intégrante de notre message forgé `M'` .

Structure du bloc traité initialement :
`[ K (16) ] [ "Hello World!" (12) ] [ Padding (36) ]` = 64 octets.

Étape 2 : Initialiser l'état SHA-256
Au lieu d'utiliser les constantes d'initialisation standards (IV), nous injectons le `tag` fourni comme état initial des registres `h0` à `h7` . Cela simule le fait que SHA-256 a déjà traité `K || M || Padding` .

Étape 3 : Traiter l'extension
Nous demandons à notre fonction SHA-256 modifiée de traiter uniquement notre extension (`"flag"`). La fonction calculera :
`H( State_initial=Tag, Data="flag" )`
Ce qui équivaut mathématiquement à :
`H( K || "Hello World!" || Padding || "flag" )`

3.2 Code Complet (Solution)

```
import struct

# --- Fonctions utilitaires SHA-256 ---

def right_rotate(value, amount):
    """Effectue une rotation binaire vers la droite (ROTR)."""
    return ((value >> amount) | (value << (32 - amount))) & 0xFFFFFFFF

def sha256_padding(length):
    """
    Génère le padding standard SHA-256.
    Le padding consiste en :
```

```
    1. Un bit '1' (octet 0x80)
    2. Des bits '0' jusqu'à ce que la taille soit congrue à 448 mod 512 (56
mod 64 octets)
    3. La longueur originale du message en bits, codée sur 64 bits (big-
endian)
    """
    padding = b'\x80'
    # Calcul du nombre de zéros nécessaires pour atteindre la fin du bloc
moins 8 octets
    padding += b'\x00' * ((56 - (length + 1) % 64) % 64)
    # Ajout de la longueur en bits à la fin (struct.pack '>Q' = unsigned
long long big-endian)
    padding += struct.pack('>Q', length * 8)
    return padding

def sha256_process_chunk(chunk, h, k_constants):
    """
    Traite un bloc de 64 octets et met à jour l'état interne (h).
    C'est ici que réside le cœur de l'algorithme SHA-256 (fonction de
compression).
    """
    w = [0] * 64
    # 1. Préparation du message schedule (W)
    # Les 16 premiers mots sont le message lui-même
    for i in range(16):
        w[i] = struct.unpack('>I', chunk[i*4:(i+1)*4])[0]

    # Les 48 mots suivants sont dérivés des précédents
    for i in range(16, 64):
        s0 = right_rotate(w[i-15], 7) ^ right_rotate(w[i-15], 18) ^ (w[i-15]
>> 3)
        s1 = right_rotate(w[i-2], 17) ^ right_rotate(w[i-2], 19) ^ (w[i-2]
>> 10)
        w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFF

    # Initialisation des variables de travail avec l'état actuel
    a, b, c, d, e, f, g, hi_val = h

    # 2. Boucle principale de compression (64 tours)
    for i in range(64):
        s1 = right_rotate(e, 6) ^ right_rotate(e, 11) ^ right_rotate(e, 25)
        ch = (e & f) ^ ((~e) & g)  # Fonction 'Choose'
        temp1 = (hi_val + s1 + ch + k_constants[i] + w[i]) & 0xFFFFFFFF

        s0 = right_rotate(a, 2) ^ right_rotate(a, 13) ^ right_rotate(a, 22)
        maj = (a & b) ^ (a & c) ^ (b & c) # Fonction 'Majority'
```

```
        temp2 = (s0 + maj) & 0xFFFFFFFF

        # Rotation des variables d'état
        hi_val, g, f, e = g, f, e, (d + temp1) & 0xFFFFFFFF
        d, c, b, a = c, b, a, (temp1 + temp2) & 0xFFFFFFFF

    # 3. Mise à jour de l'état intermédiaire
    # On ajoute les résultats compressés à l'état précédent
    return [(x + y) & 0xFFFFFFFF for x, y in zip(h, [a, b, c, d, e, f, g,
hi_val])]

# Constantes K (racines cubiques des premiers nombres premiers)
K_CONSTANTS = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2
]

# --- DONNÉES DU CHALLENGE ---
original_hash =
"f51b55788c563203fe7bc29b168030290cff087fd125ba049c2d3be351d18853"
original_message = b"Hello World!"
secret_len = 16 # La clé fait 128 bits
append_data = b"flag"

# --- EXPLOITATION ---

# ÉTAPE 1 : Calculer M' (Message complet forgé)
# ---------------------------------------------
# Nous devons reconstruire le "glue padding" qui a été haché avec le message
original.
# Ce padding dépend de la longueur totale (clé + message).
total_original_len = secret_len + len(original_message) # 16 + 12 = 28
```

```
octets
glue_padding = sha256_padding(total_original_len)

# Le message forgé est la concaténation de tout ce bloc (sans la clé) +
notre extension
forged_message_bytes = original_message + glue_padding + append_data

# ÉTAPE 2 : Calculer le Tag (Extension SHA-256)
# ----------------------------------------------
# Nous allons reprendre le hachage là où il s'est arrêté.
# L'état interne (h) est initialisé avec les valeurs du hash original.
h = [int(original_hash[i:i+8], 16) for i in range(0, 64, 8)]

# Le hachage original a traité un bloc complet de 64 octets (Clé + Msg +
Padding).
# Pour SHA-256, nous avons donc déjà traité 64 octets.
bytes_processed_so_far = 64

# Nous préparons le NOUVEAU bloc à hacher.
# Il contient notre extension ("flag") + le padding final pour ce nouveau
message global.
final_total_len = bytes_processed_so_far + len(append_data)
block_to_hash = append_data + sha256_padding(final_total_len)

# Nous traitons ce bloc avec notre fonction de compression manuelle,
# en utilisant l'état 'h' récupéré du tag précédent au lieu de l'IV
standard.
for i in range(0, len(block_to_hash), 64):
    h = sha256_process_chunk(block_to_hash[i:i+64], h, K_CONSTANTS)

# Reconversion de l'état final en chaîne hexadécimale
new_tag = ''.join(f'{x:08x}' for x in h)

print("-" * 20)
print("M' (Message complet à envoyer en hex) :")
print(forged_message_bytes.hex())
print("-" * 20)
print("Nouveau Tag valide (en hex) :")
print(new_tag)
print("-" * 20)
```

4. Résultats

Après exécution du script d'attaque, nous obtenons les valeurs requises pour valider le challenge.

Message forgé M' (en hexadécimal) :
(Correspond à `"Hello World!"` + padding + `"flag"`)

```
48656c6c6f20576f726c6421800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000e0666c6167
```

Nouveau Tag valide (en hexadécimal) :

```
8b868cea93c73c7afa7ed59207cbbda8720990c20af47ccced6dfecbd9fe19b

Ce couple `(M', Tag)` sera accepté par le serveur comme valide, car il correspond mathématiquement au calcul que le serveur ferait avec la clé secrète.
```