Doryan MENDY
Timothée LONCHAMPT

SAE 3: Sujet de crypto

Sommaire

INTRODUCTION

Dans cette introduction, nous allons redéfinir les deux parties distinctes de cette SAE. Pour commencer nous générons un module de chiffrement et de déchiffrement d'un fichier texte contenant tous les mots de passe de notre plateforme à l'aide du chiffrement RC4. Dans une deuxième partie, nous allons définir ce qu'est une fonction de hachage cryptographique notamment la fonction de hachage MD5. Nous conclurons sur l'utilisation de celle-ci dans le domaine de la cryptographie.

Algorithme RC4

Voici l'ensemble des étapes de la réalisation de notre code en python :

```
def rc4(cle, message):
    S = list(range(256))
    j = 0
```

1. `S = list(range(256))`: Initialise une liste `s` avec les valeurs de 0 à 255, représentant l'état initial de la permutation utilisée par RC4. Et `j = 0`: Initialise une variable `j` à zéro.

```
#KSA
for i in range(256):
    cle_binaire = ord(cle[i % len(cle)])
    j = (j + S[i] + cle_binaire) % 256
    S[i], S[j] = S[j], S[i]
i = j = 0
```

2. **KSA (Key Scheduling Algorithm)** : Cette partie génère la permutation initiale de la boîte de substitution (ensemble des opérations logiques toujours réalisées dans un même sens) en fonction de la clé fournie.
    - `for i in range(256)`: boucle qui parcourt les indices de 0 à 255.
    - `j = (j + S[i] + key[i % len(key)]) % 256`: Met à jour la variable `j` avec les opérations utilisé dans RC4.
    - `S[i], S[j] = S[j], S[i]`: permutation des valeurs dans la liste `S` avec les indices `i` et `j`.
    - `i = j = 0`: Réinitialise les variables `i` et `j` pour la phase suivante.

```
#PRGA
texterempli = []
for lettre in message:
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]
    k = S[(S[i] + S[j]) % 256]
    texterempli.append(chr(ord(lettre) ^ k))

resultat = ""
for char in texterempli:
    resultat += char
return resultat
```

3. **PRGA (Pseudo-Random Generation Algorithm)** : Cette partie génère le flux de clés pseudo-aléatoire en utilisant le KSA
    - `for char in message`: Parcourt chaque caractère du message en clair.
    - `i = (i + 1) % 256`: Met à jour `i` pour ne pas qu'il sorte de la boucle
    - `j = (j + S[i]) % 256`: Met à jour `j` pour ne pas qu'il sorte de la boucle
    - `S[i], S[j] = S[j], S[i]`: Échange les valeurs dans la liste `S`
    - `k = S[(S[i] + S[j]) % 256]`: Calcule la clé à utiliser pour chiffrer le caractère courant.
    - `texterempli.append(chr(ord(char) ^ k))`: Effectue l'opération XOR (ou exclusif) entre le caractère et la clé, puis ajoute le résultat à la liste `texterempli`
    - Il nous suffit de retourner le texte chiffré sous forme de chaîne de caractères avec resultat

Table généré

Voici la table généré grâce à notre programme, elle nous permettra de faire des tests sur les mots de passe introduit dans notre plateforme.

Clé	Suite chiffrante	Texte en clair	Texte chiffré
606205A19A15C196	ã·d¢Ö0ƒj Ôy	KJ5aPNoa4mKa61w6	“ ÛÑô'®ã÷b×MÂJí
37591DA92E52FB4E	ì$¸zô` ëóå,	kYAoSH7FEWNYfEqR	ì×–ªÊå÷×Rd¾x™íÑ
63E6C62E989725F8	FÅ Sõ®ÔU(‘ ‘g!h	orYC5djBtLZtue9T	®Å‰ VÝ ©ÄhM ÐbÎØh
2CB328D70607A387	Ó¨5!J}z‡²‡s,N	W4FQkXUxOUOhx86m	1³ÛÝÇH†d¤£ ØÍkµoA
C1A23D1D3CC23802	§ ^˜Å'ÅÍz ℮2 4÷	qsCtv3tKKN8nADsn	À7Ç¨_ã6¨ÐYd d /,
CEEE65AD43229D2D	”÷+Òš OlÐ*~3KCÄ¦	TNyZ8wnuwbRSHPkI	Åo8¥Là3®Ã sÛËCÇ
57F9888F82E55442	ºóåÐÙüÊÁ.¸Þzr^	MgVkCKNQd8rl97W0	í¢2äBàåMÀWõ£□G
924DC6DCBA2DBDC1	”È Êå¬* HZB	DDEf012baZ0zQGkf	℮¤§”yÕä`ÌñM+À™½ÄÏ
BA8FAF6041121B49	Xð¹ØHZB	eG3BlYZpHVvh10pd	éb¹õX ‘íä¬f5ìÁÓh
F9F9EC6E1A6BDCD2	X í ¯&¦PÙcNÂ	Tynb34YwKnKpCNfm	¢éùy«Zu4Ýäö£æÆ
68997C426349CD3F	Hë!¾ ‘Áé~Là9‹`¹‰	T96bMuq8Hv55HmZg	Hõ×°°’ bFÛá3ÏhÎ
9B6FA5F6265C1800	ˆ‰ ‘EE®þF˜Ox ·}	fGVUKbmCQ9uio3xW	h!Ñ”M¨ûøRÉ ÔRíºÄF
89196A79EB34F2EE	I6?”þùÂdo}‡}A	T96bMuq8Hv55HmZg	®¬cÔæ¦0Æš cíÉM
397B6E40DF520473	Ò"{Mmš¨ 78ÊH†d¤	NYZlFIMm0LGwmSSC	îjããd´ÐÛ%ÏÈ~Æ™ dª²

Fonction de hachage

1. Définition

Pour commencer, une fonction de hachage est comme toute fonction classique, à partir d'une donnée de départ nous pouvons en ressortir une donnée de sortie. La notion importante de la fonction de hachage résulte dans le fait que la donnée de sortie à toujours une même taille fixe. Les données peuvent être de n'importe quelle forme (texte, image, etc…). Mais dans tous les cas, ces données seront transformées en texte binaire avant d'appliquer une fonction de hachage dessus.

2. Propriété

Une bonne fonction de hachage doit présenter et résister à toutes les contraintes qu'elles peut rencontrer. Notamment aux collisions et aux préimages. Étant donné qu'une fonction de hachage transforme un message de taille arbitraire en un message de taille fixe. Par exemple, si une fonction de hachage transforme un message quelconque en message de 32 bits, on se retrouve avec 2^32 possibilités. Or il y a une infinité de messages d'entrée possibles, si je reprends l'exemple au dessus il suffit de trouver deux messages d'entrée qui ont les 32 premiers bits qui sont équivalents.
Concernant les préimages, une préimage est le fait de trouver à partir d'un message haché H, un message en clair M dont le haché est H. Si nous prenons comme exemple "Comme", on peut trouver comme préimages "Comme des enfants" ou "Comme des garcons".
Il est important de faire la nuance entre les deux, trouver une collision revient à trouver deux messages **quelconques** qui ont le même haché. Tandis que trouver un préimage revient à trouver deux messages M et M' **identiques** qui ont le même haché. Il est donc beaucoup plus difficile de trouver un préimage que de trouver une collision.
Une bonne fonction de hachage doit être indispensable dans tous les protocoles de sécurité que nous utilisons au quotidien (carte bleue, paiements, signature électronique, etc…). Elle doit donc résister aux deux contraintes présentées juste avant.
On veut qu'une fonction de hachage donne une "empreinte" (un haché) de notre donnée initiale, mais on ne veut pas qu'à partir d'une empreinte, on puisse fabriquer un message dont le haché soit cette empreinte.
Concernant les collisions, il y en aura toujours, il faut donc trouver une solution pour ne pas en repérer facilement. wsdex

3. Fonction de hachage MD5

MD5 (Message Digest Algorithm 5) est un algorithme de hachage cryptographique encore bien utilisé aujourd'hui. Il a été conçu par Ronald Rivest en 1991 comme une évolution de ses prédécesseurs MD2 et MD4. MD5 produit une empreinte de 128

bits, une empreinte est le résultat d'une donnée après l'application d'une fonction de hachage dessus.

a. Fonctionnement

MD5 prend en entrée un message de longueur variable, quel que soit le format de données (texte, fichier binaire, etc.).

b. Remplissage des données

Le message est complété à partir de la longueur spécifique compatible à l'algorithme.

c. Initiation des variables

MD5 utilise quatre registres de 32 bits (A, B, C, D) pour stocker l'état interne de l'algorithme.

d. Division des blocs

Le message est divisé en blocs de 512 bits. Chaque bloc subit une série d'opérations logiques et arithmétiques.

e. Fonction non linéaire et itératives

MD5 utilise quatre fonctions non linéaires appliquées sur les registres au-dessus pour introduire de la complexité. De plus, les blocs sont traités de manière itérative.

f. Valeur finale

L'empreinte finale est obtenue en concaténant les valeurs hexadécimales des registres A, B, C et D, formant ainsi un hash de 128 bits.

4. Exemple du MD5

A titre d'exemple, il est très compliqué d'expliquer étape par étape ce qu'il se passe notamment à cause du nombre d'opérations et de la complexité de la fonction. Pour cela nous allons utiliser les fonctions native de celle-ci en python voici le code et voici le résultat avec la chaîne de caractère "toilette"

Voici le code python :

```python
# Importer la bibliothèque hashlib
import hashlib

# Définition de  la chaîne de caractères toilette
texte_a_hasher = "toilette"

# Creation d'un objet MD5
objet_md5 = hashlib.md5()

# Convertir la chaîne en bytes
byte = texte_a_hasher.encode('utf-8')

#mettre à jour l'objet MD5
objet_md5.update(byte)

# Obtenir l'empreinte MD5 en format hexadécimal
empreinte_md5 = objet_md5.hexdigest()

# Afficher les résultats
print("Texte d'origine :", texte_a_hasher)
print("Empreinte MD5 :", empreinte_md5)
```

voici le résultat:

```
Texte d'origine : toilette
Empreinte MD5    : 535779fdef24585375be9c10994b43ab
```

Conclusion

Dans cette SAE, nous avons exploré deux aspects essentiels de la cryptographie : l'algorithme RC4 pour le chiffrement et la fonction de hachage MD5.

L'algorithme RC4, bien que vieux, reste un algorithme de chiffrement symétrique encore utilisé aujourd'hui. Nous avons implémenté une version simple en Python pour chiffrer un texte avec une clé donnée. Le RC4 utilise un mélange de permutation appelé KSA (Key Scheduling Algorithm) et un générateur pseudo-aléatoire appelé PRGA (Pseudo-Random Generation Algorithm) pour produire une suite chiffrante. La table générée avec différentes clés, suites chiffrées, textes en clair, et textes chiffrés a été présentée pour illustrer le fonctionnement de l'algorithme.

Nous avons également exploré la fonction de hachage MD5, une empreinte de 128 bits qui produit une empreinte unique pour une entrée donnée. Nous avons examiné son fonctionnement général, du remplissage des données à la génération de l'empreinte finale. Un exemple concret avec la chaîne de caractères "toilette" a été fourni.

Cependant, il est essentiel de noter que MD5 présente des vulnérabilités et est largement déconseillé pour les applications nécessitant une sécurité robuste. Des attaques telles que les collisions sont possibles, remettant en question son utilisation dans des contextes sensibles.

En conclusion, bien que RC4 et MD5 aient été des outils importants dans le passé, il est crucial d'adopter des alternatives plus modernes et sécurisées. Les avancées constantes dans le domaine de la cryptographie ont conduit au développement d'algorithmes plus robustes et résistants aux attaques.