Wa0K 01/11/2022

Write-up - Le chiffre allemand

Auteur : Wa0k CTF Flag'Malo 2022

L'énoncé du challenge nous donne énormément d'informations. Ce dont il est important à retenir :

- Le système de chiffrement est connu et l'inventeur est le lieutenant Allemand Fritz NEBEL.
- Un chiffrement par clé a été ajouté au système, et la clé est connue.
- Le flag est le lieu de la prochaine attaque.

En commençant les recherches sur Internet, nous devinons le système connu qui le chiffre ADFGVX inventé par lieutenant Allemand Fritz NEBEL et utilisé pendant la Première Guerre Mondiale.

En s'informant sur son fonctionnement, la table de substitution fournit fait directement référence à la table de substitution utilisé dans le chiffre ADFGVX. Bonne nouvelle donc !

Pour la suite, c'est l'heure de faire un peu griller des neurones et d'analyser le code source en Python !

1. Analyse globale du code

Le code source contient deux fonctions : basic_encryption, enhanced_encryption. C'est la fonction basic_encryption qui appelle la deuxième fonction enhanced_encryption.

Les deux fonctions ont des arguments explicites :

- basic_encryption a pour argument : secret, substitution_table, key.
- enhanced_encryption a pour argument : substituted_secret, key.

2. Analyse de la première fonction : basic_encryption

Celle-ci utilise une variable nommée substitution_table soit table de substitution, un hasard ?

Si nous continuons les investigations, nous pouvons déduire que la fonction effectue pour chaque lettre du secret à chiffrer, une substitution de la lettre par les coordonnées (y,x) associées dans la table de substitution donnée (cf. code ci-dessous).

```
for letter in secret:
    index = 0; limit_index = substitution_table.index(letter)
    for y in "ADFGVX":
        for x in "ADFGVX":
            substituted_secret += y+x if index == limit_index else ""
            index += 1
```

Finalement, cette fonction effectue simplement le chiffre ADFGVX avec la table de substitution que nous connaissons.

La fonction renvoie ensuite le résultat de la fonction enhanced_encryption avec pour paramètre d'entrée : le secret substitué et la clé.

```
return enhanced_encryption(substituted_secret, key)
```

Passons donc à la deuxième étape.

3. Analyse de la deuxième fonction : enhanced_encryption

Cette fonction a besoin d'une clé, clé que nous connaissons grâce à l'énoncé du challenge.

En analysant la fonction, nous comprenons que pour chaque lettre du secret, nous prenons la valeur hexadécimale de l'opération arithmétique XOR entre : la valeur ASCII de la lettre et la valeur ASCII la lettre associée dans la clé. La concaténation de toutes ces valeurs constitue le secret chiffré (cf. code ci-dessous).

```
encrypted_secret = ""
for i in range(len(secret)):
    encrypted_secret += hex(ord(secret[i]) ^ ord(key[i]))[2:].zfill(2)
```

Pour information :

- La fonction hex renvoie la valeur hexadécimale (en string) pour le nombre entier donné.
- La fonction ord renvoie la valeur Unicode (ASCII) (en base 10) du caractère donné.
- L'opérateur XOR en Python correspond au symbole ^.
- La fonction zfill(2) permet de compléter par des 0 jusqu'à ce que la chaîne de caractères soit de longueur 2.

Chaque lettre est donc codée en hexadécimal par deux caractères (0-9A-F).

Enfin, la dernière étape consiste à renvoyer le secret encodé en base64.

4. Reverse engineering l'algorithme : la fonction enhanced_encryption

Après avoir compris les différentes étapes de l'algorithme de chiffrement, il est beaucoup plus facile d'aborder le challenge puisque pour retrouver le secret en clair, il suffit d'effectuer les étapes dans le sens inverse !

Tout d'abord, nous décodons le secret chiffré de la base64 et nous obtenons le texte suivant :

```
1E0E0F091D121D0204021D0D0A1C121410140C050E090D06001D06061D1D02110B1C0F171
3160C041A090F0F151C1C1101171404081508110406 1A0E1D0E1D1416160C1B1414170808
0A0B0D0514081417160A150E12120A18061C0B1210091A021C13110A020F151015010E121
213080C0C06101E150D0D120215010513 0C050E0C0519130D080613121E0311081B09170F
0F0A061710190E1107081F081B0302000C191413070016170E0D0F1812001A020E1D03141
3140A181E0602040904090D1F171A070F001C05
```

Nous retrouvons alors nos valeurs hexadécimales, et comme analysé précédemment (cf. partie 3.) une lettre correspond à deux caractères hexadécimaux.

Prenons la première lettre pour exemple :

(1) Prenons les deux premiers caractères du texte soit 1E.

(2) Encodons la valeur 1E en base 10 => `int(secret[i:i+2], 16)`

1E correspond à 30 en base 10. 30 est le résultat de l'opération XOR entre la valeur Unicode de la lettre du secret et de la lettre de la clé.

PS : i est un indice pour parcourir le texte hexadécimal ci-dessus par pas de 2. Cela permet donc de récupérer automatiquement deux caractères hexadécimaux par deux caractères.

(3) Donc pour inverser l'opération XOR, nous devons effectuer une opération XOR entre cette valeur et la valeur Unicode de la lettre correspondante dans la clé. Nous avons donc :

`int(secret[i:i+2], 16) ^ ord(key[i//2])`

Par exemple pour la première lettre chiffrée, cela donne : 30 ^ ord("Y") = 71.

PS : Le texte hexadécimal ci-dessus a une longueur deux fois plus grande que la clé car une lettre chiffrée équivaut à deux lettres hexadécimales. Donc la lettre de la clé correspondante se situe à l'indice i//2 pour tout i quelconque.

(4) 71 est donc la valeur Unicode de la **première lettre du secret substitué** !
Donc la lettre est : chr(71) = G.

Pour obtenir le secret substitué en entier, nous pouvons utiliser une boucle for qui va itérer sur le texte hexadécimal (d'où l'utilisation de l'indice i précédemment ;-)).

Le code pour inverser la fonction enhanced_encryption est le suivant :

```python
def reverse_encryption(secret, key):
    """
    Params:
    -------
    @type secret: string
    @param secret: text to decrypt
    @type key: string
    @param key: decryption key
    """
    secret = bytes.decode(base64.b64decode(secret))
    decrypted_secret = ""
    for i in range(0, len(secret), 2):
        decrypted_secret += chr(int(secret[i:i+2], 16) ^ ord(key[i//2]))
    return decrypted_secret
```

Après exécution du programme, nous obtenons le secret substitué :

GFADVFVGFGGAADVXVXGDDXAAGVDAGVDXXDXDVGGGVGXDVFVGXDGGXGXDGVVFGVDXXDDAGAVGF
AADGVGDVGGGVGGGVGFXVGXXDXXGGDVFADXDVFGXXGVGGADXXDXDVGDAVFVXADDAAAXGGAADVF
VFADGXXGVGFXVGDVDXGGVXGVVGAVXDVGVGDAVFGVGGGGGVXDVGXDVFVG

5. Reverse engineering l'algorithme : la fonction basic_encryption

Lors de l'analyse, nous avons conclu que la fonction basic_encryption effectue simplement le chiffre ADFGVC. Au lieu de réinventer la roue, est-il possible de trouver un outil sur Internet permettant de faire ?

Sur dcode.fr, il est possible de chiffre et déchiffrer le chiffre ADFGVX.

Nous obtenons le résultat : HATEZLAPPROVISIONNEMENTENMUNITIONSLEFAIREMEMEDEJOUR TANTQUELONNESTPASVULATTAQUEDECOMPIEGNEESTIMMINENTE.

La ville de la prochaine attaque est COMPIEGNE.

Nous avons donc le flag : FMCTF{COMPIEGNE}.