Analyse d'une fonction de hashage personnalisée

Il m'a été donné plusieurs empreintes de mots de passe (ou hash en anglais) afin de valider leur solidité.

Le but, analyser ces hash et repérer des données qui peuvent mener au mot de passe en clair.

Si cette recherche est fructueuse, il faudra alors informer le responsable de ce programme du manque de sécurité de son algorithme.

I – Analyse visuelle

La première chose à faire en présence d'un hash est d'analyser sa composition. En effet, certains algorithmes laissent des traces permettant de le déterminer.

Après plusieurs recherches, j'ai eu la confirmation que ces empreintes étaient bien générées par un algorithme personnalisé.

En effet, le modèle ne correspondait à aucun algorithme de hashage connu.

Voici l'exemple d'un hash et de son mot de passe :

HZNa4PoG4
"{21`74<p197284459]p24660\p259341372\o495662567"\i7956.p21920\o56721]p19454..i7956_51b{115}3|3"

1 – Longueur

On remarque à première vue la présence de plusieurs groupes constitués d'une lettre suivie de plusieurs chiffres, tous séparés par un caractère spécial.

Avec le mot de passe en main, j'ai remarqué qu'il y a autant de ces groupes qu'il y a de caractères dans le mot de passe.

On peut donc, grâce à cette empreinte, connaître la longueur du mot de passe en clair.

2- Composition

Ensuite, en se penchant sur la composition des groupes de mots qui composent cette empreinte, on peut remarquer la présence d'une lettre avant chaque chiffre.
Cette lettre correspond au type de caractère du mot de passe en clair.
Avec l'analyse de plusieurs hash, j'ai pu remarquer que :

p ou m = majuscule

q ou f = caractère spécial

n ou o = minuscule

i = chiffre

Grâce à ces informations, nous connaissons également la structure du mot de passe.

3- Sel et clé

Enfin, il reste 2 valeurs à déterminer. En corrélant avec des algorithmes de hashage connus (bcrypt dans l'exemple), on peut remarquer que chaque hash est salé.

Ce sel est représenté par la valeur qui vient après l'underscore à la fin de l'empreinte.

On remarque également au début de chaque empreinte la présence de 2 nombres dont l'utilité n'a pas été encore démontrée.

Ce que l'on sait en revanche, c'est que ces valeurs influent sur la valeur des caractères dans l'empreinte. En effet, si un caractère est en double dans la même empreinte, il aura la même valeur.

On peut déterminer également si un caractère est en double dans un mot de passe.

4- Conclusion de la première partie

Avec ces informations, on sait désormais comment est composé un mot de passe, de sa longueur à son placement.

Avec ces informations, un pirate peut générer un dictionnaire personnalisé afin d'améliorer ses chances de parvenir au mot de passe.

Encore faudrait-il mettre la main sur le programme servant à générer cette empreinte.

En revanche, il est peu probable que cette empreinte soit cassée au vu du nombre de mots de passe qu'il faudrait tester.

On remarque cependant un manque de sécurité qui mène à la fuite de quelques informations sur le mot de passe en clair

Cet algorithme est à perfectionner, mais en l'état, la sécurité qu'il procure est correcte.

II – Analyse dynamique

En ayant mis la main sur l'application générant les hash, j'ai pu analyser l'algorithme en détail afin de déterminer la façon dont ces hash étaient générés.

1 - Difficultés

Le programme cible était compilé depuis WinDev qui possède son propre langage. Afin d'être porté vers une machine ne possédant pas d'interpréteur WinDev, le compilateur doit intégrer toutes les librairies dans l'exécutable, rendant ainsi le fichier final très lourd.

La version analysée pesait 91Mo. La possibilité d'une analyse statique (par désassemblage) d'un programme de cette taille est quasi nulle. Non seulement sa taille, mais également le fait que le programme intègre des librairies dans l'exécutable ne permet pas de se repérer ou de suivre un comportement de façon précise.

2 – Solution

Plutôt que de désassembler ou débugger ce programme (le premier cas irréalisable et le deuxième beaucoup trop imprécis), j'ai décidé d'analyser les APIs appelées par ce programme.

WinDev est un langage très verbeux, et fait donc appel à beaucoup d'API Windows.
Ma première idée fût d'intercepter l'entièreté des APIs. J'ai ensuite recherché le mot de passe en clair pour enfin tomber sur l'API WideCherToMultiByte l'utilisant.

Cette API qui permet de convertir une chaine en octets, s'avère être utilisée pour chaque chaîne de caractères traitées par WinDev.

Il ne me restait plus qu'à intercepter uniquement cette API afin de trouver la routine permettant de générer les hash.

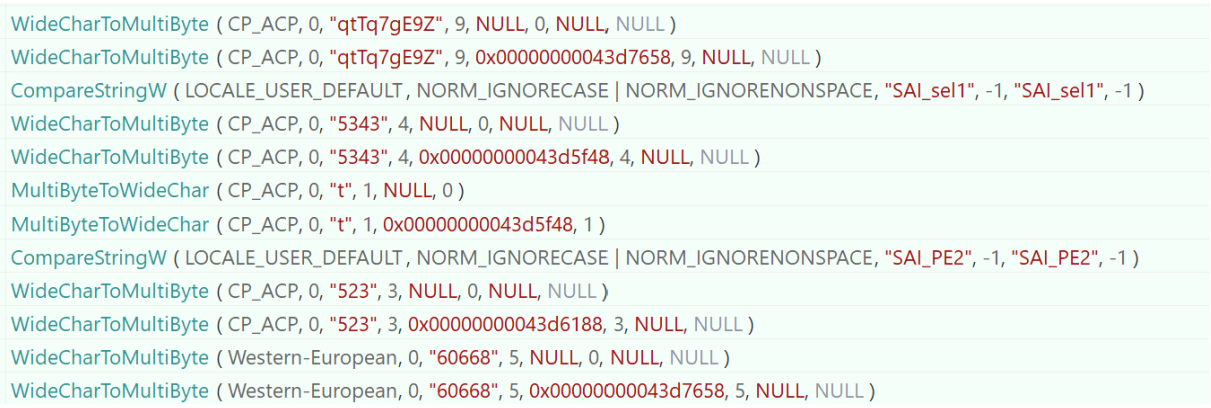

Figure 1.1 Capture d'écran de la fenêtre API Monitor

3 – Fonctionnement détaillé de l'algorithme

Grâce à ces captures, j'ai pu retracer facilement le traitement apporté à la chaîne de caractères. Voici en détail l'analyse de cet algorithme :

a) Génération d'un mot de passe aléatoire

Le mot de passe généré est d'une longueur choisie (ici, 9 caractères) et est composé de caractères alphanumériques.
Cette génération se fait apparemment en choisissant des caractères au hasard dans une liste.

b) Génération de clés

Le programme va générer 3 clés aléatoirement qui serviront à de futures opérations sur les caractères de la chaîne.
Ces clés sont toutes des nombres de 3 chiffres chacune.

c) Génération de sels

Afin d'améliorer un peu plus la sécurité de l'empreinte, il a été choisi de générer 3 sels aléatoires qui serviront à de futures opérations sur la chaîne.
Ces sels sont des nombres de 4 chiffres chacun.

d) Génération du hash

Le hash se construit d'une façon simple. On effectue des opérations sur chaque caractère de la chaîne indépendamment des autres.

Premièrement, le caractère choisi sera converti en code ASCII.
Ce code sera ensuite multiplié par l'une des 3 clés aléatoirement.

Désormais, nous parlerons de groupes, signifiant le résultat de l'opération sur un caractère.

En fonction de la position des caractères, un sel leur sera ajouté, voici le détail de ce placement :

Sel 1 : Placé à la fin du code du premier groupe
Sel 2 : Placé après le premier caractère du code du troisième groupe
Sel 3 : Placé avant le dernier caractère du quatrième groupe

p19728**4459**]p24660\p2**5934**1372\o4956**6256**7"

Ensuite, on ajoute devant chaque groupe, une lettre définie comme suit :

Si le caractère du mot de passe est

- Un chiffre : i
- Une minuscule : n ou o
- Une majuscule : p ou m
- Un caractère spécial : q ou f

Ces différents groupes seront ensuite séparés par un caractère spécial choisi au hasard.

Enfin, les clés seront intégrées dans le hash final comme suit :

Clé 1 Clé 2 Clé 3

{**2**1`**74**”groupes de mots” _**51**b{**115**}3|**3**

4 – Fonction de déhashage et réinterprêtation

Avec ces informations, j’ai pu mettre en place une fonction de « déhashage » permettant de retrouver le mot de passe à partir des hash générés par le programme.

J’ai également recréé mon interprétation du programme de génération d’empreintes en Python.

Ces 2 programmes sont disponibles en annexe à la fin de ce document.

III – Conclusion et réflexions

Avec l’analyse de ce programme, j’ai pu prouver que cet algorithme de génération d’empreinte n’était pas sécurisé.

Par définition une empreinte ne doit pas être réversible. Elle doit uniquement servir à confirmer un mot de passe. Dans cet exemple, les clés utilisées pour « chiffrer » un caractère étaient stockées dans l’empreinte.

J’interprète cette fonction comme un mélange de chiffrement et d’encodage, mais clairement pas assez fiable pour être utilisé pour sécuriser des mots de passe.

Il n’est pas recommandé de créer son propre algorithme de hashage. Il faut plutôt se baser sur des algorithmes ayant déjà fait leurs preuves, voici mes recommandations :

L’algorithme SHA256 est très rapide mais peu recommandé pour les mots de passe.
En revanche bcrypt est un peu plus lent mais reste une valeur sûre pour hasher des mots de passe.