Projet de Cryptographie

« Cassage mot de passe Windows et Linux »

Lionel Coin
Sébastien Pône

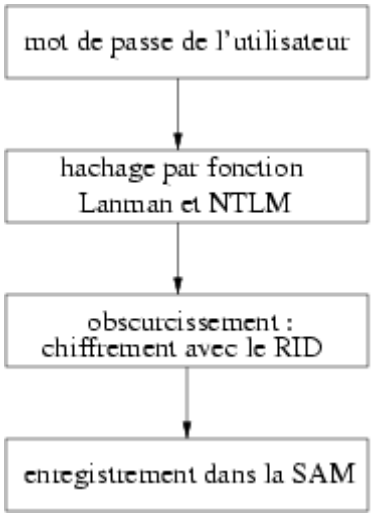
08/09

Un mot de passe est un moyen d'authentification afin de restreindre l'accès à une ressource ou un service. Ils sont notamment utilisés par les systèmes d'exploitation comme Windows et Linux pour limiter l'accès aux comptes des utilisateurs.
Ils sont stockés par le système qui vérifie ainsi leur validité lors de leur saisie. Mais comment se déroule cette opération ? Ne peut-on pas d'une manière ou d'une autre réussir à trouver, casser ce mot de passe ?

➢ Stockage et transformation des mots de passe

Afin de ne pas pouvoir récupérer simplement le mot de passe dans la base de stockage, le système d'exploitation que ce soit Windows ou Linux le stocke après lui avoir appliqué une transformation qui doit être irréversible ou à sens unique (*one way en anglais)*. Cette étape est appelé « hachage irréversible ». Il existe donc plusieurs fonctions de hachage ou algorithmes de hachage qui permettent d'obtenir ce qui est appelé empreinte.

✓ Sous Windows



Dans les systèmes de la famille NT, les mots de passe sont stockés dans une partie de la base de registre (base de données binaire renfermant toutes les données de configuration du système et des applications) nommée SAM. Celle-ci se trouve dans le fichier *$windir$\system32\config\sam* où *windir* correspond au dossier d'installation de Windows, en général « c:\windows ».
Cependant, les empreintes ne sont pas enregistrées telles quelles dans la SAM mais obscurcies par du chiffrement en utilisant l'algorithme DES. Le numéro d'identification de l'utilisateur qui est unique est notamment pris en compte pour empêcher que deux personnes ayant le même mot de passe aient la même entrée dans le fichier.

La base SAM contient les informations suivantes pour chaque utilisateur :

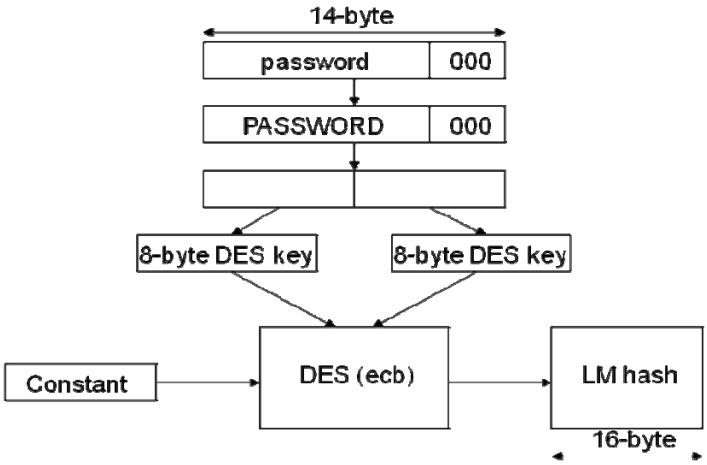
Username:500:e7a3f010af6c05b3abd7b434b51404fe:9dca6a46cb386854a40e2ac0d1de6e00:::

Nom du champ	Description
Username	Nom d'utilisateur
500	ID de l'utilisateur
e7a3f010af6c05b3	Hash **LM** (partie 1)
abd7b434b51404fe	Hash **LM** (partie 2)
9dca6a46cb386854a40e2ac0d1de6e00	Hash **NTLM**

Windows s'appuie principalement sur deux algorithmes pour enregistrer de façon irréversible les mots de passe des utilisateurs qui se nomment Lan Manager (LanMan ou LM) et NT Lan Manager (NTLM).

- **Lan Manager**

Principe :



Le mot de passe est d'abord ramené à 14 caractères. Il est ensuite mis en majuscules et divisé en deux parties de 7 caractères auxquelles des bits de parité sont ajoutés. Chaque partie est indépendamment utilisée comme clé de chiffrement DES à 56 bits pour chiffrer la chaîne *"KGS !@#$%"*. Les deux résultats de 8 octets chacun sont concaténés pour donner l'empreinte Lan Manager de 16 caractères.

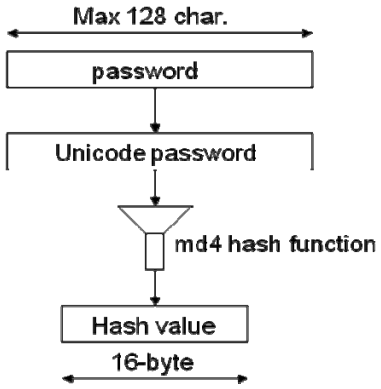
Caractéristiques :

- Limitation du mot de passe pris en compte à 14 caractères.
- Indifférenciation entre les minuscules et majuscules.
- Restriction du nombre de caractères spéciaux (10) et donc de caractères possibles (46).
- Coupure du mot de passe en 2 mots.
- Unicité de l'empreinte par même mot de passe aura (pas d'utilisation de graine).
- Empreinte à 16 caractères.

Remarque :

L'algorithme prend donc 2 mots de 7 caractères et ne fait pas de différence entre les majuscules et les minuscules, ce qui revient à $2*46^7$ d possibles soit un peu plus de $8,7x10^{11}$ possibilités différentes.
Alors que l'utilisateur pourrait s'attendre à 72^{14} soit $1,0x10^{27}$ combinaisons distinctes ! La différence est bien entendu gigantesque.

- **NT LanManager**



Principe :

Le mot de passe est tout d'abord limité à 128 caractères puis passé en Unicode, c'est-à-dire qu'après chaque caractère un caractère nul est inséré.
Cette chaîne est passée à la fonction de hachage MD4 (cf. Linux - MD5) pour donner l'empreinte NTLM de 16 caractères.

Caractéristiques :

- Limitation du mot de passe pris en compte à 128 caractères.
- Différenciation entre les minuscules et majuscules.
- Restriction limitée du nombre de caractères spéciaux (caractères diacritiques autorisés)
- Transformation du mot de passe en Unicode.
- Unicité de l'empreinte par même mot de passe (pas d'utilisation de graine).
- Empreinte à 16 caractères.

Remarque :

Le nombre de mots de passe différents est donc largement supérieur à celui pour la fonction utilisée dans le Lan Manager.
A titre de comparaison, il existe $3,7x10^{18}$ (72^{10}) mots de passe différents de 10 caractères contre $8,7x10^{11}$ possibilités pour le LM.

✓ Sous Linux

Dans les systèmes Linux, les mots de passe étaient stockés historiquement dans le fichier */etc/passwd.* Mais ce fichier étant lisible en clair pour tout le monde, les empreintes ont donc été déplacées dans des fichiers accessibles uniquement par les administrateurs, appelés les *shadow passwords*, se trouvant d'ailleurs dans le fichier */etc/shadow.* Contient les informations suivantes pour chaque utilisateur :

Username:HrLNrZ3VS3TF2:501:100:fullname:/home/username:/bin/bash

Nom du champ	Description
Username	Nom d'utilisateur
HrLNrZ3VS3TF2	Mot de passe après le passage dans une fonction de hachage
501	Numéro du compte utilisateur
100	Numéro du groupe d'utilisateurs auquel il fait parti
fullname	Autres informations sans lien avec le mot de passe, le nom complet de l'utilisateur notamment
/home/username	Répertoire personnel de l'utilisateur
/bin/bash	Programme devant être exécuté au login, principalement un « shell ».

Linux s'est d'abord appuyé sur un algorithme basé sur Data Encryption Standard (DES), puis BlowFish lui a succédé et depuis les années 1990, c'est l'algorithme MD5 qui est le plus utilisé dorénavant.

Remarque :

Il peut être intéressant de voir ce qu'est l'algorithme SHA-1 qui est important dans les algorithmes de hachage.

Le SHA-1 prend un message d'un maximum de 2^{64} bits en entrée. Quatre fonctions booléennes sont définies, elles prennent 3 mots de 32 bits en entrée et calculent un mot de 32 bits. Une fonction spécifique de rotation est également disponible, elle permet de déplacer les bits vers la gauche (le mouvement est circulaire et les bits reviennent à droite).

Le SHA-1 commence par ajouter à la fin du message un bit à 1 suivi d'une série de bits à 0, puis la longueur du message initial (en bits) codée sur 64 bits. La série de 0 a une longueur telle que la séquence ainsi prolongée a une longueur multiple de 512 bits. L'algorithme travaille ensuite successivement sur des blocs de 512 bits.

L'algorithme réalise 80 tours de boucles qui modifient le hachage et donne un résultat à chaque tour de boucle.
De plus, le SHA-1 change sa méthode de calcul tous les 20 tours et utilise les sorties des tours précédents.

À la fin des 80 tours, on additionne le résultat qui donne l'empreinte finale.

- **DES**

Principe :

Cet algorithme traite des blocs de 64 bits, dont 8 bits (un octet) servent à tester la parité (pour vérifier l'intégrité de la clé). Chaque bit de parité de la clé (1 tous les 8 bits) sert à tester un des octets de la clé par parité impaire, c'est-à-dire que l'on veut avoir un nombre de '1' impaire dans l'octet à qui il appartient. La clé possède donc une longueur « utile » de 56 bits, donc on ne se sert que de ces 56 bits dans l'algorithme, le reste ne servant pas.

Caractéristiques :

- Effectue des combinaisons, des substitutions et des permutations entre le mot de passe à crypter et la clé.
- Permet de chiffrer et déchiffrer le mot de passe en faisant des opérations possibles dans les deux sens.
- La combinaison entre substitutions et permutations est appelée code produit.

Remarque :

La clé est codée sur 64 bits et formée de 16 blocs de 4 bits, généralement notés k_1 à k_{16}. Etant donné que « seuls » 56 bits servent effectivement à chiffrer, il peut exister 2^{56} (soit 7.2×10^{16}) clés différentes !

- **MD5**

Principe :

MD5 travaille avec un message de taille variable et produit une empreinte de 128 bits.
Le message est divisé en blocs de 512 bits, on fait donc une opération appelée remplissage pour obtenir un multiple de 512 bits. Le remplissage ressemble à ceci :

- on ajoute un '1' à la fin du message
- on ajoute une séquence de '0' (le nombre de zéros dépend de la longueur du remplissage nécessaire)
- la taille du message est écrite sur un entier de 64 bits

Caractéristiques :

- Le remplissage est toujours utilisé.
- permet uniquement le cryptage d'un mot de passe.

Remarque :

La taille du message est codée en Little endian. Le message a une taille en bits multiple de 512, c'est-à-dire qu'il contient un multiple de 16 mots de 32 bits.

- **BlowFish**

Principe :

Blowfish utilise une taille de bloc de 64 bits et la clé, de longueur variable, peut aller de 32 à 448 bits. Blowfish est basé sur un schéma de Feistel avec 16 tours et utilise des S-Boxes de grandes tailles qui dépendent de la clé.

Caractéristiques :

- Nombreuses contraintes de mise en place
- lent quand il faut changer de clé
- très rapide pour le chiffrement pris séparément.

Remarque :

Les permutations dans Blowfish s'écartent des permutations aléatoires sur 14 tours.

L'attaque nécessite $2^{8r + 1}$ textes clairs connus, avec r le nombre de tours.

De plus, des clés dites faibles sont détectables et cassables avec la même attaque en seulement $2^{4r + 1}$ textes clairs connus. L'attaque ne peut être étendue au Blowfish complet avec ses 16 tours.

Nous avons vu jusqu'à présent les différentes façons permettant de « hacher » un mot de passe avec différents algorithmes. Nous allons, maintenant, voir les différentes attaques qu'il existe pour tenter de casser cette empreinte.

➢ Méthodes de craquage des mots

Tout d'abord, l'algorithme, le plus simple et direct, l'attaque par force brute.

✓ Attaque par force brute

Principe :

Elle consiste à tester toutes les combinaisons possibles les unes après les autres de façon exhaustive.

Méthode :

Pour cela, il faut parcourir toutes les combinaisons de caractères possibles c'est-à-dire tous les mots contenant un nombre de caractère inconnu qui peuvent être des minuscules, des majuscules, des caractères alphanumériques ou des symboles particuliers.

Pour chaque combinaison, il faut donc calculer l'empreinte correspondante (mot de passe « hashé ») qui est, ensuite, comparée à l'empreinte recherché.
Il est à noter que les combinaisons testées ne le sont pas dans l'ordre (a, aa, ab, ac,…) afin d'empêcher d'éventuels logiciels de sécurité de détecter l'attaque.

Avantages/Inconvénients :

Cette attaque est la seule qui donne 100% de chance de trouver le bon mot de passe car elle se base sur la théorie que chaque mot de passe est cassable, mais le temps de calcul pour trouver le bon mot de passe peut être extrêmement long.
En effet, plus le mot de passe est long et avec de nombreux caractères différents (minuscules, majuscules, alphanumériques, symboles) et plus l'attaquant mettra de temps pour trouver le bon mot de passe.
Cette méthode est très sensible aux capacités de calcul des machines effectuant l'algorithme.
On combinera souvent la force brute avec l'attaque par dictionnaire et par rainbow table pour obtenir de meilleurs résultats et un compromis temps/mémoire plus intéressant.

- Efficacité certaine

- Lenteur
- Variabilité selon la machine utilisée

Remarque :

Afin d'illustrer, le temps considérable que peut prendre cette méthode mais aussi l'efficacité certaine, nous pouvons prendre comme exemple, le défi que le RSA, organisme américain chargé notamment de tester les systèmes de sécurité, avait lancé à la planète entière en 1997. L'organisation avait alors décidé de donner 10 000 dollars à la personne ou l'organisme qui arriverait à décrypter un message codé avec une clé RC5 de 64 bits, le RC5 étant un algorithme de cryptage qui du coup n'a plus été considéré comme sur. Le codage se faisant sur 64 bits, il y avait donc au total 2^{64} clés possibles et différentes, ce qui représente pas moins de 18 446 744 073 709 551 616 possibilités !
L'organisation distributed.net a alors décidé de relever le défi, en se basant sur une méthode de brute force cracking distribuée. Pour effectuer cela, le nombre total de clés est divisé en de multiples petits paquets, qui sont ensuite envoyés à des ordinateurs clients. Ces clients calculent chacun les paquets de clés reçus.
Ils sont venus à bout du message codé en 2002 après avoir tester « seulement » 83% des possibilités (soit 15 268 315 356 922 380 000 combinaisons tout de même !) après 1726 jours, ce qui représente plus de 102 milliards de combinaisons testées par seconde ! Bien évidemment de nombreuses machines ont été utilisées pour arriver à un tel nombre, plus de 330 000 au total.

La seconde attaque que nous allons voir maintenant est l'attaque par dictionnaire.

✓ Attaque par dictionnaire

Principe :

Elle consiste à créer une base de données contenant des couples de mots de passe avec leurs empreintes puis de les comparer un à un. Soit l'empreinte est contenue dans la base et le mot de passe sera trouver, soit il n'y figure pas et dans ce cas l'attaque échouera.

Méthode :

Pour cela, il faut dans un premier temps créer et remplir la base de données appelée table qui contiendra les mots de passe à tester ainsi que leur empreinte pré-calculé. Ensuite, il suffit de la même manière que la force brute de comparer les empreintes pré-calculées à celle recherchée.
En pratique, les tables utilisées ne sont pas recréées à chaque utilisation, c'est d'ailleurs là que réside l'avancée de cette méthode : pour chaque mot de passe où l'empreinte est calculée, elle est stockée dans une table qui peut donc être utilisée une infinité de fois. C'est ainsi que des tables contenant les mots des dictionnaires de différentes langues ont été établis.

Amélioration :

Cette attaque peut être enrichie pour accroitre son efficacité. Une première méthode consiste tout simplement à utiliser des tables plus grandes et différentes tables.
Une deuxième méthode est d'effectuer une manipulation sur les mots de passe contenus dans la table, c'est-à-dire de comparer non seulement l'empreinte du mot de passe mais aussi des variantes, par exemple, les empreintes correspondantes au mot en minuscule, au mot en majuscule, au mot en partie en minuscule et en majuscule, au mot où certaines lettres ont été remplacées par des chiffres ressemblants (« 4 » pour le « A », « 3 » pour le « E »), au mot en arrière (la fin devient le début), au mot répété (deux fois le même mot, méthode qui est parfois, à tord, utilisée pour « renforcer » un mot de passe), au mot combiné avec des chiffres (mot de passe dit hybride).

Avantages/Inconvénients :

Cette méthode sera totalement efficace si le mot de passe recherché est présent dans les dictionnaires utilisés, cela peut paraitre limité mais de nombreuses personnes (à tord donc) utilisent des mots courants ou des prénoms pour se protéger. C'est pourquoi cette attaque est très répandue. Si elle ne peut concurrencer et remplacer l'attaque par force brute qui est la seule à être totalement efficace, elle en est parfaitement complémentaire. En effet, au lieu d'essayer naïvement toutes les combinaisons, il est alors plus courant d'utiliser, au premier abord, cette méthode pour réduire le nombre élevé de combinaisons.
Mais, il faut tout de même savoir que cette méthode à un coût, celui de devoir stocker ces tables qui peuvent atteindre des tailles colossales.

- Bonne efficacité
- Rapidité par rapport à la force brute

- Efficacité partielle
- Capacité de stockage importante
- Lenteur

Remarque :

Afin d'illustrer, la taille considérable d'espace de stockage nécessaire pour posséder une table la plus efficace possible, nous pouvons prendre l'exemple d'une empreinte SHA-1, fonction de hachage qui est composée de 20 octets. Le fichier contenant les empreintes de tous les mots de passe de 8 lettres (minuscules uniquement) serait de 56 000 Go, soit 112 disques durs de 500 Go ! Si la différenciation entre les minuscules et majuscules est appliquée, le fichier passe alors au million de gigaoctets, soit plus de 2 000 disques durs de 500 Go, en rajoutant les chiffres, la taille devient alors supérieure à 4 millions gigaoctets !
Cependant, cette méthode exclut de nombreuses combinaisons possibles (on utilise surtout des mots existants ou des prénoms), c'est pour cela qu'elle est largement utilisée, si les fichiers ne sont pas trop volumineux, ils sont relativement facile à stocker sur un seul disque dur. Par exemple, le dictionnaire de la langue anglaise contient environ 200 000 mots, leur empreinte SHA-1 se stockerait donc dans un « petit » fichier de 4 Mo.

Les méthodes présentées précédemment ont deux handicaps majeurs : soit un temps de calcul très long, soit un espace de stockage qui doit être important.
L'attaque qui suit dite par *rainbow table* (« table arc-en-ciel » en français) essaie justement de concilier un temps de calcul et des tailles de fichiers raisonnables tout en restant performant.

✓ **Attaque par** *rainbow table*

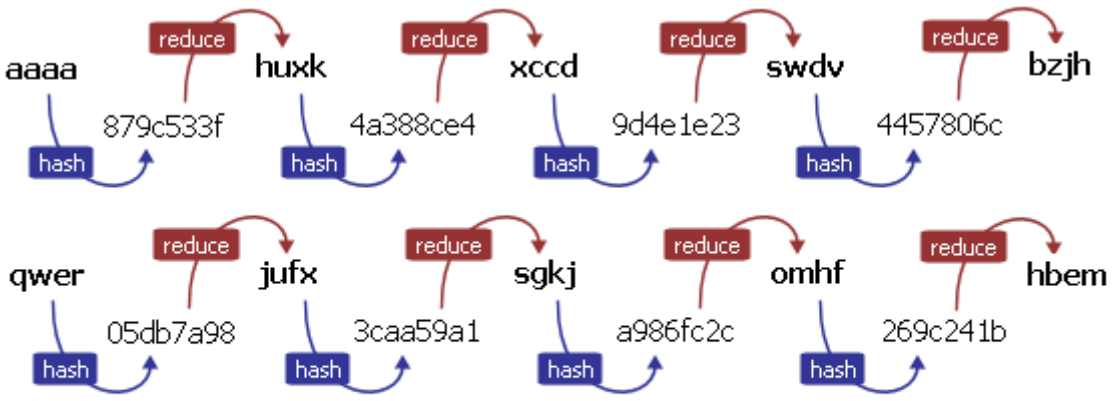
Principe :

Elle consiste à créer, comme pour l'attaque par dictionnaire, une base de données, les rainbow tables, mais contenant non plus des couples mot de passe/empreinte mais des couples constitués de deux mots de passe.

Méthode :

Il y a donc deux phases pour effectuer une attaque par rainbow table.

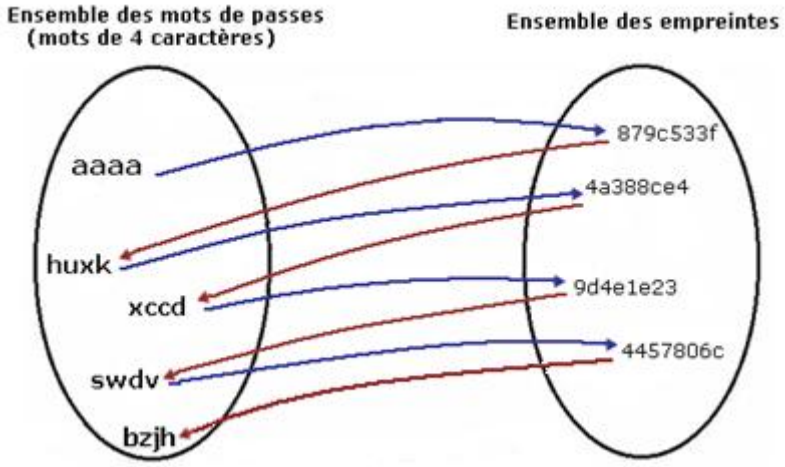
La première est la création de la table qui va servir de base pour pouvoir tester les mots de passe qui y sont stockés.



Rainbow Table

aaaa	bzjh
qwer	hbem

Comme l'expose le schéma ci-dessus (contenant deux lignes), ce n'est pas, comme pour l'attaque par dictionnaire, l'empreinte qui est gardé au côté du mot de passe mais un autre mot de passe. C'est justement l'astuce de cette méthode.

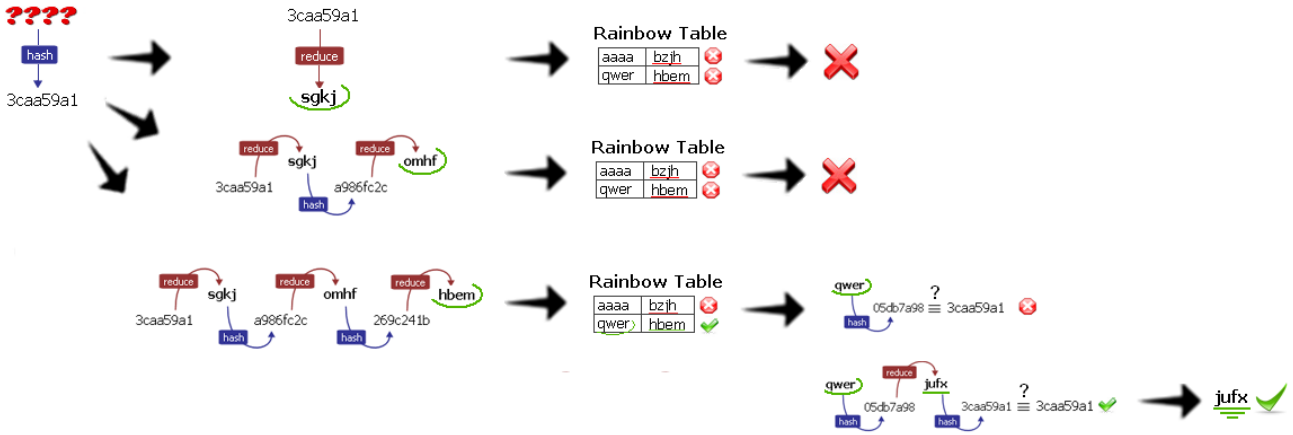


En effet, après avoir calculé l'empreinte, un nouveau mot de passe est associé à l'empreinte grâce à une fonction dite de réduction (« reduce » sur le schéma). Celle-ci est connue et doit être surjective c'est-à-dire retourner pour chaque empreinte passée en paramètre, un mot de passe qui doit bien sur être toujours le même. Cette fonction n'est évidemment pas la fonction réciproque de la fonction de hachage, elle n'existe pas ! Cette opération est répétée plusieurs fois. C'est ainsi qu'est obtenu le mot de passe qui est accolé au premier mot de passe dans la rainbow table.

La deuxième phase est la recherche du mot passe correspondant à l'empreinte. Pour cela, la comparaison s'effectue entre deux mots de passe et plusieurs étapes peuvent alors être nécessaires.
Premièrement, le mot de passe issu de la fonction de réduction est calculé puis recherché dans la rainbow table (deuxième colonne). S'il y figure, il suffit alors de répéter les mêmes opérations que pour la création de la table à savoir appliquer la fonction de réduction au mot de passe associé (première colonne) puis passer le résultat à la fonction de hachage, et cela le bon nombre de fois (le même qu'effectué lors de la création de la table moins un).

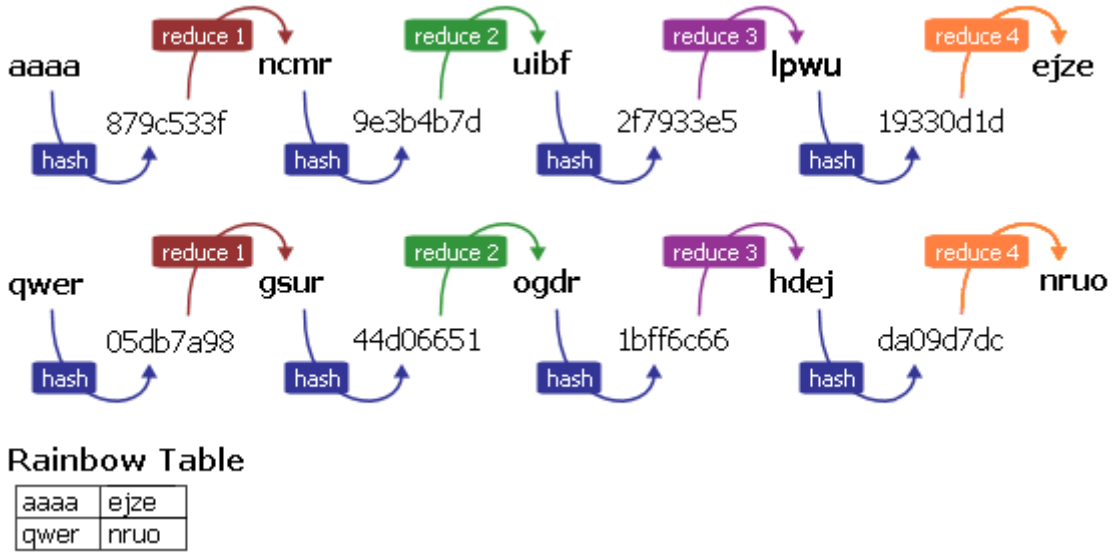
Prenons un exemple en rapport avec le schéma, cherchons à casser l'empreinte « 269c241b ». La première chose à réaliser, c'est de lui appliquer la fonction de réduction, le résultat trouvé est « hbem ». La deuxième action est la recherche dans la rainbow table : il s'y trouve et le mot de passe qui lui est associé est « qwer ». A ce nouveau mot de passe, la fonction de hachage lui est appliquée suivi de la fonction de réduction, cette opération est renouvelée, ici, trois fois, le mot de passe alors obtenu et recherché est « omhf ». Une vérification peut être faite en passant ce résultat à la fonction de hachage puis en comparant l'empreinte à celle de départ.

Si le mot de passe ne figure pas dans la table, il faut alors passer à une deuxième étape.
La fonction de réduction est alors appliquée non plus une seule fois mais deux fois : l'empreinte passe dans la fonction de réduction puis le mot de passe obtenu est haché et une nouvelle réduction est appliquée à l'empreinte. Ensuite de la même manière qu'à la première étape, le mot de passe est recherché dans la deuxième colonne de la table. S'il n'y figure pas, un nouveau couplage « hachage-réduction » est appliqué avant de rechercher ce nouveau mot de passe dans la table, et ainsi de suite...
Dans le cas où le mot de passe est trouvé, un premier hachage est effectué, l'empreinte est comparée à celle recherchée. Soit elles sont identiques et le mot de passe a été trouvé, soit elles sont différentes et une réduction, puis un hachage sont répétés autant de fois qu'il faut pour arriver à la bonne empreinte.

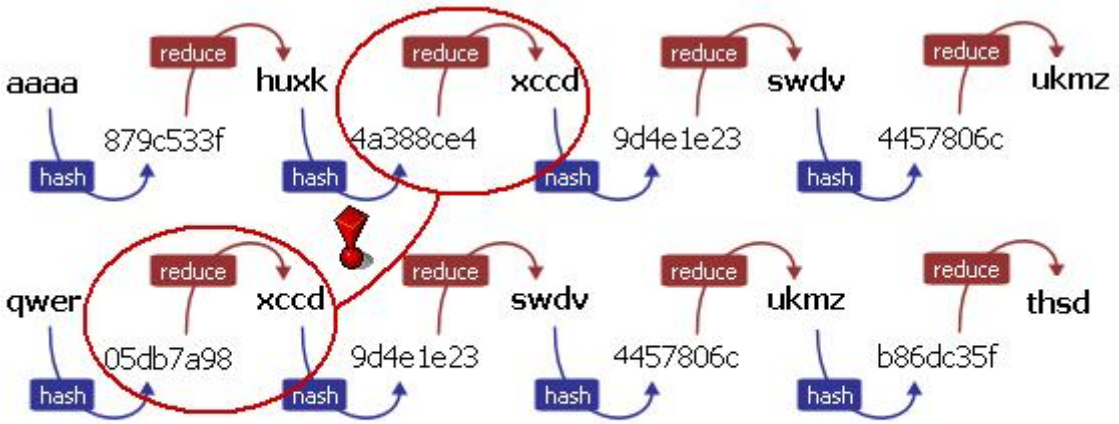


Pour expliciter, l'algorithme, déroulons un nouvel exemple, essayons de casser l'empreinte « 3caa59a1 ». Après une réduction, le mot de passe « sgkj » est trouvé mais il n'apparait pas dans la table, donc un couplage « hachage-réduction » est effectué sur ce mot, c'est « omhf » qui sort de cette opération mais il n'est toujours pas présent dans la table, un autre couplage « hachage-réduction » est appliqué. Cette fois, le mot de passe trouvé est « hbem » et il apparait dans la table, à la deuxième ligne. Le mot de passe qui lui est associé, « qwer » est récupéré, il passe alors dans la fonction de hachage, l'empreinte qui en découle est différente de celle à casser, elle est donc réduite puis l'empreinte trouvée est hachée, le résultat trouvé est ici identique. Donc l'empreinte a été cassée et le mot de passe correspondant est le résultat précédent autrement dit «jufx ».

Amélioration :



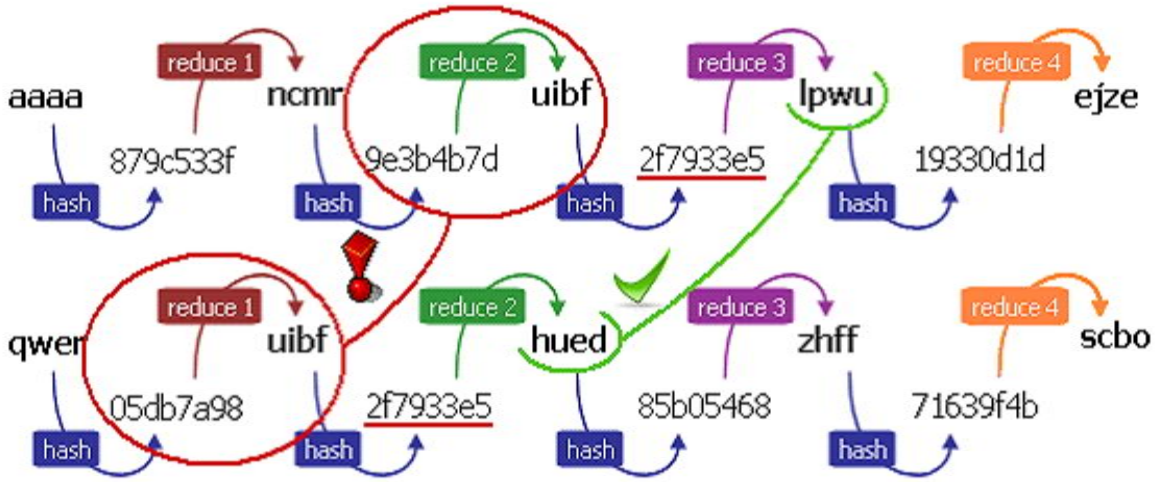
Cette attaque n'est en fait pas employée comme elle a été décrite précédemment. En pratique, il est utilisé non pas une mais plusieurs fonctions de réduction, autant que le nombre de fois qu'une réduction est appliquée par chaque mot de passe.



Rainbow Table

aaaa	ukmz
qwer	thsd

Cette amélioration a très vite enrichi cette attaque pour répondre aux problèmes des collisions. En effet, le nombre d'empreintes étant largement supérieur au nombre de mots de passe total, plusieurs empreintes ont à coup sur plusieurs mots de passe. Par conséquent, la table sera moins performante puisqu'elle permettra de retrouver moins de passe que la théorie le permet (ici, 6 au lieu de 8).



Rainbow Table

aaaa	ejże
qwer	scbo

En prenant, des fonctions de réduction différentes à chaque étape, cet effet de collision demeure mais ses conséquences sont diminuées.
En effet, sur l'exemple ci-dessus, il y a bien deux fois le même mot de passe (« uibf ») mais l'effet de cette collision n'affecte pas le reste de la « ligne ». Sept mots de passe différents sont retrouvables sur huit contre six pour la méthode à une seule fonction de réduction. De plus dans la réalité, les « lignes »

peuvent contenir des milliers de mots de passe, si toute une « ligne » est déjà contenue et donc dupliquée, cela engendre une perte de mots de passe et donc de performance.

Avantages/Inconvénients :

Cette méthode permet donc un compromis temps/espace de stockage tout en conservant une bonne efficacité. Cependant, il ne faut pas penser que tous les problèmes ou du moins inconvénients sont écartés, si la taille des fichiers peut être réduite, il ne demeure pas moins que pour qu'une telle attaque ait le plus de réussite possible, les tables utilisées doivent être très volumineuses. De même, en ce qui concerne le temps nécessaire, cette méthode permet d'accélérer les comparaisons mais elle peut tout de même être très longue. Enfin, l'inconvénient majeur de cette méthode est l'efficacité, certes elle reste bonne mais là aussi, elle n'est pas assurée, comme pour l'attaque par dictionnaire, et si le mot de passe n'est pas apparu lors de la création de la table, il n'y a aucune chance de casser l'empreinte qui lui correspond.

- Compromis temps/espace de stockage
- Bonne efficacité

- Efficacité partielle
- Capacité de stockage
- Lenteur

Remarque :

Afin d'illustrer, le compromis temps/espace de stockage, nous pouvons donner un exemple en utilisant toujours le SHA-1 comme algorithme de hachage.
Générons une rainbow table possédant 100 000 mots de passe auxquelles nous avons appliqué une réduction puis le couplage « hachage-réduction » afin d'obtenir 100 000 mots de passe différents. Une telle table permet donc retrouver 100 millions de mots de passe et donc de casser 100 millions d'empreintes.
Avec l'attaque par dictionnaire, il aurait fallu tout simplement stocker les 100 millions mots de passe autrement dit, nous aurions eu un fichier de 2Go. Alors que cette méthode réduit la taille du fichier à 2Mo puisqu'il contient 100 000 fois moins de mots de passe !
Cependant, rappelons-nous le nombre gigantesque de combinaisons : les (seuls) mots de passe de 8 caractères alphanumériques (minuscules et majuscule non différenciés) représentent, pas moins, de 2 821 109 907 456 combinaisons différentes !

Nous pouvons aussi remarquer sur cet exemple, la différence entre le nombre de mot de passe et le nombre d'empreintes possibles : il existe donc moins de $3x10^{15}$ mots passe différents alors qu'il existe 2^{160} soit $1,5x10^{48}$ empreintes !

➢ Salage

Nous savons que les mots de passe sont « hachés » pour obtenir une empreinte avant d'être stockés en dur. Mais ces mots de passe sont cassables assez facilement grâce aux algorithmes vus précédemment.
Il existe une façon de compliquer les choses pour ces algorithmes, en rajoutant ce qu'on appelle le salage. Le salage consiste à rajouter une séquence de bits à la fin du mot de passe afin de changer le mot de passe final et ainsi obtenir une nouvelle empreinte plus difficile à casser.

Il peut y avoir différents cas, si le salage n'est pas aléatoire, on peut retrouver le mot de passe de la même façon que vue précédemment (grâce aux algorithmes de cassage) mais on passera beaucoup plus de

temps avant de trouver le mot de passe.
En effet, le nombre de combinaisons à rechercher sera encore plus grand, car en plus du mot de passe, il faudra tenter toutes les combinaisons possibles pour la partie « salée » cela revient à calculer un nombre de combinaison égale à : *Nombre de combinaisons du mot de passe x Nombre de combinaisons du salage*.
Cela allongera donc considérablement le temps de calcul ainsi que la taille utilisée pour les dictionnaires ou les Rainbow Table.
Par contre, si le salage est aléatoire, le mot de passe sera bien plus difficile à découvrir.

Le désavantage du salage de mot de passe vient du fait de son temps de calcul plus long afin de déterminer l'empreinte lorsque l'on défini le mot de passe.

➢ Conclusion

Nous avons donc vu brièvement les algorithmes de hachage qui ont été mis en œuvre pour protéger nos mots de passe et les méthodes qui peuvent être utilisées afin de les casser.

Tandis que nous pouvions penser et croire naïvement qu'un mot de passe est quasiment introuvable (sauf si c'est le petit nom du chien de la famille), il est finalement loin d'être impossible voire même pas très difficile de casser un très grand nombre de mots de passe utilisés.

Cependant, en suivant certaines règles certes pas très naturelles ni très pratiques (comme mettre un caractère propre à sa langue), nous pouvons aisément prévenir la majorité des attaques et augmenter la protection de nos comptes.

Pour conclure, il faut rester extrêmement vigilant et attentif aux avancées de la cryptographie qui est sans cesse en mouvement, en recherche de faille et ce n'est finalement qu'une question de temps pour que la sureté d'un algorithme prétendue très robuste soit mise à mal.

➢ Références

Sites Web :

http://www.hsc.fr/

---> http://www.hsc.fr/ressources/breves/rainbowtables.html.fr

--->http://www.hsc.fr/ressources/articles/hakin9_edito_empreintes/GL-Edito_Hakin9-2-2007-empreintes.pdf

http://www.mieuxcoder.com

--> http://www.mieuxcoder.com/2008/01/02/rainbow-tables/

http://fr.wikipedia.org/

--> http://fr.wikipedia.org/wiki/Philippe_Oechslin

http://fr.wikipedia.org/wiki/Attaque_par_dictionnaire

http://en.wikipedia.org/

--> http://en.wikipedia.org/wiki/NTLM

http://abcdrfc.free.fr/rfc-vf/rfc1321.html#desc

http://technet.microsoft.com/en-us/magazine/2006.08.securitywatch.aspx

http://ditwww.epfl.ch/SIC/SA/SPIP/Publications/spip.php?article738

http://www.commentcamarche.net/contents/crypto/des.php3

http://www.lockdown.co.uk/?pg=combi

Cours :

http://home.hefr.ch/Schuler/cours/securite/9_PasswordCrack.ppt

Revues :

MISC numéro 2 - Avril 2002

http://www.hsc.fr/ressources/articles/mdp_misc2/

MISC numéro 5 Fevrier 2003

http://www.hsc.fr/ressources/articles/mdp_misc5/

Annexes

Algorithme de force brute en C testant toutes les possibilités de mots avec une longueur minimale et maximale pour le mot de passe recherché

```
void BruteForce(int LongMin, int LongMax)
{
  Const char ALL[105] =
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890<>,?;.:/!§*µù%$£¤¨+=
})]à@ç^\\_`è|-[({'#\"é~&";

  int Lg = LongMin - 1;
  int i, x, y;

  char* Buff = (char*) malloc(Lg); // Creer le Buff de Lg octet

  while(Lg != LongMax)
  {
    realloc(Buff, Lg); // Augmente la taille du buff
    int Nchar[Lg];
    for(i=0; i<=Lg; i++)
    {
      Buff[i] = ALL[0]; // Remplie le Buff du premier char
      Nchar[i] = 0;
    }
    while(Nchar[0] != 105)
    {
      for(x=0; x<=105; x++)
      {
        Buff[Lg] = ALL[x];
        Nchar[Lg] = x;
        printf("%s \n", Buff);
        nbMDP++;
      }
      for(y=Lg; y>=0; y--)
      {
        if((Nchar[y] == 105)&&(Nchar[0]!= 105))
        {
          Nchar[y] = 0;
          Nchar[y-1]++;
        }
        Buff[y] = ALL[Nchar[y]];
     }
   }
   Lg++;
 }
 free(Buff);
}
```