Mots de passe : hashage et salage, suffisant ?

Récemment, plusieurs cas de vol de mots de passe ont frappé des sociétés du net; problème : les mots de passe étaient parfois stockés en clair dans la BDD.

Bon, nous faisons tous des erreurs de jeunesse, mais je m'étonne toujours que des gros sites omettent encore de brouiller les mots de passe, et quand je dis brouiller, je ne parle pas de chiffrement (qui suppose qu'une opération de déchiffrement est possible), mais de **hashage et de salage** !

Je ne prétends pas être un expert en sécurité, mais je vais essayer de vous fournir une structure de base pour l'authentification par mot de passe de vos utilisateurs (pour les pressés, l'implémentation est en fin d'article 🙂).

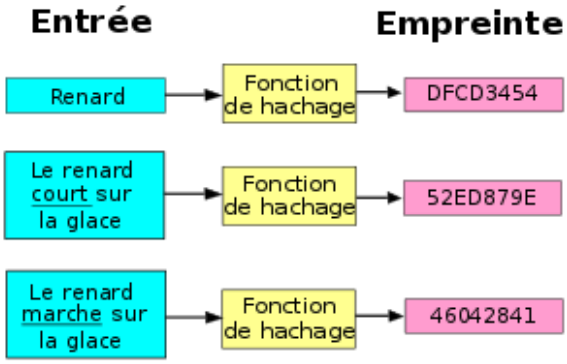
Pourquoi brouiller le mot de passe ?

Je pense que la nécessité de ne pas stocker un mot de passe en clair est évidente pour tout le monde : ainsi l'accès à votre BDD ne suffit pas à avoir les informations de connexion d'un utilisateur (et je ne parle pas juste de piratage : un administrateur/développeur malintentionné ne peut pas lire vos mots de passe via un SELECT).

Comment s'y prendre ?

En utilisant une fonction de hachage, c'est à dire (dixit wikipedia) :

> « une fonction particulière qui, à partir d'une donnée fournie en entrée, calcule une *empreinte* servant à identifier rapidement, bien qu'incomplètement, la donnée initiale »



La caractéristique principale d'une fonction de hachage en cryptographie est qu'**on ne peut pas facilement effectuer l'opération inverse** (à partir de DFCD3454, retrouver 'Renard') : exactement ce qu'il nous faut.

Il existe plein de fonctions de hachage (MD5, SHA-1, SHA-256, RIPEMD-160 …), mais certaines d'entre elles ne doivent plus être utilisées :

- MD5 ne doit plus être utilisé comme algorithme de signature (collision en quelques secondes)
- SHA-1 est aussi vulnérable à la collision
- SHA-256 et RIPEMD-160 sont pour l'instant OK

Ce n'est pas suffisant !

Et non …

Les pirates ont à leur disposition des dictionnaires appelés « **rainbow tables**« .

L'idée est très simple : Imaginons que je récupère une liste de mots de passe hashés en SHA-1. Que faire pour les forcer ?

Et si j'avais une gigantesque base de mots de passe générés automatiquement et déjà hashés en SHA-1, je pourrais comparer directement les hashs et en déduire le mot de passe ? C'est précisément le principe des rainbow tables : des tables précalculées de hashs qui permettent de trouver très rapidement le mot de passe initial.

Il faut donc rajouter une étape à notre algorithme …

Le salage

L'idée du salage est d'ajouter au mot de passe de l'utilisateur une chaîne aléatoire (le sel) qui va compliquer l'utilisation des rainbow tables sur le hash en multipliant le nombre de hash possibles pour un même mot de passe.

Un premier algorithme

Pour gérer nos mots de passe, il suffirait donc de :

- demander son mot de passe à l'utilisateur
- générer un sel
- le stocker
- l'ajouter au mot de passe
- hasher le sel+mot de passe
- stocker le résultat

- demander son mot de passe à l'utilisateur
- lire le sel enregistré
- l'ajouter au mot de passe
- hasher le sel+mot de passe
- comparer le résultat obtenu avec le hashage stocké

Bon … c'est ok sur le principe, ce type d'algorithme est utilisable et est même assez répandu, mais il reste un **gros problème** : les fonctions de hashage à notre disposition …

Le problème avec les fonctions de hachage

Le souci est que les fonctions de hachage disponibles sont **très rapides** ! Oui, vous avez bien lu : la rapidité est un défaut.

Les fonctions de hachage ont généralement comme objectif d'être rapide (notamment pour générer des signatures), mais dans le cas d'une authentification, **la vitesse est notre ennemi** !

Si l'algorithme est rapide, le crackage est rapide; même si on rajoute un sel, une attaque par dictionnaire peut donc suffire à craquer notre mot de passe (la puissance de calcul actuelle est telle qu'avec la bonne carte graphique vous pouvez génére 2,5 millions de SHA-1 en une seconde)

Alors, une solution ?

Oui, il y a une solution : l'algorithme **bcrypt**.

En vulgarisant un peu, il s'agit d'une fonction de hachage reposant sur l'algorithme Blowfish. Elle est conçue de façon à pouvoir être ralentie en augmentant le nombre de passage dans la fonction (workfactor).

Elle embarque la gestion d'un salage pour contourner les rainbow tables. Mais cette fois, le temps de crackage d'une clé dépassera la centaine de millisecondes (voire plus, en fonction des paramètres de bcrypt).

En conclusion

Le plus simple actuellement est de gérer les mots de passe avec bcrypt. En java, vous pouvez utiliser la bibliothèque jbcrypt, dont voici le bloc de dépendance maven :

view source

print?

```
1  < dependency >
```

```
2      < groupId >org.mindrot</ groupId >
```

```
3      < artifactId >jbcrypt</ artifactId >
```

```
4      < version >0.3m</ version >
```

```
5  </ dependency >
```

L'utilisation de la bibliothèque est assez simple (exemple issu du site de jbcrypt) :

view source

print?

```
1  // Hashage d'un mot de passe
```

```
2  String hashed = BCrypt.hashpw(password, BCrypt.gensalt());
```

```
3
```

```
4  // Il est possible d'augmenter la complexité (et donc le temps
```

```
5  // de traitement) en passant le "workfactor" en paramètre
```

```
6  // ici : 12 La valeur par défaut est 10
```

```
7  String hashed = BCrypt.hashpw(password, BCrypt.gensalt( 12 ));
```

```
8
```

```
9   // Vérification d'un mot de passe à partir du hash
```

```
10  if (BCrypt.checkpw(candidate, hashed))
```

```
11      System.out.println( "It matches" );
```

```
12  else
```

```
13      System.out.println( "It does not match" );
```

Il suffit de stocker le hash du mot de passe, et tout est géré : le sel est embarqué dedans ainsi que le « workfactor ».

Update

Pour info, voici quelques temps de hachage sur ma machine (Core i5 à 2,4 GHz) :

> workfactor = 5 – temps moyen = 4ms
> workfactor = 6 – temps moyen = 6ms
> workfactor = 7 – temps moyen = 13ms
> workfactor = 8 – temps moyen = 26ms
> workfactor = 9 – temps moyen = 51ms
> workfactor = 10 – temps moyen = 102ms
> workfactor = 11 – temps moyen = 217ms
> workfactor = 12 – temps moyen = 422ms
> workfactor = 13 – temps moyen = 820ms
> workfactor = 14 – temps moyen = 1650ms
> workfactor = 15 – temps moyen = 3259ms

A vous de moduler votre workfactor en fonction de vos besoins de qualité de service et de sécurité.

Comments are closed.

Geek et fier de l'être

Author WordPress Theme by Compete Themes