Ryx

Zeste de savoir

Les fonctions de hachage cryptographiques

12 août 2019

Table des matières

Les fonctions de hachage sont un outil informatique et mathématique devenu incontournable à tel point que vous utilisez plusieurs fonctions de hachage chaque jour sans forcément vous en rendre compte. Nous nous intéresserons ici aux fonctions de hachage cryptographiques.

Mais qu'est-ce qu'une fonction de hachage cryptographique ? Quelles sont les caractéristiques qu'une bonne fonction doit avoir ? Comment peut-on en attaquer une ?

On peut répondre à toutes ces questions, et on va même le faire .

1. Remarque préliminaire

Je parle ici majoritairement des fonctions de hachage cryptographiques, donc elles doivent avoir des contraintes niveau sécurité. Les fonctions de hachage peuvent bien entendu servir à plein de choses autres que de la cryptographie.

En cryptographie, un principe fondamental consiste à faire reposer la sécurité d'un cryptosystème sur sa complexité mathématique (principe de Kerckhoff).

> Contenu masqué n°1

En aucun cas une fonction doit être difficile à attaquer car son principe est gardé secret : les fonctions doivent être publiques. C'est le cas du RSA par exemple, qui fait reposer sa sécurité sur la difficulté de factoriser des grands nombres premiers.

C'est la même chose pour les fonctions de hachage cryptographiques : même si elles ne reposent pas sur un problème mathématique, elle doivent être publiques et ne pas faire reposer leur force sur le secret.

2. Qu'est-ce qu'une fonction de hachage ?

##Définition

Une fonction de hachage, c'est en fait tout simple. Cela consiste en une fonction qui transforme une donnée quelconque en une donnée de taille fixée. C'est tout.

Bien entendu, la donnée en question peut avoir plusieurs formes. Ce peut être du texte, une image, ... mais dans tous les cas la donnée sera transformée en un texte binaire avant qu'on lui applique la fonction de hachage.

Pour bien voir de quoi il s'agit, on va l'illustrer.

###Exemple

On va utiliser une fonction de hachage bidon : on prend une fonction qui renvoie les 64 premiers bits (les 8 premières lettres) de l'entrée. Ainsi, toute entrée donnera bien un message de taille fixée (64 bits), donc c'est bien une fonction de hachage.

Pour "Zeste de savoir" :

	En ASCII	**En binaire**	**En hexadécimal**
Message initial	Zeste de savoir	01011010 01100101 01110011 01110100 01100101 00100000 01100100 01100101 00100000 01110011 01100001 01110110 01101111 01101001 01110010	5a 65 73 74 65 20 64 65 20 73 61 76 6f 69 72
Haché	Zeste de	01011010 01100101 01110011 01110100 01100101 00100000 01100100 01100101	5a 65 73 74 65 20 64 65

###Exemple moins bidon

On va maintenant définir une fonction moins bidon qui nous servira pour des exemples dans la suite de ce tuto : h_{ex}

On pose h_{ex} à valeurs dans $[0, N-1]$ défini par : pour un mot x écrit en ASCII de longueur l,

$h_{ex}(x) = (\sum_{i=0}^{l-1} x[i]B^{l-i-1}) \mod N$

Avec B une puissance de 2 et N premier.

> Contenu masqué n°2

On va poser par exemple $B = 256$ et $N = 251$.

Cette fonction est une fonction de hachage sur 8 bits, car les valeurs prises peuvent aller de 0 à 251 (car on fait un modulo $N = 251$).

Pour "Zeste de savoir" :

	En binaire	**En hexadécimal**
Message initial (x)	01011010 01100101 01110011 01110100 01100101 00100000 01100100 01100101 00100000 01110011 01100001 01110110 01101111 01101001 01110010	5a 65 73 74 65 20 64 65 20 73 61 76 6f 69 72
Haché ($h_{ex}(x)$)	10001111	8f

##Dans la vraie vie

Il y a plein de fonctions de hachage.

En vrac, on peut nommer les MD : MD4 était utilisé il y a fort longtemps dans les années 90, MD5 était utilisé jusqu'à récemment. Les SHA : SHA-1 devrait être enterré pour de bon en 2018, SHA-2 et SHA-3 sont actuellement utilisées.

Quelques exemples de hachés avec différentes fonctions de hachage :

Message original	**MD4**	**MD5**	**SHA-1**
Zeste de savoir	7096e36e73cf1a30cd28b6	3f68686e417ee	a742e60e93971ee9c4a0764
Zeste de savoie	52d43d8cf1b544ba65b3fd	911ba6cf2aa	4f0cb684474cb8b85114(
Roger le tavernier	4ef1e80cfa57834cc856b9	27f04e07d929	085348a163728def59253d

On constate que les hachés font tous 32 caractères hexadécimaux (donc 128 bits).

Bien entendu, ces fonctions sont bien plus compliquées que les fonctions bidons que l'on a vu au dessus (et heureusement d'ailleurs, nous verrons pourquoi bientôt).

###Exemple

On va détailler un peu à quoi ressemble une de ces fonctions de hachage en prenant le MD5, qui est une fonction de hachage sur 128 bits.

Quelques notations pour le schéma issu de wikipédia ci-dessous :

- $[<<<]_s$ est une rotation de s bits vers la gauche
- $[+]$ est l'addition modulo 2^{32}
- $\oplus, \vee, \wedge, \neg$ sont les opérations booléennes xor, ou, et et non

Le hachage se passe comme suit :

- on effectue un bourrage si nécessaire pour que le message initial fasse un multiple de 512 bits
- on sépare le message en paquets de 32 bits : M_i

— pour chaque paquet M_i, on applique une opération de MD5 (ce qui est présenté sur le schéma ci-dessous) en utilisant les blocs A, B, C et D de l'itération précédente

A la première itération, on initialise les blocs A, B, C, et D avec des constantes.

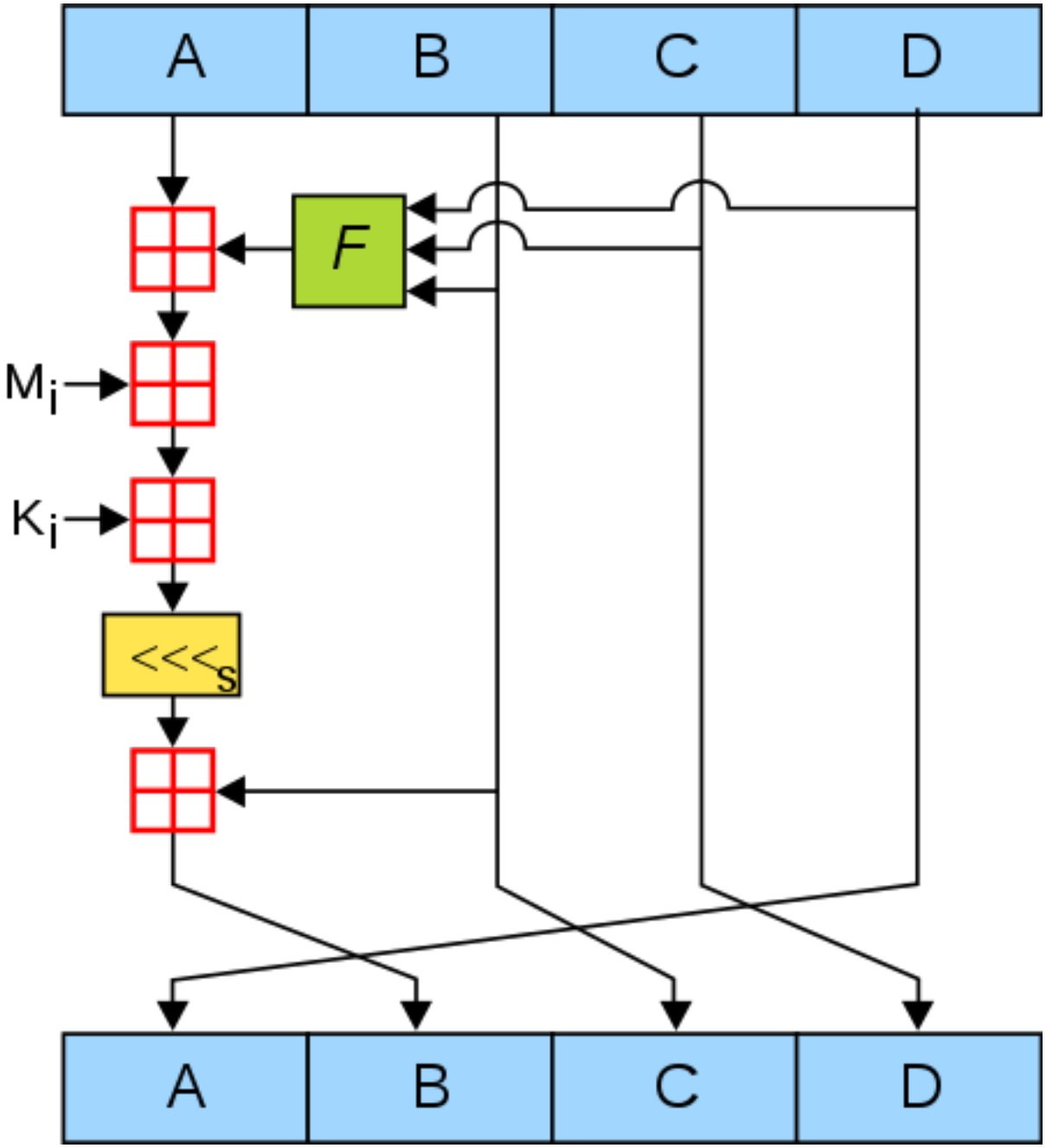


FIGURE 2. – Une opération de MD5. K_i est une constante et F une fonction non-linéaire qui varie - Wikipédia CC BY-SA

La fonction F est parmi les suivantes :

— $F_1(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$
— $F_2(B, C, D) = B \wedge D \vee (C \wedge \neg D)$
— $F_3(B, C, D) = B \oplus C \oplus D$
— $F_4(B, C, D) = C \oplus (B \vee \neg D)$

Pour obtenir le haché final, on fait la somme de tous les blocs A obtenus, la même chose pour B, C et D et on concatène les 4 blocs ainsi obtenus.

3. Collisions et pré-images

Tout d'abord, il y a quelques principes théoriques qu'il faut présenter avant de parler plus en profondeur des fonctions de hachage.

##Les collisions

Comme on l'a déjà dit auparavant, une fonction de hachage transforme un message de taille arbitraire en message de taille fixe. Il y a donc un problème : comme il y a plus de messages possibles que de hachés possibles, plusieurs messages peuvent avoir le même haché.

Contenu masqué n°3

Ainsi, si on a deux messages qui ont le même haché, on dit que l'on a une collision.

###Exemple :

Si on reprend notre fonction de hachage bidon de tout à l'heure, on peut trouver une collision facilement :

Message	Haché
Fonction de hachage	Fonction
Fonction de hachis parmentier au piment	Fonction

On a trouvé une collision. Et vous remarquez immédiatement que c'est très facile de trouver des collisions. Il suffit de trouver deux messages qui commencent par les mêmes 8 premiers caractères (mais ce peut être n'importe quels 8 caractères).

En revanche, trouver une collision sur h_{ex}, ça paraît tout de suite plus compliqué. Il faut trouver x et y tels que : $h_{ex}(x) = h_{ex}(y)$ Soit : $(\sum_{i=0}^{l-1} y[i]B^{l-i-1}) \mod N = (\sum_{i=0}^{l-1} x[i]B^{l-i-1}) \mod N$

##Les pré-images

Trouver une pré-image consiste à trouver, à partir d'un haché H, un message M dont le haché est H.

###Exemple

Si on se donne "Zeste de" comme haché avec notre fonction bidon, on peut trouver comme pré-image "Zeste de savoir", mais aussi "Zeste de gargle blaster pangalactic".

###Seconde préimage

Si l'on se donne un message M, trouver une seconde pré-image consiste à trouver un autre message M' tel que M et M' ont le même haché.

Par exemple, si on se donne "Zeste de savoir" comme message initial, une seconde préimage peut-être "Zeste de citron" par exemple, car ces deux messages ont même haché.

##La nuance à bien comprendre

Trouver une seconde préimage et trouver une collision sont deux choses très différentes :

- pour trouver une collision, on veut trouver **deux messages quelconques** qui aient le même haché
- pour trouver une seconde préimage d'un message M, on veut trouver **un seul message qui ait le même haché que M**

En pratique, trouver une seconde préimage est beaucoup plus compliqué que de trouver une collision, parce que l'on a pas le choix du premier message.

###Trouver une collision sur h_{ex}

Vu que l'on a le choix des mots, on va essayer d'utiliser des mots faciles pour que les hachés ne soient pas trop compliqués. On remarque par exemple que sur des exemples à une seule lettre, le haché est simplement la valeur ASCII de la lettre (ex : $h_{ex}('a') = 97$). Si on a un mot comme $'aa'$ dont le haché est 80, on regarde dans la table ASCII, et on voit que $h_{ex}('P') = 80$.

On a bien trouvé une collision.

4. Caractéristiques d'une bonne fonction de hachage

Je ne pense pas qu'il y ait de bonne ou de mauvaise situation[1]. En revanche, ce qui est certain, c'est que d'un point de vue cryptographique, il y a des bonnes et des mauvaises fonctions de hachage.

Vu que vous en utilisez tous les jours dans des protocoles de sécurité (carte bleue, paiements, signature électronique, etc...), il vaudrait mieux que les fonctions que l'on utilise soient bonnes. Nous allons donc voir les caractéristiques qui font une bonne fonction de hachage.

Il y a pour faire une bonne fonction de hachage cryptographique certaines contraintes majeures à respecter. Le problème, c'est qu'elles sont difficiles à définir formellement : c'est toujours relatif à la puisance de calcul par exemple.

##Résistance à la préimage

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. Cela revient à résister à la préimage.

Si on prend notre exemple de tout à l'heure, si on nous donne le haché ("Zeste de"), on peut trouver un message dont le haché est "Zeste de", comme par exemple : "Zeste de carbonade flamande", et ce très facilement.

##Résistance aux collisions

On arrive donc à la deuxième grosse condition pour avoir une bonne fonction de hachage : la résistance aux collisions.

Il ne s'agit pas qu'il n'y ait pas de collisions : il y en aura toujours. Il s'agit simplement qu'il n'y ait pas de solution facile pour fabriquer des collisions.

Par exemple, sur notre fonction bidon, il y a un moyen facile de trouver des collisions (prendre deux messages qui ont les mêmes 8 premiers caractères). On aimerait bien que pour trouver une collision, il n'y ait pas d'autre moyen que de hacher plein plein de message et de tester s'il n'y a pas deux fois le même résultat.

###L'attaque des anniversaires

Une fonction de hachage prend ses valeurs dans l'ensemble de entiers (on considère toutes les données comme des entiers), et est à valeurs dans un domaine défini précisément par le nombre n de bits maximal d'une image (par exemple, une fonction peut être définie sur 128 ou 256 bits). Ainsi, une fonction à valeurs dans 256 bits aura 2^{256} valeurs différentes possibles.

Si la fonction de hachage est définie sur n bits, on aura très probablement une collision après avoir calculé $2^{n/2}$ hachés. C'est ce que l'on appelle l'attaque des anniversaires. Cette attaque consiste simplement en la chose suivante : hacher $2^{n/2}$ messages différents comme un bourrin et regarder si deux messages ont le même haché (il y a une forte probabilité pour que ça arrive).

Contenu masqué n°4

L'idéal, ce serait donc qu'il n'y ait pas de meilleure solution que de faire l'attaque des anniversaires.

Cela s'appelle l'attaque des anniversaires en référence à un résultat de probabilités un peu contre-intuitif (article wikipédia).

L'attaque des anniversaires permet de trouver une collision, et non pas une seconde préimage. Pour trouver une préimage naïvement, c'est beaucoup plus compliqué : il faut faire une recherche dans les messages (il y en a une infinité) jusqu'à ce que l'on en trouve une qui fonctionne.

###Les attaques sur les fonctions de hachage

On appelle attaque sur une fonction de hachage un procédé qui permet de trouver une collision en calculant moins de hachés que ce que l'on ferait avec l'attaque des anniversaires. Ainsi, si on parle d'une attaque de complexité 2^p, c'est qu'il faut calculer 2^p hachés en moyenne pour obtenir une collision.

On a trouver comment faire des collisions sur h_{ex} par exemple, donc elle n'est pas résistante aux collisions.

##Avec des vraies fonctions de hachage

Sur des vraies fonctions de hachage, c'est très difficile de trouver des collisions, car ces fonctions sont beaucoup plus complexes que celles que l'on a pu vois pour l'exemple. Si vous faites un

petit tour sur la page wikipédia du SHA-1 par exemple, vous verrez que ce n'est pas si simple à comprendre.

D'ailleurs, le Sha-1 a fait son temps car cette fonction de hachage n'est plus sécurisée. Pour l'historique, le Sha-1 est une fonction de hachage cryptographique de la NSA sur 160 bits, et depuis déjà quelques années, on remettait en cause la fiabilité de cette fonction. Le 23 février 2017, Google (en collaboration avec le CWI, des chercheurs néerlandais) à annoncé avoir trouvé une collision.

Les conséquences sont que le Sha-1 va être progressivement remplacé par d'autres fonctions plus sécurisées, comme le Sha-2 ou Sha-3 (très originaux comme noms), ou encore bcrypt qui, considérée comme une des meilleures fonction de hachage à ce jour, à l'avantage d'être adaptative (plus d'infos ici). En outre, chrome n'accepte plus les certificats Sha-1 et Firefox a prévu de les arrêter fin 2017.

Il y a un exemple de deux pdf ayant le même Sha-1 mais étant différents ici . Hint pour les daltoniens : l'un a un en-tête rouge, l'autre bleu.

###L'attaque

L'attaque des anniversaires ne fonctionne évidemment pas, car 2^{80} ($= 2^{160/2}$), ça fait tout de même cent mille milliards de milliards, et rien que pour stocker les résultats, il faut la bagatelle de cent millions de pétaoctets.

Concernant l'attaque utilisée sur le Sha-1, Wang Xiaoyun (une informaticienne, oui, ça existe, c'est quoi ces préjugés ?) avait trouvé en 2005 une attaque en 2^{69}. C'est beaucoup mieux que le 2^{80} ($=2^{n/2}$ de l'attaque des anniversaires), mais c'est compliqué à mettre en pratique (surtout avec la puissance de calcul de 2005), mais ça explique que le Sha-1 était déjà à l'agonie même si on avait pas trouvé de collision. Mais avec SHAttered en 2^{63}, on a fini par trouver une collision.

Pour résumer, l'utopie d'une fonction de hachage cryptographique est qu'il n'y ait pas d'attaque plus efficace que celle des anniversaires ($2^{n/2}$).

##Autres caractéristiques

Voici quelques autres caractéristiques qui ne sont pas aussi fondamentales que les deux précédemment énoncées, mais qui sont tout de même indispensables pour les fonctions de hachage cryptographiques.

###Effet avalanche

C'est une caractéristique de l'algorithme de hachage qui en plus aide à la résistance à la pré-image. Elle décrit le fait qu'il y ait des modifications de plus en plus importantes au fur et à mesure que l'on avance dans l'algorithme.

Si l'on respecte ce critère, deux données qui sont très semblables auront des hachés totalement différents.

####Exemple

Voici deux textes très proches (1 seul bit de différence), mais leur SHA-1 sont très différents.

Texte initial	Haché (SHA-1)
Salut, et encore merci pour le poisson.[2]	1a7d9145a4ece2a416df65d766942aac85408449

Salut, et encore merci pour le poisson/	a6164ea2e82df86a7f1ff94da0dc36510b34f096

###Rapidité de calcul

Un haché doit être rapide à calculer, mais pas trop :

— il faut que ça soit rapide parce qu'on l'utilise beaucoup, et que l'on a pas envie d'attendre à chaque fois que l'on veut hacher quelque chose
— mais pas trop quand même pour éviter que les attaques par brute-force ne soient trop efficaces

5. A quoi sert une fonction de hachage en cryptographie ?

Voici quelques exemples d'utilisation de fonction de hachage. Il y a bien entendu une multitude d'utilisations, mais nous n'en présentons que quelques-unes.

##La signature électronique

Si vous n'êtes pas familliers avec les notions de clés cryptographiques, faites donc un tour sur le tutoriel sur le RSA .

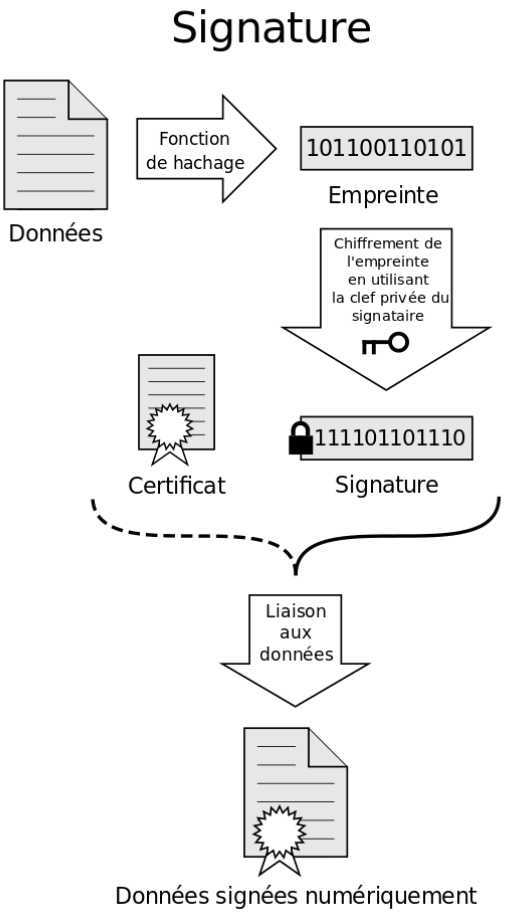
Si l'on veut garantir l'authenticité de l'émetteur d'un document, on utilise ce que l'on appelle une signature électronique. Par la suite, on appelle H la fonction de hachage. On appellera C l'action de chiffrer et D déchiffrer.

— A veut envoyer un message M à B
— A crée un haché du document H(M)
— A chiffre H(M) avec sa clé privée : C(H(M))
— A envoie M et C(H(M))
— B récupère tout ça
— B crée un haché du document H(M)
— B utilise la clé publique de A et compare H(M) à D(C(H(M)))

Alors, si c'est bien A qui a envoyé les documents, D(C(H(M))) = H(M) et B est certain de l'émetteur.

1. Asterix et Obelix : mission Cléopâtre
2. H2G2

5. A quoi sert une fonction de hachage en cryptographie ?



Résumé sur l'image ci-dessous (source : wikipédia) :

Imaginez que nous n'ayons pas utilisé de fonction de hachage, alors avec un document très gros on devrait chiffrer et déchiffrer des documents énormes, transmettre beaucoup plus de données, ... Avec une fonction de hachage, on a toujours on manipule des messages de taille fixes.

###Quel peut être le problème d'une collision ?

Le problème, c'est que l'on peut faire des attaques très vilaines en partant d'une collision. Un exemple sera plus parlant qu'un long discours :

M crée deux documents d1 et d2 ayant le même haché (c'est une collision). M envoie le document d1 à A, qui accepte et signe le haché et envoie la signature à M.

M attache la signature de d1 à d2, et envoie le d2 à B.

B croit alors que A a signé le document d2, alors que ce n'est pas vrai (A a signé d1).

####Exemple avec la fonction bidon

Cet exemple n'est pas du tout réaliste, mais c'est pour bien comprendre le principe.

— Le document d1 est le suivant : "Tu me dois 10€"
— Le document d2 est le suivant : "Tu me dois 10000000€"

Ils ont le même haché par la fonction bidon ("Tu me do").

M envoie d1 à A, qui est d'accord pour payer 10€. A signe donc le document en ajoutant sa signature S. Cela donne l'ensemble d1+S qui dit que A est d'accord sur les termes du document d1.

Mais alors, M signe alors d2 avec S et obtient d2+S avec S qui est une bonne signature de A sur d1, mais aussi sur d2 car d1 et d2 ont les mêmes hachés.

Maintenant, d'après le document d2+S que M possède, A est d'accord pour payer 10000000€, ce qui n'est évidemment pas le cas.

##Les mots de passes

Autre exemple : quand on gère un système avec des mots de passes, sur un site web par exemple. On utilise en général des bases de données, mais on ne veut surtout pas que les mots de passes soient en clair dans la base de données. Sinon, si quelqu'un réussit à accéder aux bases de données, il a tous les couples identifiants / mots de passes, et c'est vraiment très mauvais.

On va alors stocker tous les hachés des mots de passes.

— Pour s'identifier, l'utilisateur entre son login et son mot de passe. Le site web va alors hacher le mot de passe entré et le comparer au haché présent dans la base de donnée. Si les deux coïncident, c'est bon et l'utilisateur est bien identifié.
— Si un intrus accède aux bases de données, il a accès aux hachés, et ça ne lui sert à rien, car s'il entre un couple identifiant/haché qu'il a trouvé dans la base de données, le haché sera re-haché et ne coïncidera plus.

Donc si vous avez un compte sur un site qui est capable de vous redonner votre mot de passe en clair si vous l'avez oublié, fuyez, pauvres fous, car il a les mots de passe en clair dans une base de donnée !

###Ajoutons du sel

Question : est-ce que c'est suffisant pour que ce soit à peu près sécurisé ?

Réponse : NON !

Question : Mais qu'est-ce que pourquoi de comment ?

Mettons-nous dans la peau d'un méchant vilain pirate intergalactique et supposons avoir accès aux bases de données. Les gens ont une mémoire de poisson rouge, donc ils oublient leurs mots de passe si ceux-ci sont trop compliqués. Conclusion : les gens utilisent des mots de passes bidon.

Prenons les 100 mots de passes les plus utilisés (”123456”, ”password”, ”12345”, ”12345678”, qwerty”, ”123456789”, ”1234”, ”baseball”, ”dragon”, ”football”, ...). Il y aura au moins une personne assès stupide dans les quelques milliers d'utilisateurs qui sont enregistrés dans la base de donnée pour avoir utilisé un tel mot de passe.

Il suffit alors de hacher tous ces 100 mots de passe bidons en utilisant la fonction de hachage du site, et de regarder si quelqu'un dans la base de données a un haché qui correspond à l'un de ces 100 hachés.

Si oui, alors on a son mot de passe, et on a gagné, et lui a perdu.

####Un peu de sel dans votre hachis ?

C'est ici qu'intervient le sel. Il s'agit d'une donnée en plus que l'on ajoute pour éviter les attaques commes celles que l'on vient de voir.

Le sel est une chaîne de caractères aléatoire, donc différente pour chaque utilisateur que l'on concatène avec le mot de passe avant de le hacher.

Sont alors stockés dans la base de données : identifiant + hachage(mot de passe+sel) + sel

Pour s'authentifier, si l'utilisateur rentre son mot de passe, le sel est récupéré, et on peut faire un test de hachés comme d'habitude. Aucun changement du côté utilisateur n'est donc à déplorer.

Ce qui change, c'est pour notre pirate galactique :

— avant, il fallait calculer 100 hachés **au total**
— maintenant, vu que chaque utilisateur a un sel différent, il faut calculer 100 hachés **pour chaque utilisateur**, ce qui est bien plus coûteux

En pratique, notre pirate préféré utilise une table arc-en-ciel (qui contient non pas 100 mais des milliers de mots de passes). Dans le cas sans sel, il y a un seul calcul de haché à faire par mot de passe, que l'on réutilise pour chaque utilisateur. Dans le cas du sel, il y a un calcul de haché à faire par mot de passe et par utilisateur, ce qui est bien plus coûteux.

i

On pourrait se dire que le calcul est très rapide à faire, mais si on a une bonne fonction de hachage, les calculs vont être suffisamment lents à calculer pour que le brute-force ne fonctionne pas.

Conclusion : notre pirate s'est fait rouler dans la farine par du haché au sel (bon on va arrêter là avec les métaphores culinaires).

6. Quelques outils de hachage supplémentaires

Voici quelques outils en bonus pour la culture générale. On ne développera donc pas les exemples.

##La construction de Merkle-Damgard

Le but est le même que pour les fonctions de hachage : on veut éviter les collisions et les attaques sur les préimages. Le fonctionnement est cependant un peu différent (voir la page wikipédia).

On dispose de :

— une fonction de compression à deux entrées f qui doit être résistante aux collisions et à la préimage.
— un vecteur d'initialisation v
— un message découpé en blocs (et complété si besoin par du bourrage) $b_1, b_2, .., b_n$

Alors : On calcule $r_1 = f(v, b_1)$, puis $r_2 = f(r_1, b_2)$, $r_3 = f(r_2, b_3)$, ... et $r_n = f(r_{n-1}, b_n)$. Le haché est alors r_n (on peut éventuellement effectuer une dernière opération dessus avant).

##MAC

Le code d'authentification de messages (MAC : Message Authentication Code) est lui aussi un peu différent des fonctions de hachage. On l'utilise entre autres pour vérifier qu'un message est bien intègre, et n'a pas subi de modifications.

La principale différence avec les fonctions de hachage est l'utilisation d'une clé secrète.

Par exemple, le CBC-MAC est un algorithme MAC utilisant du chiffrement par blocs (CBC : cipher block chaining)

Voilà, j'espère que vous y voyez un peu plus clair concernant les fonctions de hachage.

Ceci n'était qu'un aperçu sans trop de math, donc pour ceux qui veulent voir à qui ressemble la théorie des fonctions de hachage, voici un ads de 2008 .

De plus, il y a plein d'autres applications utilisant des fonctions de hachage non cryptographiques (tables de hachage pour gérer des données, filtre de Bloom en bioinformatique, ...).

Merci pour votre lecture !

Contenu masqué

Contenu masqué n°1

On pourrait nuancer l'obligation de suivre les principes de Kerckhoff, mais ils sont encore largement suivis, donc on va partir du principe qu'ils sont quand même nécessaires. Retourner au texte.

Contenu masqué n°2

On préfère prendre N premier pour éviter des collisions "arithmétiques". Retourner au texte.

Contenu masqué n°3

Si une fonction de hachage transforme un message quelconque en message de 64 bits, il y a $2^{64} = 18446744073709552000$ possibilités. Mais comme il y a une infinité de messages possibles en entrée, il y a forcément plusieurs messages qui donnent le même haché : c'est le principe des tiroirs/pigeons/chaussettes. Retourner au texte.

Contenu masqué n°4

$2^{n/2}$ est appelée la valeur limite de l'anniversaire (birthday bound dans la langue de Shakespeare). C'est mathématiquement le nombre moyen de messages différents à hacher pour obtenir une collision. Retourner au texte.