Cryptographie
Fonctions de hachage et contrôle d'intégrité

Carlos Aguilar
carlos.aguilar@enseeiht.fr

IRIT-IRT

Plan

Les fonctions de hachage cryptographiques

Tiré de `http://www.unixwiz.net/techtips/iguide-crypto-hashes.html`

Définition

Fonction déterministe connue de tout le monde qui à partir d'une entrée de taille arbitraire calcule une empreinte de taille fixe.

En pratique

$ cat smallfile

```
This is a very small file with a few characters
```

$ cat bigfile

```
This is a larger file that contains more characters.
This demonstrates that no matter how big the input
stream is, the generated hash is the same size (but
of course, not the same value). If two files have
a different hash, they surely contain different data.
```

$ ls -l empty-file smallfile bigfile linux-kernel

```
-rw-rw-r--    1 steve    steve           0 2004-08-20 08:58 empty-file
-rw-rw-r--    1 steve    steve          48 2004-08-20 08:48 smallfile
-rw-rw-r--    1 steve    steve         260 2004-08-20 08:48 bigfile
-rw-r--r--    1 root     root      1122363 2003-02-27 07:12 linux-kernel
```

$ md5sum empty-file smallfile bigfile linux-kernel

```
d41d8cd98f00b204e9800998ecf8427e  empty-file
75cdbfeb70a06d42210938da88c42991  smallfile
6e0b7a1676ec0279139b3f39bd65e41a  bigfile
c74c812e4d2839fa9acf0aa0c915e022  linux-kernel
```

Propriétés (uniformité)

C'est une opération en sens unique

MD5 : 128 bits $\rightarrow 2^{128}$ hachés (hashes, digest, checksums) possibles pour une infinité d'entrées (pour chaque haché il y a une infinité d'antécédents)

Ils sont distribués aléatoirement même pour des messages liés : soit m un message quelconque et h son haché, pour un message $m' \neq m$ même en essayant de le choisir pour retomber sur h on a $\Pr(\mathrm{MD5}(m') = h) \simeq 1/2^{128}$

Exemple : l'effet avalanche

$ cat file1
```
This is a very small file with a few characters
```

$ cat file2
```
this is a very small file with a few characters
```

$ md5sum file?
```
75cdbfeb70a06d42210938da88c42991  file1
6fbe37f1eea0f802bd792ea885cd03e2  file2
```

Changer un bit dans le message d'entrée change complètement la sortie

Fonction de hachage cryptographiquement sûre

Définition

C'est une fonction de hachage pour laquelle il n'y a pas d'algorithme en temps polynomial et avec une probabilité de succès non négligeable permettant de :

- à partir d'une empreinte, trouver un antécédent (*first preimage resistance*)
- à partir d'un antécédent, trouver un autre antécédent donnant la même empreinte (*weak collision resistance*, ou *second preimage resistance*)
- trouver deux antécédents avec une même empreinte quelconque (*strong collision resistance*)

Intérêt

On ne veut pas que la fonction soit inversée
On veut être certain que si le haché n'a pas changé, le message non plus

Même si on est face à quelqu'un de malicieux

Objectifs de sécurité pour une fonction de k bits

Non inversibilité

Haché $\Rightarrow 2^k$ essais pour trouver un antécédant

Résistance faible aux collisions

Antécédent $\Rightarrow$ Haché $\Rightarrow 2^k$ essais pour trouver un autre antécédant

Résistance forte aux collisions

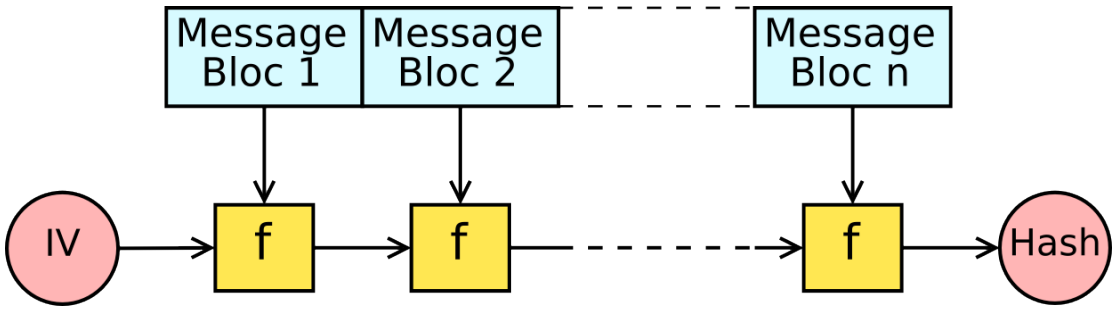
Paradoxe des anniversaires $\Rightarrow 2^{k/2}$ essais pour trouver deux messages avec un antécédant

Paradoxe des anniversaires

Quand on tire uniformément dans un ensemble de N éléments, dès qu'on a tiré de l'ordre de $\sqrt{N}$ on a une chance de l'ordre de $1/2$ d'avoir deux éléments identiques

Construction classique

Construction de Merkle-Damgård



Source : `https://fr.wikipedia.org/wiki/Construction_de_Merkle-Damgard`

Tout dernier bébé du NIST

SHA-3 suit une nouveau modèle (sponge construction) ...

Fonctions standards

La famille SHA (Secure Hash Algorithm)

- SHA-0 (1993-1995, plus utilisé)
- SHA-1 (1995-aujourd'hui, très utilisé)
  - 160 bits de sortie
  - Cassé, collisions en 2^{60} (pas de collision publique)
- SHA2 (2002-aujourd'hui, utilisé)
  - plusieurs versions (SHA-224, SHA-256, SHA-384, SHA-512, ...)
  - seulement fragilisées si exécutées sur 31/64 rounds
- SHA3 (standard depuis le 08/2015)
  - Ne remplace pas SHA2, le but est de varier les constructions
  - même versions (SHA3-224, SHA3-256, SHA3-384, SHA3-512, ...)
  - plus taille variable SHAKE128(M,d), SHAKE256(M,d) (256 et 512 bits ! ! !)

Autres

MD5 (128 bits, collision en 2^{18}) mais utilisé parfois
MD2, MD4, RIPEMD, Whirpool

Plan

Vérification d'intégrité

Téléchargement d'un logiciel

- ProFTPD -

Highly configurable GPL-licensed FTP server software

MD5 sums and PGP signatures of release files

MD5 Digest Hashes

```
417e41092610816bd203c3766e96f23b proftpd-1.2.8p.tar.bz2
abf8409bbd9150494bc1847ace06857a proftpd-1.2.8p.tar.gz
7c85503b160a36a96594ef75f3180a07 proftpd-1.2.9.tar.bz2
445fbf24e2ec300800a184eb81296bda proftpd-1.2.9.tar.gz
d834bb822816a2ce483cc2ef1a9533e7 proftpd-1.2.10rc3.tar.bz2
1e306d2f54ea92895ecff6659498b911 proftpd-1.2.10rc3.tar.gz
```

Si on ne considère pas les attaques malicieuses un code correcteur ou fonction de hachage classique suffit (propriétés statistiques équivalentes)

Variante pour le stockage

- On crée les hachés (par ex. des fichiers importants de notre OS) et on les garde en lieu sûr (rempl. du canal sûr).
- Quand on veut vérifier que notre OS n'est pas vérolé on recalcule les hachés et on les compare à ceux stockés en lieu sûr.

Vérification d'intégrité : sécurité

Face à une attaque après le premier hachage

L'antécédent et l'image sont déjà fixés, il faut trouver un autre antécédent $\Rightarrow$ Weak collision resistance

Par exemple pour une attaque lors du téléchargement ou de la restauration

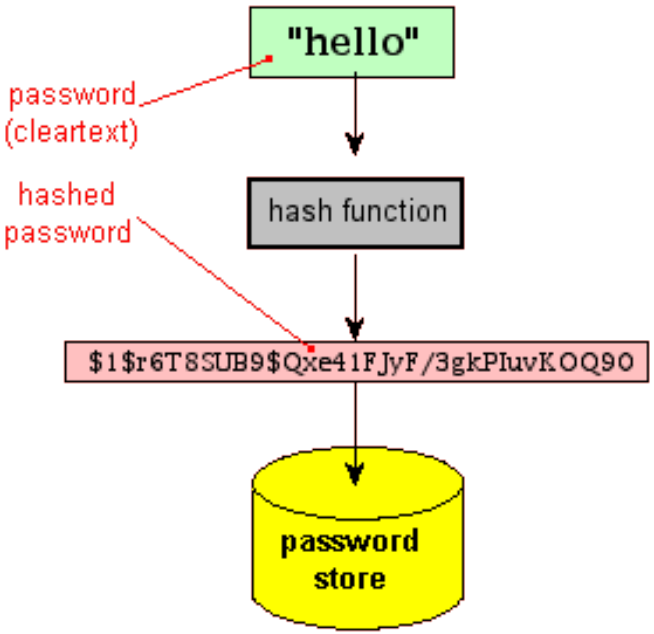
Face à une attaque avant le premier hachage

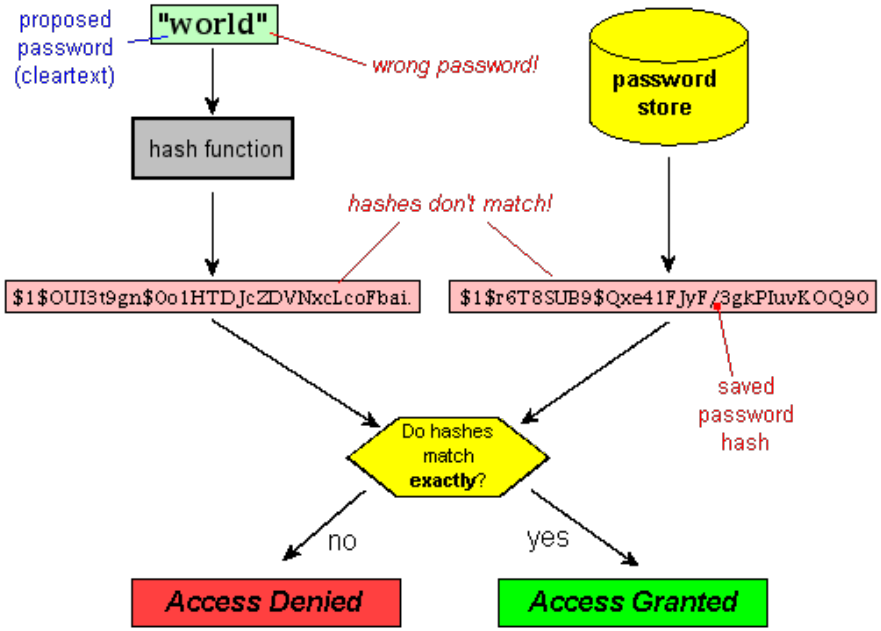
L'attaquant peut créer deux antécédents de son choix donnant le même haché $\Rightarrow$ Strong collision resistance

Par exemple, un attaquant peut créer un couple anodin/vérolé avec un même checksum. Il envoie généralement le bon (qui est vérifié par la communauté) et quand il veut il envoie le vérolé sans que le changement soit détecté par une vérification de haché.

Stockage de mots de passe

Stockage/Vérification





N'a pas un grand intérêt si les mots de passe ont une faible entropie

Sécurité

On veut être sûr qu'il est pas possible de calculer un antécédent. Sinon un attaquant accédant à la base des hachés peut obtenir un mot de passe (l'antécédent) qui passera le test du hachage.

Stockage de mots de passe : attaques

Types de mot de passe

```
loweralpha-numeric = abcdefghijklmnopqrstuvwxyz0123456789
mixalpha-numeric   = abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
ascii-32-95        =!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ\[]^_`%
                     abcdefghijklmnopqrstuvwxyz{|}~
```

Types d'attaque

Attaques par dictionnaire tout hacher voir si on tombe sur le résultat
Tables pré-calculées $\rightarrow$ recherche en temps constant (temps $\rightarrow$ mémoire)

Rainbow tables : ruse pour chercher en temps ℓ stocker en mémoire n une table pré-calculée de taille $n * \ell \Rightarrow$ loweralpha-numeric^{14} en qqs minutes

Utilisation d'un sel (salt)

On ne hache pas directement les mots de passe, on concatène un sel aléatoire qu'on garde en clair à côté du hachage

Engagements

Se mettre d'accord sur un aléa

Alice : $alea_1 \leftarrow \{0,1\}^k$, Alice to Bob : $alea_1$,
Bob : $alea_2 \leftarrow \{0,1\}^k$, Bob to Alice : $alea_2$
Qui peut tricher ?
Comment le réparer ?

Tirer à pile ou face électroniquement [2]

Enchère aveugle sans tiers de confiance

Au tableau !

Engagements

Se mettre d'accord sur un aléa

Alice : $alea_1 \leftarrow \{0,1\}^k$, Alice to Bob : $alea_1$,
Bob : $alea_2 \leftarrow \{0,1\}^k$, Bob to Alice : $alea_2$
Qui peut tricher ? Bob
Comment le réparer ?
Au début Alice to Bob : $CSHF(alea_1)$ puis à la fin, Alice to Bob : $alea_1$

Tirer à pile ou face électroniquement [2]

Enchère aveugle sans tiers de confiance

Au tableau !

Engagements

Se mettre d'accord sur un aléa

Alice : $alea_1 \leftarrow \{0,1\}^k$, Alice to Bob : $alea_1$,
Bob : $alea_2 \leftarrow \{0,1\}^k$, Bob to Alice : $alea_2$
Qui peut tricher ? Bob
Comment le réparer ?
Au début Alice to Bob : $CSHF(alea_1)$ puis à la fin, Alice to Bob : $alea_1$

Tirer à pile ou face électroniquement [2]

Alice : $b \leftarrow \{0,1\}$, $mask \leftarrow \{0,1\}^k$, Alice to Bob : $CSHF(mask\|b)$, Bob : $b' \leftarrow \{0,1\}$, Bob to Alice : b', Alice to Bob : $mask$

Enchère aveugle sans tiers de confiance

Au tableau !

Autres

Preuves de travail

Hashcash : pour qu'un mail soit envoyé il lui faut une en-tête dont le haché commence par 20 bits à zéro (en changeant un champ avec un index)

C'est le même principe qui est utilisé dans le mining des bitcoins

Dérivation de clés

A partir d'un secret maître générer une arborescence de clés
Pour des questions de hiérarchie
Pour des questions de renouvellement

Les détails font intervenir d'autres primitives $\Rightarrow$ on y reviendra

Signatures électorniques

On ne signe pas des messages mais des hachés de messages

Les détails font intervenir d'autres primitives $\Rightarrow$ on y reviendra

Plan

Les MAC (1/2)

Les MAC (Message Authentication Code)

Série de bits permettant de vérifier qu'un message :
- a été généré par quelqu'un connaissant une clé secrète
- n'a pas été modifié (réseau ou dispositif de stockage)

$\rightarrow$ garant de l'intégrité/authenticité mais attention au rejeu !

Principe

Au lieu d'envoyer m on envoie $m||t$, avec $t = MAC(sk, m)$
À la réception on calcule $t' = MAC(sk, m)$ et on vérifie que $t = t'$

Les MAC (2/2)

Et les CRC ?

Pas de secret pas d'authentification !
Un attaquant peut modifier m et recalculer $t = CRC(m)$
Les CRC ne sont pas construits pour résister à un adversaire malicieux

Sécurité : résistance aux MACs forgés

- attaquant demande les MAC pour $m_1, \ldots, m_n$ de son choix
- on lui envoie les MAC demandés $t_1, \ldots, t_n$
- l'attaquant crée un $(m, t) \neq (m_i, t_i)$

Le MAC est considéré sûr si pour tout algorithme polynomial exécuté par l'attaquant la probabilité de trouver un couple (m, t) valide est négligeable

Bits de clé, bits de MAC, et sécurité (1/2)

Question

Soit un MAC prenant en entrée une clé de 128 bits et donnant en sortie 30 bits. Un attaquant peut :

- créer un t valide pour un m en 2^{30} opérations
- créer un t valide pour un m en temps constant avec une probabilité de succès d'au plus $1/2^{30}$
- créer un t valide pour un m en 2^{128} opérations
- créer un t valide pour un m en temps constant avec une probabilité de succès d'au plus $1/2^{128}$

Bits de clé, bits de MAC, et sécurité (1/2)

Question

Soit un MAC prenant en entrée une clé de 128 bits et donnant en sortie 30 bits. Un attaquant peut :

- créer un t valide pour un m en 2^{30} opérations
- créer un t valide pour un m en temps constant avec une probabilité de succès d'au plus $1/2^{30}$
- créer un t valide pour un m en 2^{128} opérations
- créer un t valide pour un m en temps constant avec une probabilité de succès d'au plus $1/2^{128}$

Bits de clé, bits de MAC, et sécurité (2/2)

Deux paramètres de sécurité ?

Avec un MAC de taille k_M et une clé de taille k, un attaquant pourra :

- tout casser (trouver la clé) en temps 2^k
- créer un faux message avec une probabilité $1/2^{k_M}$ (sans pouvoir vérifier s'il a réussi ou pas avant de l'envoyer)

On peut montrer que réduire k_M pour un MAC sûr en tronquant ne réduit pas la sécurité (autrement qu'en augmentant la probabilité d'un faux)

Attaques

Attention dès qu'on émet $2^{k_M/2}$ messages avec un MAC il commence à avoir des collisions qui peuvent être utilisées pour forger des paquets

Il est recommandé d'avoir k_M suffisamment grand pour qu'on n'atteigne pas cette limite avant de changer de clé. Typiquement $k_M \in \{64, 96, 128\}$

Construction Hash-based MAC (HMAC)

Principe

Cas particulier de MAC basé sur une fonction de hachage (souvent SHA1)

Pourquoi c'est sûr ?

$HMAC(sk, m) = H(sk \oplus C_1 \| H(sk \oplus C_2 \| m))$ Pourquoi c'est sûr ? c.f. [1]

$HMAC(sk, m) = CSHF(sk \| m)$ est-ce sûr ?

- Oui, mais la construction au dessus l'est plus
- Non

Usage

SSL/TLS (e.g. https), IPSec, SSH, etc.

Construction Hash-based MAC (HMAC)

Principe

Cas particulier de MAC basé sur une fonction de hachage (souvent SHA1)

Pourquoi c'est sûr ?

$HMAC(sk, m) = H(sk \oplus C_1 \| H(sk \oplus C_2 \| m))$ Pourquoi c'est sûr ? c.f. [1]

$HMAC(sk, m) = CSHF(sk \| m)$ est-ce sûr ?

- Oui, mais la construction au dessus l'est plus
- Non

DM : à partir de $CSHF(sk \| m)$ on peut trouver $CSHF(sk \| m \| Padding \| X)$

Usage

SSL/TLS (e.g. https), IPSec, SSH, etc.

Les MAC basés sur une chiffrement par blocs en CBC

Taille constante : CBC-MAC

Couper m en blocs et faire un CBC avec un IV nul : sûr si taille constante
Si taille variable, le même problème que pour $CSHF(k||m)$ apparaît

Taille variable

On modifie le comportement pour le dernier bloc :

- ECBC-MAC : rajoute un chiffrement supplémentaire avec une deuxième clé secrète (on dit souvent CBC-MAC alors qu'on utilise un ECBC-MAC)
- CMAC (NIST) : utilise l'unique clé secrète pour modifier le dernier bloc avant chiffrement

Dans les deux cas les attaques par extension sont bloquées

Usages et standards

ANSI X9.9 et X9.19 (Secteur bancaire) ; FIPS 186-4 (Digital Signature Algorithm) ; WiFi (WPA2)

Attention

Ne jamais utiliser la même clé pour le chiffrement et un CBC-MAC (effondrement de la sécurité)

De manière générale : différents usages $\Rightarrow$ différentes clés

Avant de finir ... modèle de l'Oracle Aléatoire

Oracle aléatoire

Fonction déterministe H que l'on programme au fur et à mesure
On commence avec une liste de valeurs retenues vide $V = \{\}$
Quand quelqu'un demande la valeur de $H(x)$:

- on vérifie s'il y a un couple (x, y) dans V
- si c'est le cas on retourne y
- sinon on tire y aléa unif. de k bits, on ajoute (x, y) à V et on retourne y

Modèle

Une fonction de hachage se comporte comme un oracle aléatoire
Permet de faire de nombreuses preuves
Théorique : fonction de hachage déterministe $\Rightarrow$ Pas de source d'aléa

Fin !