Introdução à Criptografia:
Funções Criptográficas de Hash

Roteiros e tópicos para estudo por

Vinícius da Silveira Serafim
professor@serafim.eti.br
http://professor.serafim.eti.br

Palavras-chave: hash, integridade, armazenamento de senha

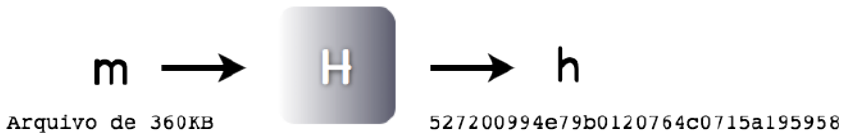
Funções Criptográficas de Hash

As funções criptográficas de hash, ou simplesmente funções de hash, possuem diversas características interessantes e que, num primeiro momento, podem fazer você se perguntar se elas possuem alguma utilidade.

A primeira delas é que essas funções são *unidirecionais*, ou seja, existe somente uma operação possível (cálculo do hash) e que não é reversível. Veja a figura abaixo:

Cuidado para não confundir!

O termo função de hash é também utilizado para mapear elementos em uma estrutura de dados chamada hash table. São coisas completamente distintas!!! Fique sempre atento ao contexto em que o termo função de hash é utilizado.



A mensagem *m* é dada como entrada para a função de hash que retorna o valor do hash, ou simplesmente o hash, da mensagem.

$$h = H(m)$$

Uma vez calculado o hash é impossível, a partir dele, obter-se novamente a mensagem *m*.

Outra característica é o fato de que, não importando o tamanho ou formato da mensagem *m*, *o hash (valor calculado) terá sempre um tamanho fixo em bits* (tipicamente 128bits, 160bits ou 256bits). Em outras palavras, se utilizarmos uma função de hash de 128bits, uma mensagem *m* de 360KB, como a do exemplo, ou uma mensagem de 16GB, terão valores de hash com o mesmo tamanho: 128bits.

A terceira característica é a *resistência à colisões*. Se, ao calcularmos o hash de duas mensagens diferentes com uma determinada função de hash, os valores de hash

resultantes forem iguais, temos o que é chamado de colisão. Uma boa função criptográfica de hash deve ser resistente a essas colisões. Por exemplo, consideremos duas mensagens diferentes m_a e m_b, então:

- O hash da mensagem m_a seria: $h_a = H(m_a)$
- O hash da mensagem m_b seria: $h_b = H(m_b)$

Se h_a e h_b forem iguais temos uma colisão.

Ok, entendemos o que é uma colisão. Mas serão elas possíveis? Com certeza, basta analisarmos quantas entradas e saídas possíveis existem para uma função de hash. Como a entrada é uma mensagem com qualquer formato e qualquer tamanho, podemos considerar que o número de entradas possíveis é infinito (∞). Vejamos agora a saída, se utilizarmos uma função de hash de 128bits, o número de saídas possíveis é 2^128. Já para uma função de hash de 160bits, o número de saídas possíveis seriam 2^160.

2^128 ou 2^160 são números muito grandes, porém o infinito é... infinito. Sendo assim, podemos afirmar que para qualquer função de hash haverão colisões. Portanto, resistência à colisões não significa que não deve existir colisões, mas sim que: “embora colisões existam, elas não podem ser encontradas.” (FERGUSON, 2003)

E, é claro, quanto maior o valor do hash de uma função de hash, mais difícil é encontrar colisões. Assim, é muito mais difícil encontrar uma colisão com uma função de hash de 256bits do que com uma função de hash de 128bits.

Em resumo

- Funções de hash geram um valor *único e exclusivo* para uma determinada massa de dados. Altere um único bit dessa massa e o valor do hash será completamente diferente;
- O hash é irreversível. Não há informação suficiente no valor do hash *h* para recuperar a mensagem *m*; e
- O valor de hash tem sempre um tamanho fixo e pequeno de bits (512bits é um dos maiores).

Funções de hash atualmente em uso

As duas funções de hash mais comuns são a MD5 (de 128bits) e o SHA-1 (de 160bits). Porém, do SHA-1, temos variações com valores de hash maiores como o SHA-256 (de 256bits) e o SHA-512 (de 512bits).

Embora todos os acima citados sejam seguros, Ferguson e Schneier recomendam o uso do SHA-256 ou SHA-512. (FERGUSON, 2003)

Eu coloco da seguinte maneira, não use MD5 e o SHA-1 para aplicações realmente críticas como, por exemplo, controlador de um reator nuclear. Nesse caso acredito que nem o SHA-256 eu utilizaria. ;-)

Para que serve então uma função de hash?

Alguns dos principais usos são:

- Verificação de integridade de arquivos
- Armazenamento de senhas
- Resolver o problema da velocidade na assinatura digital

Verificação de integridade de arquivos

Consideremos o exemplo em que Alice deseja armazenar um arquivo num servidor e, mais tarde, ao necessitar do arquivo quer ter certeza de que o mesmo continue íntegro.

Alice pode então calcular o hash *h* do seu arquivo *m* e guardar o hash em algum local seguro:

Alice: h = H(m)

Alice: [h]

Alice então envia apenas o arquivo *m* para ser armazenado no servidor:

Alice: m > Servidor

Mais tarde, quando precisar do arquivo, Alice recebe o arquivo do Servidor e calcula novamente o hash:

Alice: m < Servidor

Alice: h' = H(m)

Agora Alice tem dois valores de hash, o valor *h* que ela havia calculado antes de enviar o arquivo para o servidor e o valor *h'* que ela calculou depois de recuperar o arquivo do servidor. Agora basta que Alice compare esses dois valores:

Alice: i = (h == h')

Se *h* for igual à *h'* então a integridade foi mantida, ou seja, o arquivo não sofreu alteração alguma. É exatamente o mesmo arquivo que Alice armazenou anteriormente no servidor.

Porém, se *h* for diferente de *h'* isso irá indicar que a integridade foi comprometida, ou seja, alguém (ou alguma coisa) alterou o arquivo armazenado no servidor. Alice não pode mais confiar no conteúdo do arquivo.

Note que, para essa verificação funcionar, Alice tem que guardar o primeiro hash calculado em algum lugar seguro (ex.: no seu próprio computador). Se ela armazenasse o hash *h* junto com o arquivo *m* no servidor, nada impediria que um atacante alterasse o arquivo e então calculasse novamente o hash e substituísse o valor calculado pela Alice.

Mais tarde Alice, ao recuperar o arquivo e verificar a integridade, não perceberia a alteração.

Uma solução para isso é usar um HMAC (Hash Message Authentication Code). Um HMAC é basicamente uma função de hash com uma chave:

$$h = H(K,m)$$

A ideia é que somente quem conhece a chave K possa gerar hashes válidos. Seguindo com o mesmo exemplo anterior, ao calcular o hash do arquivo Alice adicionaria uma chave somente de seu conhecimento:

$$\text{Alice: } h = H(K,m)$$

Alice então armazena tanto o hash *h* quanto o arquivo *m* no servidor:

$$\text{Alice: } m,h > \text{Servidor}$$

Mais tarde, ao recuperar o arquivo do servidor, Alice calcula novamente o hash acrescentando a chave que somente ela conhece:

$$\text{Alice: } m,h < \text{Servidor}$$

$$\text{Alice: } h' = H(K,m)$$

E então compara o valor do hash armazenado no servidor com o novo valor de hash calculado:

$$\text{Alice: } i = (h == h')$$

Mais uma vez, se os hashes forem iguais o arquivo não foi alterado. Caso contrário, os hashes serão diferentes e Alice terá certeza de que houve alguma alteração não autorizada no arquivo e/ou no valor do hash armazenado.

O que muda nesse caso é que, um atacante pode alterar o arquivo, mas não consegue gerar um hash que seja igual a um hash calculado pela Alice pois ele não conhece a chave por ela utilizada.

Armazenamento de senhas

A forma correta de armazenar senhas é com a utilização de funções de hash! Veja como não fazer isso no roteiro sobre algoritmos de chaves simétricas.

Eis a solução correta: o primeiro passo é o cadastramento da senha feito pelo usuário. O usuário informa a sua senha ao sistema:

$$\text{Usuário: senha} > \text{Sistema}$$

O sistema então calcula o hash da senha e o armazena em seu banco de dados:

$$\text{Sistema: } h = H(\text{senha})$$

Sistema: [h]

Mais tarde, para autenticar o usuário: o usuário informa a senha ao sistema e o sistema calcula novamente o hash da mesma.

Usuário: senha > Sistema

Sistema: h' = H(senha)

Então o sistema compara o hash recém calculado com o hash armazenado para aquele usuário no banco de dados.

Sistema: i = (h == h')

Se os hashes forem iguais significa que o usuário forneceu a mesma senha cadastrada anteriormente. Se forem diferentes, o usuário esqueceu sua senha ou um atacante está tentando se passar por ele.

Essa solução, embora ainda não completa, já fornece uma boa segurança: a senha não é armazenada no banco de dados, mas somente o hash e o hash é irreversível. Assim, nem um atacante externo (cracker) nem um interno (administrador ou DBA mal intencionado) conseguem saber qual é a senha utilizada pelo usuário. Isso é muito importante, pois frequentemente usuários utilizam a mesma senha em diversos sistemas diferentes.

Porém ainda temos um detalhe a ser resolvido. Dois usuários que utilizem exatamente a mesma senha, terão, no banco de dados, valores de hash idênticos. É claro que o hash continua sendo irreversível, porém agora o atacante pode vir a saber quais usuários utilizam a mesma senha.

Imagine que em um determinado sistema, Alice e Bob usam a mesma senha 12345, teríamos então na tabela de usuários no banco de dados:

```
alice: 8cb2237d0679ca88db6464eac60da96345513964
  bob: 8cb2237d0679ca88db6464eac60da96345513964
```

A solução é bem simples: salgar o hash. O sal é alguma informação randômica e não sigilosa que é juntada à senha antes de calcular o hash. Assim, vamos rever nosso processo.

Cadastramento da senha:

Usuário: senha > Sistema

Sistema: h = H(senha + sal)

Agora o sistema gerou um sal (ex: três caracteres randômicos) e juntou à senha antes de calcular o valor do hash. Mais tarde o sistema necessitará não só do hash mas também do sal para autenticar o usuário. Assim o sistema armazena no banco o hash e o sal do usuário:

Sistema: [h] [sal]

Mais tarde, para autenticar o usuário:

Usuário: senha > Sistema

Sistema: h' = H(senha + sal)

E, mais uma vez, o sistema compara o hash armazenado com o hash recém calculado para decidir se o usuário é quem ele realmente diz ser. E veja como ficam armazenados os hashes das senhas dos usuários Alice e Bob neste sistema:

```
alice: 598c39303ddd085d26c4b3a5f478e6f20ec10c25,wrt
  bob: 4fc65e108e3573fd690116637953009cd8ca64b3,p8n
```

Ambos continuam usando a mesma senha, 12345. Porém a senha de cada um foi salgada com três caracteres randômicos distintos (destacados em negrito acima). Esses três caracteres adicionais alteram completamente o valor do hash e, mesmo não sendo sigilosos, não comprometem em absoluto a irreversibilidade da função de hash utilizada.

Portanto, em qualquer sistema que armazene senhas de usuários, armazene apenas o sal e o hash salgado. ;)

Resolver o problema da velocidade na assinatura digital

Para compreender a solução deste problema, tenha em mente os três itens da seção "Em resumo" deste roteiro e volte ao roteiro sobre algoritmos de chaves assimétricas.

Bibliografia

FERGUSON, N. F.; SCHNEIER, B. Practical Criptography. 1a edição. Wiley, 17 de abril de 2003. 432 pág.

PFLEEGER, C. P. Security in Computing. 4a edição. Prentice Hall, 23 de outubro de 2006. 880 pág.