Uma visão geral sobre Funções HASH : Teoria e segurança

31 Aug 2018

English version

A definição mais comum para funções de *hash* seria “uma função que mapeia uma string de bits de comprimento arbitrário para uma string de bits de comprimento fixo”.

Funções de *hash* são unidirecionais, ou seja, uma função sobrejetora que não permite inversa, logo, é fácil calcular o valor de *hash* a partir de uma entrada qualquer de comprimento arbitrário.

Em contrapartida deve ser difícil retornar ao valor anterior da mensagem a partir de um dado valor de *hash*.

Propriedades Hash

Para uma aplicação criptográfica, há três propriedades desejáveis:

Resistência à colisão

É computacionalmente inviável encontrar qualquer par *(x, y)* tal que *H(x)=H(y)*

Resistência à primeira inversão

Para qualquer valor *h* dado, é computacionalmente inviável encontrar *x* tal que *H(x)=h*.

Resistência à segunda inversão

Para qualquer bloco de dados x, é computacionalmente inviável encontrar *y* diferente de *x*, tal que *H(y)=H(x)*.

A situação ideal é que todo valor de *hash* gerado seja realmente único para cada valor de entrada, mas isso não é possível. Nas funções de *hash* os valores de *hash* possíveis de serem gerados são finitos, pois têm um tamanho fixo de saída e como o tamanho da entrada é teoricamente infinita, pelo Princípio da Casa dos Pombos, haverá colisões com duas entradas distintas, resultando no mesmo *hash*.

Porém essas funções foram construídas para minimizar colisões válidas, ou seja, é muito improvável que duas mensagens que possuem sentido no contexto resultem na mesma *hash*.

Exemplos de uso de hash

As funções de *hash* têm uma grande variedade de usos em segurança, tais como:

Autenticação de mensagens

Mecanismo ou serviço usado para verificar a integridade de uma mensagem; permite assegurar que a informação recebida seja igual à mensagem enviada (sem modificação, inserção, supressão ou *replay*);

Assinaturas digitais

O valor de resumo de uma mensagem é cifrado com a chave privada do emissor, qualquer usuário que possua a chave pública pode verificar a integridade da mensagem que é associada com a assinatura digital;

Arquivo de senhas

Um resumo de uma senha fica armazenado em um arquivo do sistema operacional em vez de se armazenar a senha propriamente dita, caso o arquivo de senha seja violado, o atacante só conseguirá obter o *hash* da senha;

Detecção de intrusos e vírus

Para cada arquivo F do sistema, armazena-se também o *hash H(F)*, se houver qualquer alteração em F, esta será percebida;

Construção de funções pseudoaleatórias (PRF) ou construção de gerador de números pseudoaleatórios (PRNG)

Para geração de chaves simétricas.

Blockchain

Os dados na *blockchain* são "*hashes*" em cada bloco. Se o bloco for alterado, ou seja, alguém tentou mudar quantos *bitcoins* possuía ou quanto deveria enviar, o valor de *hash* seria diferente e todos poderiam detectar que alguma coisa mudou.

O valor *hash* do bloco anterior é usado para calcular o valor *hash* do bloco atual, criando um link entre os blocos.

São exemplos de algoritmos, com link das suas implementações de *hash*:

- SHA-1; Inseguro
- SHA-256;
- MD5; Inseguro

Guardar suas senhas no banco usando *hash* é uma boa prática (no momento atual já podemos dizer que é obrigatória) para caso de quando seu banco de dados for exposto por algum *cracker*, ele não consiga ter acesso direto à senha dos seus clientes.

Então é só colocar criar um *hash* para minha senha e problema resolvido? Não totalmente.

Como as hash são quebradas

Pelas propriedades de *hash* vimos que dado um *hash* *H(x)* não deveria ser computacionalmente viável descobrir a entrada *x*, logo não é possível "descriptografar" a *hash*, afinal ela é uma operação que não permite retorno.

Mas sabemos que dada uma entrada sempre é gerado o mesmo *hash*, e os *crackers* também sabem disso. Então há alguns ataques conhecidos para *hash* de senhas.

Ataques de dicionários e força bruta

O jeito mais simples de tentar achar um *hash*.

Um ataque de dicionário usa um arquivo contendo palavras, frases, senhas mais comuns e coisas assim para calcular a *hash* de cada uma e verificar se bate com alguma da lista ou do banco de dados.

Já o ataque de força bruta testa todas as possibilidades de palavras com um numero definido de caracteres e verifica se o calculo do *hash* é igual a algum da lista.

Tabela de consulta

Tabela de consulta é um método extremamente efetivo para quebrar várias *hashes* rapidamente. A ideia geral é pré-computar as *hashes* de senhas de um dicionário e guardar

esse valor com sua senha correspondente. Uma boa implementação de uma tabela de consulta pode processar centenas de *hashes* por segundo, e conter bilhões de *hashes*.

Tabela de consulta reversa

Do inglês, *Reverse Lookup Tables* (não sei se é a melhor tradução), esse ataque permite um atacante aplicar um dicionário ou um ataque de força bruta para várias *hashes* ao mesmo tempo.

Primeiramente, o atacante cria uma tabela de consulta que mapeia cada *hash* de senha dos contratos da base de dados para uma lista de usuários que tem aquela *hash*.

```
Searching for hash(apple) in users' hash list...     : Matches [alice3, 0bob0, charles8]
Searching for hash(blueberry) in users' hash list... : Matches [usr10101, timmy, john91]
Searching for hash(letmein) in users' hash list...   : Matches [wilson10, dragonslayerX, joe1984]
Searching for hash(s3cr3t) in users' hash list...    : Matches [bruce19, knuth1337, john87]
Searching for hash(z@29hjja) in users' hash list...  : No users used this password
```

O atacante então faz um ataque de dicionário ou força bruta, e ao descobrir uma senha ele já possui uma lista de usuários que a possuem. Esse ataque é bastante efetivo por que é comum muitos usuários terem a mesma senha.

Rainbow tables

Uma **rainbow table** é uma tabela de consulta de *hashes* pré calculados. É um exemplo prático do trade-off Memória e Tempo.

Ele se parece com a tabela de consulta, exceto que sacrifica a velocidade de quebrar a *hash* para fazer as tabelas menores. Por serem menores, as soluções para mais *hashes* podem ser armazenadas com o mesmo espaço, sendo mais efetivo em termos de armazenamento.

Então, exemplificando
Temos um cliente que possui a senha 123456 e que é armazenada usando o algoritmo SHA-1, então no banco terá a *hash* 7c4a8d09ca3762af61e59520943dc26494f8941b

Quando o *cracker* tiver posse desse *hash* basta usar um site ou algum programa especifico, que testa se possui essa *hash* no banco de dados e retorna o valor que gerou ela. Como por exemplo o http://md5decrypt.net/en/Sha1/#answer

A lista das 100 senhas mais usadas, provavelmente já está em todos os bancos possíveis de *hash*, por isso existem campanhas para usar senhas cada vez mais difíceis para dificultar esses bancos de *hashes*. Mas mesmo com esses problemas, os usuários não querem ter o problema de usar senhas mais complexas, então como desenvolvedor, sabendo desses ataques, qual é a melhor solução para esse problema? Colocar mais "sal" no seu *hash*, ou seja, usar *salt*.

salt

Salt e implementações de segurança

O conceito de *salt*, é colocar alguma informação adicional na senha para adicionar complexidade ao calcular o *hash*, por exemplo:

```
hash("hello")                  = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

Ao colocar informação adicional na senha fica mais difícil de criar uma tabela de consulta.

O que evitar ao utilizar salt

Reusar o *salt*

Mesmo sendo um *salt hardcoded* no programa, ou gerado uma vez randomicamente. É inefetivo por que se dois usuários tem a mesma senha, eles terão o mesmo *hash*. Um ataque de tabela de consulta reversa, dessa forma um ataque de força bruta poderia ser aplicado, simplesmente testando as possibilidades de *salt* e uma vez encontrado o *salt*, todas as suas senhas ficam vulneráveis.

Usar *salt* muito curto

Se seu *salt* é muito curto, um atacante pode construir uma tabela de consulta para todos os possíveis *salts*. Por exemplo, se você usa três caracteres ASCII, então tem somente 857,375 *salts* possíveis. E isso para termos computacionais não é muito, se para cada tabela de

consulta contem somente 1MB das senhas mais comuns, gerar elas seria uns 837GB, e já tem HD de 1TB custando uns 200 reais hoje, e é um investimento muito barato comparado ao estrago que faria ao sistema.

Usar username como salt

Embora devessem ser únicos no sistema, eles geralmente são usados por outras contas em outros serviços. Um atacante poderia fazer a relação user-senha e criar tabelas de consulta e quebrar *hash* que tem *username* como *salt*.

Para ser impossível computacionalmente criar uma tabela de consulta para cada *salt* possível, o *salt* deveria ser longo. Uma boa regra é usar um *salt* que tem o mesmo tamanho da saída da função *hash*, a saída do SHA-256 é 256 bits (32 bytes), então o *salt* deveria ter ao menos 32 números randômicos.

Perceba que o *salt* não precisa ser segredo, ele só deve ajudar a evitar os ataques que listei. Eu particularmente gosto de usar o *hash* do *timestamp* da hora que a senha foi alterada, desse jeito deixo todas as propriedades satisfeitas e ainda deixo a data no formato que defino tornando ainda mais difícil a tarefa de gerar uma tabela de consulta.