Um Estudo comparativo de funções hash para senhas criptografadas

Alexandre Felipe Müller de Souza

Curso de Pós Graduação em Redes e Segurança de Sistemas
Pontifícia Universidade Católica do Paraná

Curitiba, Fevereiro de 2015

Resumo

O objetivo deste trabalho é desenvolver um estudo teórico e comparativo sobre os principais processos de se armazenar senhas atualmente. Entre estes processos estão políticas de senhas e algoritmos de *hash* utilizados para criptografia. Serão apresentados aspectos de implementação de cada um, assim como o estado da arte na segurança de cada um. Além disso o trabalho demonstra a facilidade de ataque em diversas situações reais.

1 Introdução

1.1 Breve explicação da temática

O avanços nos diversos tipos de autenticações por *tokens* com certificados criptográficos e reconhecimento biométrico ainda não substituíram o método de autenticação por senha. A autenticação e obtenção de privilégios de sistemas através de uma palavra passe ainda tem sido utilizada em 95% de sistemas cliente servidor[1] pelo seu baixo custo e facilidade de implementação. Por outro lado vários dos problemas com a autenticação por senhas se devem não só a usuários, mas aos desenvolvedores e administradores. Os usuários tem papel fundamental de não compartilhar suas senhas com outros e trocá-las constantemente. Os analistas de TI, por outro lado, tem a tarefa de desenvolver um sistema que só permita o acesso àquele que realmente possui a senha e ninguém mais.

Apesar de parecer simples para os usuários as implementações destes sistemas não são triviais. A forma mais segura e usual de se processar uma autenticação por senha é gerando uma referencia da senha chamada de *Hash*. Dessa forma nem mesmo os administradores possuem a senha armazenada na sua forma textual. Entretanto eles podem comparar o resultado da função *hash* usando a senha fornecida, com o resultado da função *hash* da senha gerada no passado. Sendo assim não é necessário armazenamento da senha em forma textual. A segurança desse processo depende exclusivamente da função *hash* utilizada[2]. Desta forma todos sistemas que fornecem recurso de lembrete de senha são potencialmente inseguros.

1.2 Motivação do tema

Estudar a segurança do armazenamento de senhas é de fundamental importância para segurança da maior parte dos sistemas em redes nos dias de hoje. Funcionários das organizações com acesso de leitura ao banco de dados poderiam ler e utilizar as senhas. Além disso invasores tem obtido fácil acesso a banco de dados de sistemas em produção através de vários tipos de ataques. Por isso é preciso criptografar os dados de senhas de uma forma segura.

1D Roig - US Patent 7,509,495, 2009; Google Patents

2Mihir Bellare , Ran Canetti , Hugo Krawczyk - Keying hash functions for message authentication (1996)

Este trabalho propõem estudar qual a forma mais segura de se armazenar senhas, desde a política de requisitos da senha, até o algoritmo utilizado com base de dados de sistemas em produção. Para isso será realizado um ataque a cada um dos algoritmos em um parque computacional modesto e comparado a chance de acerto. Os fontes do protótipo de testes será disponibilizado para uso de administradores e trabalhos futuros. Além disso o ataque será baseado nas estatísticas dos padrões das senhas e não no caso qualquer de entrada como nos artigos similares. Este trabalho pretende mostrar que mesmo em determinados métodos que não possuem um ataque considerado eficiente, ao não se definir os critérios corretos para definição da senha, a segurança continua comprometida.

2 - Revisão bibliográfica

2.1 Trabalhos similares

O comparativo de funções hash é bem explorado pela dissertação de mestrado de Joel Lathrop[3]. Neste trabalho o autor traça um histórico da evolução da segurança das funções hash começando pelo MD5 que foi amplamente utilizado, até 2004 quando se publica um método (ataque diferencial) capaz de achar dois blocos de texto que geram uma colisão em poucas horas. Esse trabalho foi liderado pelo pesquisador chinês Xiaoyun Wang[4]. Com aperfeiçoamentos posteriores a esse tipo de ataque uma colisão do MD5 poderia ser encontrada em qualquer máquina modesta.

O desdobramento dessa insegurança levou a edição do concurso da National Institute of Standards and Technology (NIST) para escolha da função hash mais segura, sendo vencedora a Keccak que recebeu o nome de SHA-3. Foram analisados 51 algoritmos incluindo uma versão aprimorada do MD5, o MD6, que foi descartado por conter pequenas vulnerabilidades. Apesar do trabalho de Wang, ser aplicável para funções hash de forma geral, até agora os melhores resultados ficaram na quebra do MD5.

O trabalho de Lathrop testou candidatos do concurso do NIST utilizando um método chamado Cube Attack. Este método é genérico e pode ser usado pra qualquer algoritmo de criptografia sem levar em consideração seus detalhes de implementação. O trabalho testa em condições bem reduzidas de complexidade dois algorimos: Essence e Keccak (vencedor SHA3). A conclusão aponta que o Keccak é mais seguro, entretanto já era esperado pelo número de rounds (que quanto mais rounds possui a função hash, mais difícil é o cube attack).

O presente trabalho se distingue do que já foi estudado no tema, por ser um caso de uso específico de funções hash. O armazenamento de senhas se distingue de uma simples assinatura de autenticidade e integridade de um texto longo por ser computacionalmente fácil (com conjunto bem mais reduzido de tentativas). Mais sobre estas diferenças poderão ser vistas ao longo do trabalho.

2.2 Algoritmos mais usados atualmente

Apesar de esse ser um tema amplamente discutido é difícil de achar dados sobre o qual é o algoritmo mais utilizado em sistemas de produção na atualidade. Segundo Lathrop[5], MD5 seria a função hash mais usada de toda existência. Cita-se também na literatura técnica informal e formal os algoritmos: MD5, SHA1 ou SHA3. Por outro lado não existe pesquisa que aponte qual é a real utilização de cada um. Este ponto pode ser importante de estudos no futuro. Com base na publicação da agência americana NIST, sugerindo o uso das famílias SHA para resumo de mensagens[6], podemos considerar que esse seja importante de se estudar. Outra família importante

3Cube Attacks on Cryptographic Hash Functions
4Xiaoyun Wang and Hongbo Yu, How to Break MD5 and Other Hash Functions
5Cube Attacks on Cryptographic Hash Functions
6FIPS PUB 180-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Secure Hash Standard (SHS)

de se estudar é dos algoritmos MD4, MD5 e MD6 pelas RFCs já publicadas sobre o uso de HMAC que os dão valor histórico[7] [8] [9].

2.2 Funções Hash

Funções hash são funções algorítmicas que geram uma saída de tamanho limitado dado uma entrada de tamanho arbitrário. Dessa forma estas funções são análogas as funções sobrejetoras do estudo de conjuntos. Quanto a segurança podemos citar os seguintes aspectos:

- *Pré Imagem - Dada uma função hash h e um hash y, achar uma mensagem de valor x tal que h(x) = y*
- *Segunda Pré Imagem - Dada uma função hash h e uma mensagem de valor x, achar outra mensagem de valor x' tal que x' ≠ x e h(x') = h(x)*
- *Colisão - Dada uma função hash h, achar duas mensagens de valor x e x' tal que x ≠ x', mas h(x) = h(x')*
- *Extensão de comprimento - Dada uma função hash h, um hash de valor h(x), e o comprimento da mensagem |x|, achar um x' tal que h(x||x') pode ser calculado, onde || representa concatenação.*

Retirado de Lathrop [10] tradução própria

A dificuldade de se atender a esses aspectos de segurança ficam por conta da análise combinatória e probabilidade. Se por um lado um hash resultante de 128 bits tem 2^{128} possíveis combinações, bastam 2^{64} testes em média para se encontrar uma colisão. Essa conclusão é resultado do chamado *paradoxo do aniversário*, que se chama paradoxo por mero preciosismo já que não contraria a lógica, entretanto é contra intuitivo.

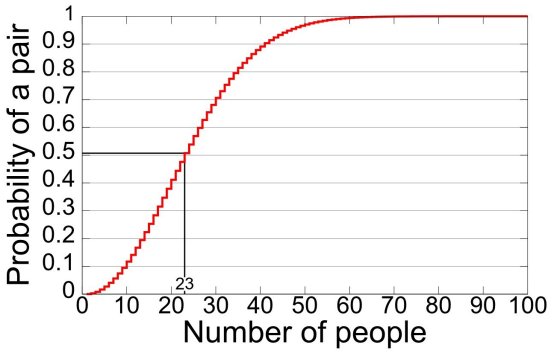


Figura 1: Probabilidade de mesmo aniversário em função do número de pessoas[9].

2.2.1. Merkle-Damgård

Resultado de um artigo de 1989 de dois pesquisadores que levam cada um estes nomes[11], o método de construção Merkle-Damgård é a base dos algoritmos da família MD e SHA1. É considerado um método capaz de criar hash resistente a colisão usando várias funções de compressão em cadeia. A ideia por de trás desse método é que se a função de compressão é resistente a colisão, então o algoritmo todo é resistente.

Nesse contexto de entradas de tamanho limitado, o que dá a característica difícil de se reverter (obter a entrada através da saída) é justamente cada passo utilizar a saída do antecedente

7R. Rivest, RFC1320; MIT Laboratory for Computer Science and RSA Data Security 1992
8R. Rivest, RFC1321; MIT Laboratory for Computer Science and RSA Data Security 1992
9RIVEST, Ronald L. - The MD6 hash function A proposal to NIST for SHA-3
10Lathrop , Joel - Analysis of NIST Cryptographic Hash Function, página 4
11Ivan Bjerre Damgard, A Design Principle of Hash Functions

como entrada. Dessa forma, com todos os passos em série, descobrir qual é o valor de entrada com base na saída é como descobrir os operandos através do resultado. A dificuldade então ser torna a mesma de resolver um sistema de equação linear possível, mas indeterminado.

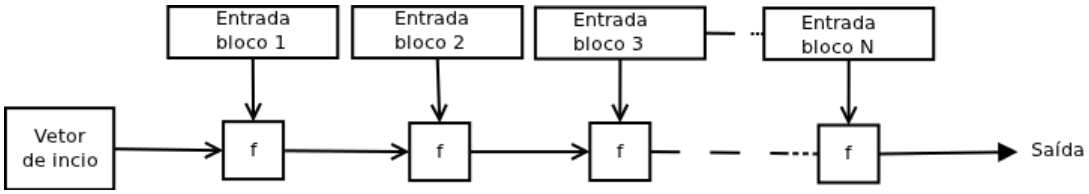


Figura 2: Exemplo simplificado da construção Merkle-Damgård

Neste gráfico a função de compressão é chamada de f, e todo processo começa com um vetor inicial. Geralmente ao final do processo existe mais uma função de finalização, com objetivo de embaralhar um pouco mais a saída.

2.2.2. Ataque Diferencial

O método de criptoanálise diferencial é utilizado em vários algoritmos de criptografia de forma geral. Este método consiste em testar como as diferenças na entrada geram resultados na saída. Através desse entendimento é possível saber qual entrada poderia gerar uma saída específica.

O ataque específico do MD5, utiliza o "ou exclusivo" como função de diferença, mas também utiliza subtração modular como medida de diferença de forma combinada. Estes ataques bem sucedidos utilizando ambos necessitam de uma porção de parâmetros que aumentam ou diminuem tanto a chance de encontrar a segunda imagem quanto o tempo de processamento. Entretanto este método baseia-se em modificação das mensagens para obtenção de uma segunda imagem, o que enfraquece o algoritmo, mas não permite a simples obtenção de uma pré imagem.

2.2.3. Rainbow Table

A ideia do ataque de rainbow table é acelerar o tempo de processamento utilizando se da memória para armazenar estados já processados anteriormente. Enquanto os ataques de força bruta utilizam somente o processamento e as tabelas de hash já processadas utilizam o máximo de memória, a ideia por de trás desse método é conciliar as duas coisas. Este método é mais indicado para senhas maiores e mais complexas, enquanto a força bruta é indicada para os outros casos.

Esse método é um pouco mais intrusivo a implementação de cada função hash, pois ele explora as funções de redução especificamente.

2.2.4. MD5

O algoritmo de MD5 foi publicado 1992 com objetivo de substituir o vulnerável MD4. Ambos geram uma saída de 128 bits (16 bytes) tipicamente expressa em 32 caracteres hexadecimais.

Ao receber a entrada de tamanho já limitado, vetor de 16 números inteiros da linguagem C, o MD5 funciona em 64 rounds de 4 tipos de operações: F, G, H e I. Cada round utiliza como entrada a saída do anterior (como é definido no meta processo Merkle-Damgard) com operações lógicas definidas a seguir:

- F(B,C,D) = (B.C) + (~B.D)
- G(B,C,D) = (B.D) + (C.~D)
- H(B,C,D) = B⊕C⊕D
- I(B,C,D) = C ⊕ (B+~D)

A notação usada é a usual de lógica: adição significa "or bit a bit", multiplicação significa "and bit a bit", til significa inversão dos bits e ⊕ significa "or exclusivo".

Abaixo podemos ver os primeiros 3 rounds da função MD5, utilizando-se da função de compressão F:

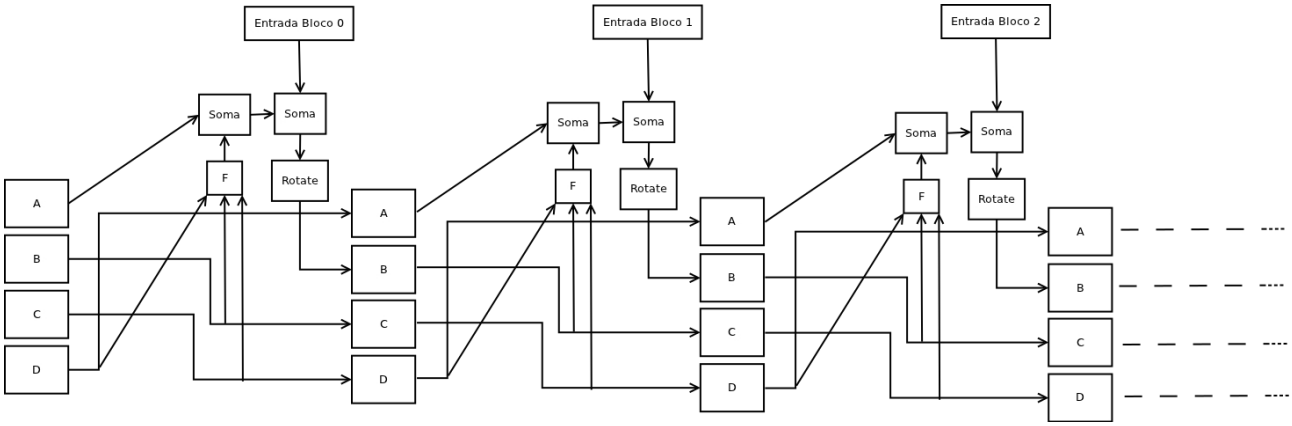


Figura 3: Exemplo dos 3 primeiros rounds do MD5

Os 4 registradores A, B, C e D são pré-determinados pelo algoritmo. A função de compressão F é usada por 16 rounds, até toda entrada ser consumida. Logo após essa fase mais 16 rounds são aplicados com a função G, seguido pela H e por último a função de compressão I. Ao todo a entrada de 16 números ou blocos é consumida 4 vezes. Ao final o vetor de inicialização é somado com os valores de A, B, C e D e tem-se a hash de resultado.

Em qualquer tipo de ataque o MD5 tem se mostrado frágil, mesmo com ataque de força bruta a saída de tamanho 128 bits leva a probabilidade média de colisões em tempo razoável (chamado de ataque do aniversário). Além da função de finalização (as 4 últimas somas), o último round poder ser parcialmente eliminado em um ataque devido a possibilidade de desfazer seus últimos passos. Como as constantes são conhecidas elas podem ser subtraídas, e as rotações desfeitas. Mais especificamente dos 64 rounds somente 61 se tornam necessários em média.

2.2.5. SHA-1

O algoritmo do SHA-1 faz parte da família de algoritmos SHA-0, SHA-1 e SHA-2. Ele gera uma saída de 20 bytes, o que combinacionalmente é melhor que o MD5. Além disso não se teve até hoje um ataque bem sucedido ao SHA-2 que é uma versão mais segura do SHA-1.

Em termos de funcionamento ele é parecido com as outras funções hash por utilizar o meta processo Merkle-Damgård. São definida 5 constantes aleatórias: A, B, C, D e E (ao invés de 4 do MD5), a entrada é preenchida num buffer completado com zeros e o tamanho da string é concatenada ao final do buffer. Então há 80 rounds, cada 20 passos com cada uma das funções a seguir:

f1(B, C, D) = (B.C) + (B.D)
f2(B, C, D) = B ⊕ C ⊕ D
f3(B, C, D) = (B.C) + (B.D) + (C.D)
f4(B, C, D) = B ⊕ C ⊕ D

A notação usada é a usual de lógica: adição significa “or bit a bit”, multiplicação significa “and bit a bit”, til significa inversão dos bits e ⊕ significa “or exclusivo”.

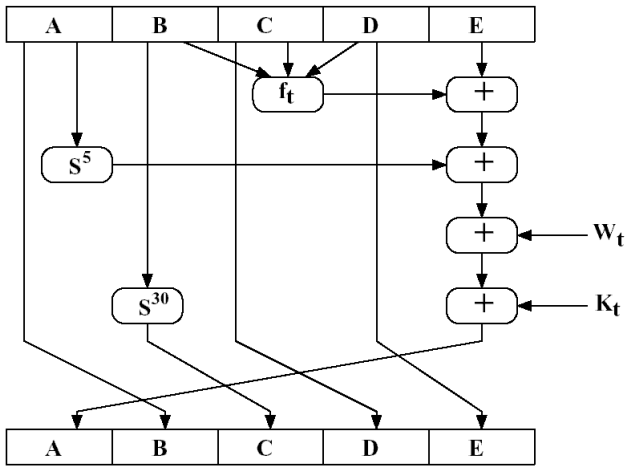


Figure 9.6 Elementary SHA Operation (single step)

Figura 4: Um passo do SHA1 extraído de [12]

O algoritmo SHA1 é muito parecido com o MD5, só aumentando o número de passos e a quantidade de registradores intermediários que levam a uma hash maior. Apesar dessa segurança melhorada, vários esforços de colisão pela força bruta (com uso de GPU) tem obtido resultados positivos ao atacar sistemas que usam SHA-1. Por estes motivos não há razão para que não se adote SHA-2 ou SHA-3 que são mais seguros.

2.2.6. SHA-2

O algoritmo de hash SHA2 é uma versão aprimorada do SHA1 foi desenvolvido e publicado pela agência americana NSA. Estas publicações foram resultados dos avanços em ataques ao SHA1. Este algoritmo pode ser configurado desde 224 a 512 bits de saída que aumenta sua segurança. Apesar de mais seguro, esse algoritmo compartilha a funcionamento básico do seu predecessor, e apesar de ainda não estar em risco os ataque recentes ao SHA1 forçaram o desenvolvimento do SHA3.

2.2.7. SHA-3

Este algoritmo de hash foi publicados pela agência americana NSA e é considerado o mais seguro atualmente. Este também pode ser configurados desde 224 a 512 bits de saída.

O SHA-3 em especial foi vencedor do concurso do NIST como mais seguro de vários candidatos. A possibilidade combinatória tanto do SHA2 quanto SHA3 (512 bits) é muito melhor aos seus concorrentes, mesmo levando o paradoxo do aniversário em consideração, pois 2^{256} é um conjunto muito grande. É seguro também fundamentalmente não se basear em Merkle-Damgård, mas invés disso ser baseado em uma ideia similar chamada função esponja. Como a maioria dos ataques anteriores exploravam funções internamente semelhantes, o uso de outro mecanismo o torna mais seguro. A construção esponja leva esse nome (a grosso modo) por "absorver" a mensagem em pedaços por uma "esponja" que então quando apertada tem-se a saída.

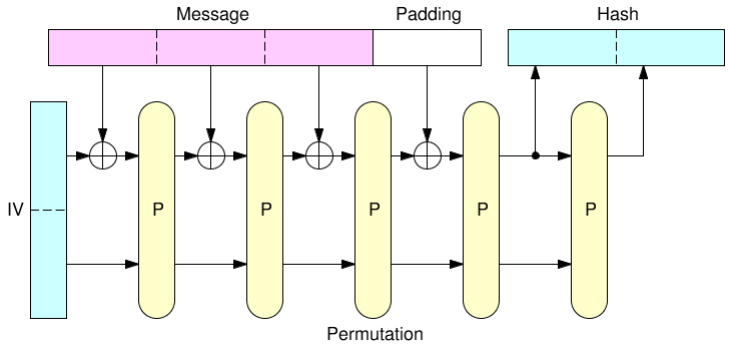


Figura 5: Como funciona função esponja[14]

2.3 Salt

Com objetivo de aumentar a segurança das senhas armazenadas o administrador pode-se optar por concatenar outro texto ao final da senha antes de usar a função hash. Essa prática comum, é chamada de Salt (Sal) e tem objetivo de aumentar a chave. Essa alternativa aumenta a possibilidade de quebra em casos em que o invasor desconheça a parte do texto que foi concatenada ao final. Além disso o ataque diferencial obtém dois blocos distintos de texto que geram uma colisão, ao se colocar uma condição a mais, que deve além de gerar uma colisão terem o mesmo texto concatenado, o ataque fica mais difícil.

Uma alternativa um pouco mais segura, seria concatenar um texto diferente para cada senha, como por exemplo a data de aniversário do usuário. Isso eliminaria completamente os ataques utilizando tabelas e ataque diferencial, mas não impede ataques por rainbow tables ou força bruta quando o atacante obtém a informação. Entretanto com incrementos (salt) suficientemente grandes e distintos para cada usuário, o ataque de rainbow tables se torna impossível[12].

3 Metodologia Proposta

3.1 Obtenção e geração de uma amostra de senhas

Para obter uma lista de senhas reais, pode-se recorrer a projetos de software de código aberto que já possuem listas de senhas. Este projetos utilizam-se de dados de sistema em produção que foram invadidos e os dados das bases invadidas disponibilizados por invasores. Isso nos dá um panorama real de como os usuários definem suas senhas e quão fácil ou difícil será obter uma entrada que colide com a *hash*. Estes projetos ajudam os administradores a aumentar a segurança dos seus sistemas. Com base na lista de milhares de senhas[13] foi selecionado uma amostra de 2 mil senhas aleatórias para tornar viável o processamento e ao mesmo tempo relevante estatisticamente. Essa lista, que pode ser obtida em anexo[14], foi considerada e analisada com as seguintes características:

- Comprimento das senhas
  - 4 = 469 (9.9%)
  - 5 = 509 (10.1%)
  - 6 = 1726 (34.52%)
  - 7 = 1065 (21.3%)
  - 8 = 928 (18,56%)
  - 9 = 177 (3,54%)
  - >10 = 125 (2.5%)

12Alexander, Steven. Password Protection for Modern Operating Systems - USENIX Association Junho 2004
13Top Network Security Tools. http://sectools.org/
14Lista de 2 mil senhas usadas nos testes http://creamcracker.googlecode.com/svn/wiki/passwordList_2k.txt

- Caracteres mais comuns
  - e = 9,5% de todos os caracteres
  - a = 9,0%
  - r = 6,3%
  - o = 5,7%
  - s = 5,7%
  - i = 5,6%

Essa análise sobre os padrões das senhas é fundamental num ataque do tipo força bruta. Usualmente o algoritmo de enumeração das senhas trata cada combinação de carácteres com igual importância. Por outro lado se pensarmos na probabilidade de ocorrência de cada um, podemos levar prioridade na escolha e no caso médio ter uma chance de acerto muito maior (encontrar a colisão mais rapidamente). Outra informação importante é quanto ao tamanho (comprimento) das senhas: 97,5% das senhas tem menos de 10 caracteres. Como a enumeração de senhas é um conjunto que cresce exponencialmente em função do tamanho essa informação é preocupante, pois senhas menores diminuem muito o campo de busca.

3.2 Geração das hashes em cada um dos algoritmos e resultados obtidos na tentativa de quebra de cada um

Como independentemente de existir ou não um método de quebra possível, como análise diferencial, cube attack ou rainbow table; ataque de força bruta é sempre a primeira tentativa para aqueles métodos considerados atualmente seguros. A sua segurança entretanto advém do cálculo do tempo que se leva a descoberta de uma colisão com a capacidade computacional atual. Entretanto se não forem definidas políticas corretas para definição das senhas, independente da função Hash, a segurança ficará exposta.

Para demonstrar que é possível atacar um banco de dados de senhas, independente da função hash definida, utiliza-se SHA1 e MD5 apenas como controle de referencia e SHA2 256 e SHA3 512 como comparativo. As tentativas se concentram em cinco rodadas pra cada algoritmo de acordo com o tamanho do dicionário: 17 caracteres mais comuns, 27 letras minúsculas, 53 letras, todos os 63 caracteres alpha-numéricos e todos os 89 caracteres imprimíveis. Cada uma fazendo as tentativas com comprimentos de senha de 4 caracteres em diante com tempo limite de processamento em 20 segundos para cada hash. O objetivo deste teste não é simular um ataque real, visto que uma consulta a uma tabela seria muito mais eficiente, mas analisar **em média** qual a facilidade de se obter uma pré-imagem.

A enumeração usada das tentativas foi através de uma função recursiva:

```
Função gera ( comprimento n )
dicionario=”abcde”
se n = 0
        imprime palavra
senão
para cada letra em dicionário
        palavra = letra + gera (n-1)
```

No código real a lista das letras (dicionário) usadas está na ordem em que elas mais ocorrem. De forma que a letra mais comum (de maior probabilidade) é colocada na primeira posição. Isso faz com que as letras de maior probabilidade sejam preenchidas nas primeiras posições, e portanto sejam testadas antes. No final todas as alternativas serão geradas igualmente, mas na média o acerto é maior. Além disso a implementação em C também considera um processo para cada primeiro nível da recursão, sendo assim pode utilizar vários processadores ao mesmo tempo. Cada thread do processo então já tem uma letra definida na primeira posição, sendo o número de threads o tamanho

do dicionário.

```
Forca_Bruta(tamanho)
/*Realiza força bruta em palavra de tamanho especifico */
pra cada Letra em dicionario
        Palavra[1] = Letra /*Primeira posição da palavra recebe a letra */
        chama thread (Resto da Palavra,tamanho-1) /*O resto da palavra será processada em uma thread*/

pra cada Letra em Dicionario
        aguarda fim da thread
retorna;
```

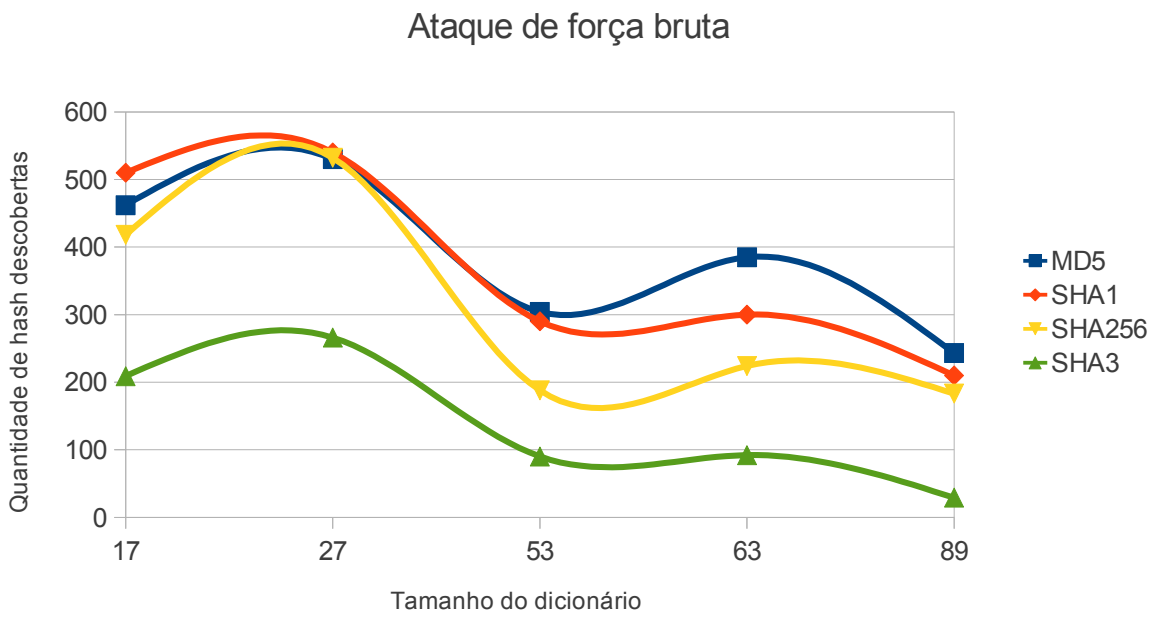
Como dito anteriormente, foi usado em todos os testes 5 tipos de dicionários. De tamanhos variando de 17 a 89.

4 Experimentação e resultados

4.1 Tempo de quebra com força bruta

Os testes com dicionários menores (somente caracteres mais comuns) se mostraram mais eficientes às tentativas com mais caracteres. Esse resultado não é tão evidente quanto parece a primeira vista, porque como existe um limite de tempo de 20 segundos, o número de tentativas deve ser exatamente o mesmo para cada rodada. Os testes com os caracteres mais comuns conseguem testar senhas de comprimento até 7 caracteres nesses 20 segundos, enquanto o teste com todos os caracteres imprimíveis não ultrapassa comprimento 5.

Para os referidos testes foi utilizado um computador com processador Intel Core i3 2.7 Ghz, com 2 núcleos, e um código em C com compilador GCC rodando em GNU/Linux 2.6.38.



É possível notar no gráfico que é mais eficiente testar caracteres comuns e suas combinações a utilizar todo o dicionário de caracteres imprimíveis. Como na amostra é possível encontrar todo tipo de senha menos tempo de processamento foi necessário para encontrar as palavras simples. Outro ponto que chama atenção se deve ao fato de que ao todo 862 *hashes* MD5 únicas foram

descobertas em todos os testes (43% da amostra) com apenas 20 segundos de processamento.

Por outro lado os testes com SHA1, usando a mesma lista, não se demonstraram em nada mais seguros ao MD5. O número total de *hashes* únicas foi de 830 (41,5% da amostra), o que tecnicamente é o mesmo nível de segurança pra este tipo de ataque.

Em todos os testes há um comportamento não linear, com pequeno aumento nos dicionários de comprimento 27 e 63. No primeiro caso (dicionário de tamanho 27) ao diminuir o dicionário a tamanhos inferiores a um alfabeto as combinações em média não são capazes de gerar palavras completas (usadas em senhas). Há portanto em função de duas variáveis (tamanho das palavras e tamanho do dicionário) um ponto de máximo que está num alfabeto mais completo. No segundo caso (dicionário de tamanho 63) esse comportamento se deve a divisão dos dicionários que por exemplo, ao incluir um grupo de caracteres maiúsculos antes de incluir alguns dígitos muito comuns, insere caracteres menos prováveis e diminui a chance de descoberta da hash.

Outro teste que podemos inferir é da garantia de descoberta para cada dicionário e comprimento. O tempo que se leva para exaurir todas as possibilidades para cada comprimento de senha, em função de cada tamanho de dicionário pode ser visto na tabela a seguir:

dicionário\comprimento	4	5	6	7
17	0,014s	0,09s	1,3s	20s
27	0,04s	0,8s	19s	9m 57s
53	0,51s	21s	15m	12h 35m
63	0,96s	50s	50m	2d
89	2,96s	3m 55s	5h	16d

Tabela 6: Tempo levado para se testar todas as possibilidades do MD5

Como pode ser visto, para o MD5 todas as possibilidades garantem uma descoberta da pré imagem primária em tempo relativamente possível. Só o ataque de força bruta por si só já desqualifica o MD5 para ser usado no armazenamento de senhas. O mesmo teste com SHA1 mostra, que apesar de mais difícil, não apresenta dificuldade que refute completamente sua insegurança.

Dicionario\tamanho	4	5	6	7
17	0,029s	0,235s	3,722s	1m 7s
27	0,1s	2,2s	1m 6s	29m
53	1,3s	1m 7s	1h	2d
63	2,6s	2m 30s	2h 35m	6d
89	9,2s	12m 25s	16h	54d

Tabela 7: Tempo levado para se testar todas as possibilidades do SHA1

Por outro lado o teste com SHA2 256 mostra que com capacidade computacional modesta ainda é relativamente difícil descobrir uma imagem desde que os requisitos da senha sejam bem definidos. Infelizmente com o avanço da velocidade dos processadores este quadro pode se tornar obsoleto em alguns anos:

Dicionario\tamanho	4	5	6	7
17	0,05s	0,594s	9,76s	2m
27	0,23s	5,9s	2m 40s	1h
53	3,20s	2m 40s	2h	4d
63	6,40s	6m	6h	15d
89	23,68s	30m	2d	160d

Tabela 8: Tempo levado para se testar todas as possibilidades do SHA256

Esse aumento da segurança em comparação ao MD5 e ao SHA1, se deve exclusivamente ao

custo de processamento do SHA2. Caso fosse possível implementar uma função hash em hardware para que o custo de processamento ficasse próximo de um curto ciclo de máquina, somente a enumeração das possibilidades já representaria um custo considerável:

Dicionario\tamanho	4	5	6	7
17	0,003s	0,013s	0,1s	1,4s
27	0,008s	0,06s	1,3s	33s
53	0,045s	1,38s	1m 10s	1h
63	0,087s	3,33s	3m 3s	3h
89	0,216s	16,3s	21m 40s	1d 4h

Tabela 9: Tempo levado para se enumerar todas as possibilidades de senhas

O algoritmo SHA3 como de se esperar é o mais difícil entre todos os candidatos, sendo uma linha bem abaixo dos outros. Entretanto também possibilita a descoberta por si só de uma quantidade razoável de senhas (pouco menos de um quarto da amostra total).

Sob o ponto de vista da criptografia o que garante a segurança é a impossibilidade temporal e econômica de processamento reverso. O tempo de processamento deve ser maior que a própria validade da informação, mesmo em máquinas potentes. Isso só é obtido com aumento da complexidade do ataque de força bruta. Infelizmente o uso de caracteres excessivamente simples reduz a base da exponencial de possibilidades, ao mesmo tempo que comprimentos muito curtos são expoentes muito pequenos. A verdadeira segurança do armazenamento de senhas (no contexto de strings pequenas e fáceis de lembrar) só acontece quando os dois critérios são aumentados independente do algoritmo. Essa ideia específica do contexto de senhas é comprovada pelos testes em que nenhuma hash "descoberta" era uma segunda pré imagem.

5 - Conclusão

Muitas são as formas de se armazenar as senhas dos usuários nos sistemas que utilizam essa autenticação. O algoritmo utilizado é fundamental para que esse seja um armazenamento seguro, entretanto esse fator isoladamente não oferece segurança para todos os usuários. São diversos ataques diferentes que podem ser realizados as bases de dados criptografadas e tanto o uso do salt de uma forma inteligente quanto as políticas das senhas devem ser implementadas.

Com base nos testes, podemos concluir que senhas de pouco comprimento, ou senhas com caracteres excessivamente comuns (independente do comprimento) são fáceis de descobrir seja qual for a forma criptografada. Os resultados deixam claros que somente com uma boa política de senhas: com comprimento mínimo de 8 ou 9 caracteres e obrigatoriedade de caracteres diversos (letras, número e caracteres especiais) pode-se trazer uma boa segurança. Esse requisito levaria a mais de um dia para simples enumeração de todas as possibilidades e mais de meses para qualquer algoritmo citado.

O resumo total da análise deste trabalho tem a mesma conclusão de todas as área da segurança da informação: Não existe método totalmente seguro e aprova de ataques, entretanto com o uso de todos os recursos possíveis pode-se aumentar a segurança a colocando em patamares que minimizam seu impacto.

6- Referências Bibliográficas

[1] D Roig - US Patent 7,509,495, 2009; Google Patents

[2] Mihir Bellare , Ran Canetti , Hugo Krawczyk - Keying hash functions for message authentication (1996)

[3] Cube Attacks on Cryptographic Hash Functions

[4] Xiaoyun Wang and Hongbo Yu, How to Break MD5 and Other Hash Functions

[5] FIPS PUB 180-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Secure Hash Standard (SHS)

[6] R. Rivest, RFC1320; MIT Laboratory for Computer Science and RSA Data Security 1992

[7] R. Rivest, RFC1321; MIT Laboratory for Computer Science and RSA Data Security 1992

[8] RIVEST, Ronald L. - The MD6 hash function A proposal to NIST for SHA-3

[9] Wikimedia common, Birthday Paradox http://commons.wikimedia.org/wiki/File:Birthday_Paradox.svg

[10] Lathrop , Joel - Analysis of NIST Cryptographic Hash Function

[11] Ivan Bjerre Damgard, A Design Principle of Hash Functions

[12] William Stallings, Cryptography and Network Security: Principles and Practice

[13] Alexander, Steven. Password Protection for Modern Operating Systems - USENIX Association Junho 2004

[14] Alan Kaminsky, Cryptography Lecture Notes. Rochester Institute of Technology http://www.cs.rit.edu/~ark/fall2012/482/module11/notes.shtml

[15] Top Network Security Tools. http://sectools.org/

[16] CreamCracker Brute Force Password Cracker http://code.google.com/p/creamcracker/

[17] Lista de 2 mil senhas usadas nos testes http://creamcracker.googlecode.com/svn/wiki/passwordList_2k.txt