

BIOHACKING, DEEP WEB E CRIPTOGRAFIA

# HASH

ALESSANDRO VINÍCIUS VIEIRA



7

## LISTA DE FIGURAS

Figura 7.1 - Cálculo de uma função Hash .....	5
Figura 7.2 - Criando um arquivo texto no Linux.....	12
Figura 7.3 - Gerando um Hash MD5 no Linux.....	13
Figura 7.4 - Site de download do Kali Linux informando o hash de verificação das ISOs .....	14
Figura 7.5 - Gerando <i>hash</i> SHA-256 no Linux .....	15
Figura 7.6 - Gerando hash SHA-512 no Linux .....	16

EMANIP

## SUMÁRIO

7 HASH .....	4
7.1 Introdução .....	4
7.2 O que são as funções hash? .....	4
7.3 Para que serve uma função hash? .....	6
7.4 Funções hash mais utilizadas .....	10
7.5 Lab .....	12
REFERÊNCIAS .....	18

FELIPE BRENO SUGISAWA ALTRAN

## 7 HASH

### 7.1 Introdução

É comum encontrar quem confunda Criptografia, principalmente falando dos princípios de cifragem e encodamento, com função *hash*. Porém, apesar de visualmente parecidos, são técnicas matemáticas completamente distintas.

As funções *hash* têm um papel fundamental no campo da Segurança da Informação, pois por meio delas é possível garantir princípios como integridade e confidencialidade às implementações de sistemas informáticos modernos.

Neste capítulo, vamos apresentar os conceitos sobre as funções *hash*, suas definições matemáticas, as famílias das funções mais utilizadas e ainda as possibilidades de sua implementação em nosso dia a dia.

### 7.2 O que são as funções hash?

Conforme Tavares (2016), as funções criptográficas de hash, ou simplesmente, funções hash, possuem diversas características interessantes e que, num primeiro momento, podem fazer você se perguntar se elas possuem alguma utilidade.

De acordo com Serafim (2012), funções de resumo (hash) recebem uma entrada e produzem uma saída, normalmente chamada de *hash-code*, resultado (ou valor) do resumo, ou simplesmente resumo. Uma função de resumo  $h$  mapeia cadeias de bits de comprimento arbitrário (embora finito) em cadeias de bits de tamanho fixo ( $n$ ). Para um dado domínio  $D$  e um contradomínio  $R$ , com  $h : D \rightarrow R$  e  $|D| > |R|$ , a função é de muitos para um, o que implica necessariamente a existência de colisões, isto é, valores de  $x \neq x^1$  tais que  $f(x) = f(x^1)$ .

As funções de resumo são utilizadas para muitas aplicações não relacionadas à criptografia (métodos de busca, por exemplo) e, nesse escopo, trataremos apenas

das funções de resumo (hash resumo) criptográfico (cuja propriedade será vista adiante) e serão chamadas simplesmente de funções de resumo.

Se considerarmos que o domínio de  $h$  consiste em entradas de  $t$  bits ( $t > n$ ) e que a função  $h$  é aleatória (no sentido de que todas as saídas são igualmente prováveis) então cerca de  $2^{t-n}$  entradas seriam mapeadas para uma mesma saída e duas entradas escolhidas aleatoriamente gerariam a mesma saída com probabilidade  $2^{-n}$  (independentemente de  $t$ ). A ideia é que o valor do resumo sirva como uma representação compacta (às vezes chamada de resumo criptográfico, *imprint*, *digital fingerprint* ou *message digest*) da cadeia de entrada e pode ser usada como se identificasse unicamente aquela cadeia.

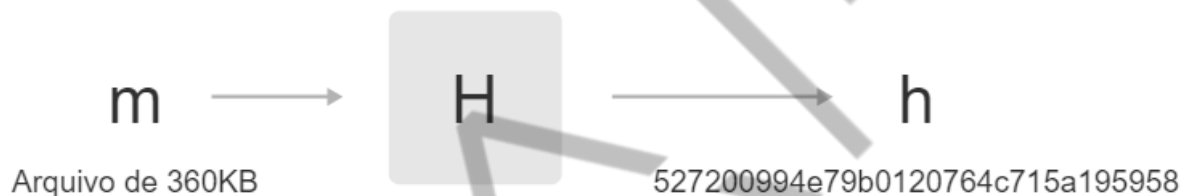


Figura 7.1 - Cálculo de uma função Hash  
Fonte: Elaborado pelo autor (2020)

Segundo Tavares (2016), e de acordo com a Figura “Cálculo de uma função Hash”, 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 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.

Tavares (2016) nos explica que a terceira característica é a resistência a colisões. Se, ao calcularmos o *hash* de duas mensagens diferentes com uma determinada função 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 ( $\infty$ ). 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  $2^{160}$ . Os números  $2^{128}$  ou  $2^{160}$  são muito grandes, porém o infinito é... infinito. Sendo assim, podemos afirmar que para qualquer função de hash haverá colisões. Portanto, resistência a colisões não significa que não deva existir colisões, mas, sim, que: embora colisões existam, elas não podem ser encontradas.

### 7.3 Para que serve uma função hash?

De acordo com Tavares (2016), alguns dos principais usos para as funções *hash* são:

- Verificação de integridade de arquivos.
- Armazenamento de senhas.
- Resolução do problema da velocidade na assinatura digital.

#### a) Verificação de integridade de arquivos

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

Alice pode calcular o *hash* “h” do seu arquivo “m” e guarda-lo em algum local seguro:

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

Alice: [h]

Alice, então, envia apenas o arquivo  $m$  para ser armazenado no Servidor:

Alice:  $m > \text{Servidor}$

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

Alice:  $m < \text{Servidor}$

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. Neste momento, basta que Alice compare esses dois valores:

Alice:  $i = (h == h')$

Se  $h$  for igual a  $h'$ , 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 (por exemplo: no seu próprio computador). Se ela armazenasse o *hash*  $h$  junto ao arquivo “ $m$ ” no servidor, nada impediria que um atacante alterasse o arquivo e então calculasse novamente o hash e substituísse o valor calculado por 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 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, ao calcular o hash do arquivo, Alice adicionaria uma chave de seu conhecimento apenas:

Alice: $h = H(K,m)$
---------------------

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

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:

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

Alice: $h' = H(K,m)$
----------------------

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

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 por Alice pois ele não conhece a chave por ela utilizada.

#### b) Armazenamento de senhas

A forma correta de armazenar senhas é com a utilização de funções hash! Eis a solução correta: o primeiro passo é o cadastramento da senha feito pelo usuário. O usuário informa a sua senha ao sistema:

Usuário: senha > Sistema
--------------------------

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



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 seu hash.

Usuário: senha > Sistema

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

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 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 resolver. 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 (ou *Salt*, do inglês) é alguma informação randômica e não sigilosa que é juntada à senha antes de calcular o hash. Assim, vamos rever nosso processo.

c) Cadastramento da senha:

Usuário: senha > Sistema
Sistema: $h = H(\text{senha} + \text{sal})$

Agora o sistema gerou um sal (por exemplo: 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: $h' = H(\text{senha} + \text{sal})$

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. Veja como ficam armazenados os hashes das senhas dos usuários Alice e Bob nesse 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. Esses três caracteres adicionais alteram completamente o valor do hash e, mesmo não sendo sigilosos, não comprometem, em absoluto, a irreversibilidade da função hash utilizada.

## 7.4 Funções hash mais utilizadas

Vamos abordar as famílias de funções *hash* mais utilizadas na computação moderna, conforme Serafim (2012).

A família MD5 foi projetada pela RSA (MD2) e por Ron Rivest (MD4 e MD5) e compreende:

- MD2 - RFC 1319.

- MD4 - RFC 1320.
- MD5 - RFC 1321.

Todos os membros da família geram um resumo de 128 bits. Durante muito tempo o MD5 foi largamente usado, chegando a ser considerado quase um sinônimo de função de resumo. No entanto, devido aos ataques, esta função tem caído em desuso, sendo substituída por outras que têm tamanhos de resumo maiores, como o SHA-1.

Os algoritmos da família SHA (*Secure Hash Algorithm*) foram especificados pelo padrão FIPS (*Federal Information Processing Standards*) 180-2 do US National Institute of Standards and Technology (órgão do governo norte-americano).

O padrão original (FIPS 180-1) especificava apenas o algoritmo conhecido como SHA-1, que tem um resumo (message digest) de 160 bits. O FIPS 180-2 (publicado em agosto de 2002) acrescentou três novos algoritmos (SHA-256, SHA-384 e SHA-512) e uma mudança (publicada em fevereiro de 2004) adicionou o algoritmo SHA-224. A RFC 3174 também descreve o SHA-1, trazendo inclusive uma implementação em linguagem C do algoritmo.

Os algoritmos da família SHA estão relacionados ao MD5 mas utilizam tamanhos de resumo maiores (a partir de 160). Essa semelhança é, inclusive, fonte de críticas, visto que ataques ao MD5 podem eventualmente servir de “entrada” para possíveis ataques às funções da família SHA.

Os algoritmos hoje aceitos pelo padrão são:

- SHA-1.
- SHA-224.
- SHA-256.
- SHA-384.
- SHA-512.

O padrão descreve também uma forma de truncar o resultado para obter resumos de outros comprimentos (por exemplo 96 bits), como conveniência para facilitar a interoperabilidade de aplicações utilizando tais resumos. No entanto, o

padrão não faz nenhuma alegação quanto à segurança de tais resumos truncados e proíbe a sua utilização em aplicações padronizadas que referenciam aquele padrão.

## 7.5 Lab

Neste Lab, vamos usar alguns utilitários de linha de comando Linux para geração de hash de determinados arquivos, e para tal, utilizaremos nossa VM do Kali Linux. Porém antes vamos preparar o cenário para execução do nosso LAB.

Abra o terminal e crie um arquivo de texto com o seguinte comando:

```
echo "Aula de funções hash - FIAP" > arquivo.txt
```

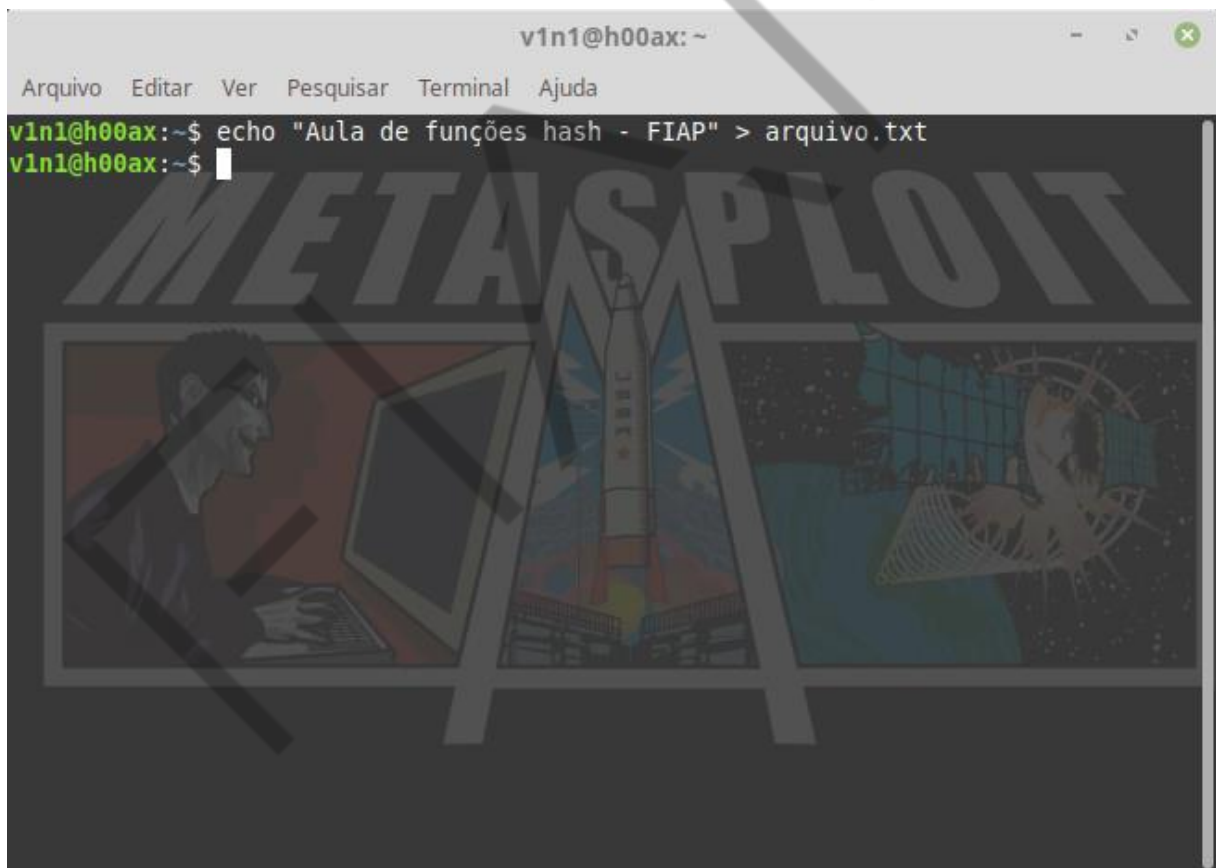


Figura 7.2 - Criando um arquivo texto no Linux  
Fonte: Elaborado pelo autor (2020)

A partir desse momento já podemos prosseguir na execução do nosso laboratório.

a) Verificando o hash MD5

Para verificar manualmente o hash MD5 do seu arquivo basta executar:

```
md5sum arquivo.txt
```

A saída do comando será semelhante à da Figura “Gerando um Hash MD5 no Linux”:

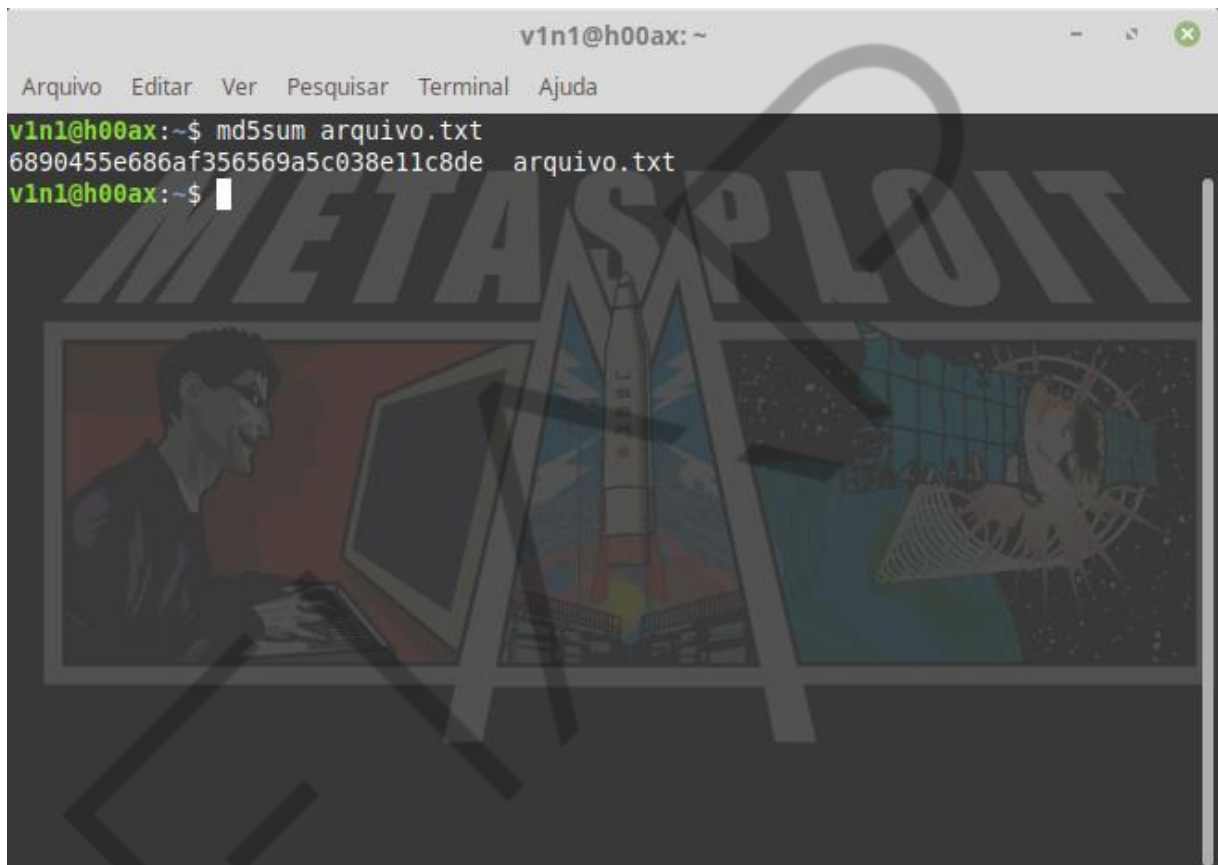



Figura 7.3 - Gerando um Hash MD5 no Linux  
Fonte: Elaborado pelo autor (2020)

Se o arquivo alvo do LAB fosse uma ISO baixada da internet, por exemplo, você poderia utilizar o md5sum para conferir se o download foi realizado sem perda de pacotes. Bastaria, portanto, verificar se o *hash* gerado era idêntico ao informado no site.



## Download Kali Linux Images

We generate fresh Kali Linux image files every few months, which we make available for download. This page provides the links to download Kali Linux in its latest official release. For a release history, check our Kali Linux Releases page. Please note: You can find unofficial, untested weekly releases at <http://cdimage.kali.org/kali-weekly/>. Downloads are **rate limited to 5 concurrent connections**.

Image Name	Download	Size	Version	SHA256Sum
Kali Linux Light Armhf	<a href="#">HTTP</a>   <a href="#">Torrent</a>	694M	2019.1a	3aaf70621ab0313b14259780d2c50334e23445fc8b27aae30f5b3be94fafad8b
Kali Linux Mate 64 Bit	<a href="#">HTTP</a>   <a href="#">Torrent</a>	3.2G	2019.1a	179135fe3abc6ee365558db88e9abc1c32f634c9118d053245390e5c44a9013f

Figura 7.4 - Site de download do Kali Linux informando o hash de verificação das ISOs

Fonte: Elaborado pelo autor (2020)

Também é possível checar "automaticamente" se o arquivo está ou não corrompido, para isso você deverá baixar e salvar o arquivo de verificação MD5 na mesma pasta onde baixou o seu pacote/software/ISO (geralmente disponibilizado na própria página de download) e executar o comando `md5sum` com a opção `-c` (checar):

```
md5sum -c [arquivo-de-verificação]
```

A saída do comando (no meu caso utilizei o pacote LibreOffice) foi a seguinte:

```
$ md5sum -c LibreOffice_5.2.2_Linux_x86-64_rpm.tar.gz.md5
LibreOffice_5.2.2_Linux_x86-64_rpm.tar.gz: SUCESSO
```

Caso apareça a mensagem "SUCESSO" ao rodar o comando acima, indica que o seu arquivo está íntegro, mas se aparecer a mensagem "FALHOU", indica que o seu arquivo está corrompido, nesse caso você terá que baixá-lo novamente.

b) Verificando o hash SHA-256

O processo é semelhante ao MD5 descrito acima, mudando somente o comando. Para verificar manualmente o hash SHA-256 do seu arquivo basta executar:

```
sha256sum arquivo.txt
```

A saída do comando será semelhante à esta da Figura “Gerando hash SHA-256 no Linux”.

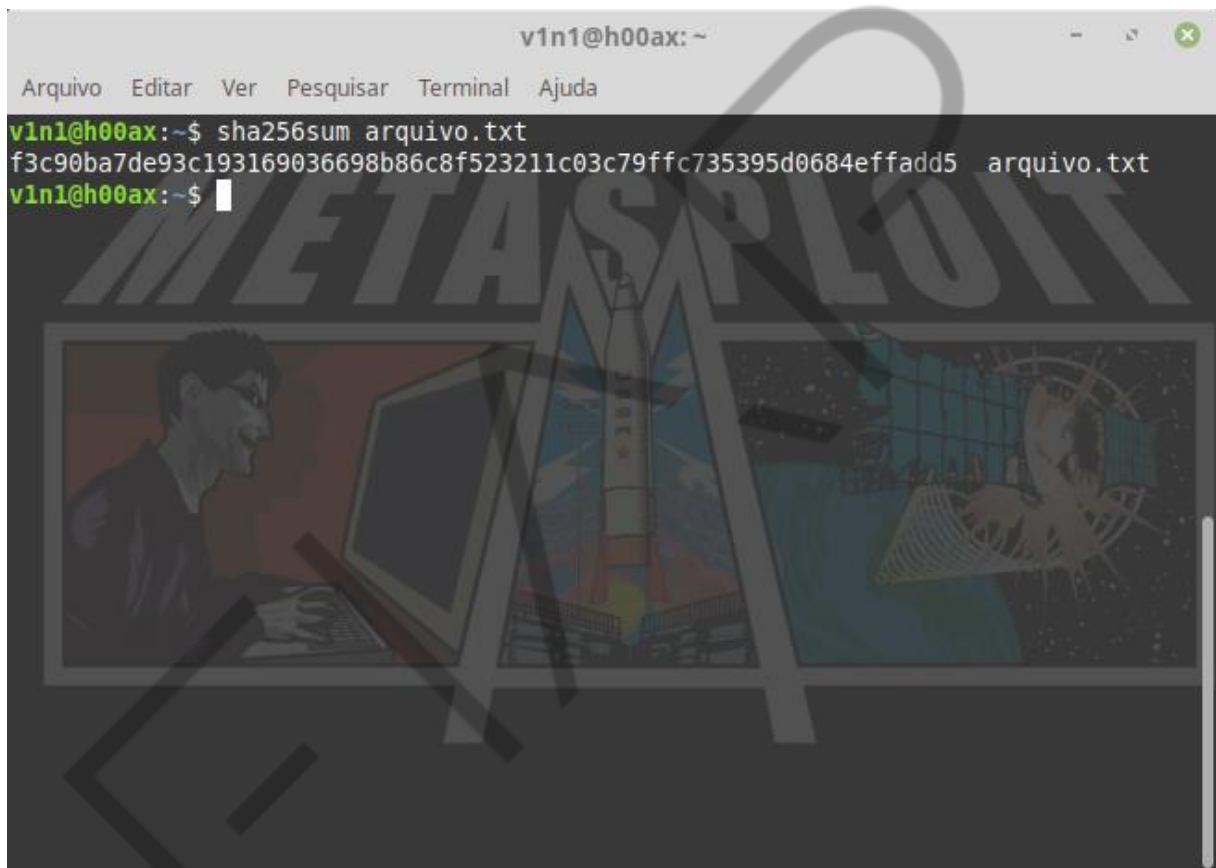


Figura 7.5 - Gerando *hash* SHA-256 no Linux  
Fonte: Elaborado pelo autor (2020)

Da mesma forma que o MD5, você pode comparar o hash alfanumérico que aparecerá no terminal com o hash SHA-256 disponibilizado no site onde baixou seu arquivo, se forem iguais, isso indica que seu arquivo está íntegro, mas se o resultado obtido no terminal for diferente, o arquivo está corrompido, neste caso você deverá fazer novamente o download do arquivo.

Para verificar "automaticamente" se o arquivo está ou não corrompido você deverá baixar e salvar o arquivo de verificação SHA-256 na mesma pasta onde baixou o seu pacote/software/ISSO. Execute o comando sha256sum com a opção -c (checar):

```
sha256sum -c [arquivo-de-verificação]
```

### c) Verificando o hash SHA-512

O processo é semelhante ao do SHA-256 descrito acima, mudando somente o comando. Para verificar manualmente o *hash* SHA-512 do seu arquivo, basta executar:

```
sha512sum arquivo.txt
```

A saída do comando será semelhante a da Figura “Gerando hash SHA-512 no Linux”

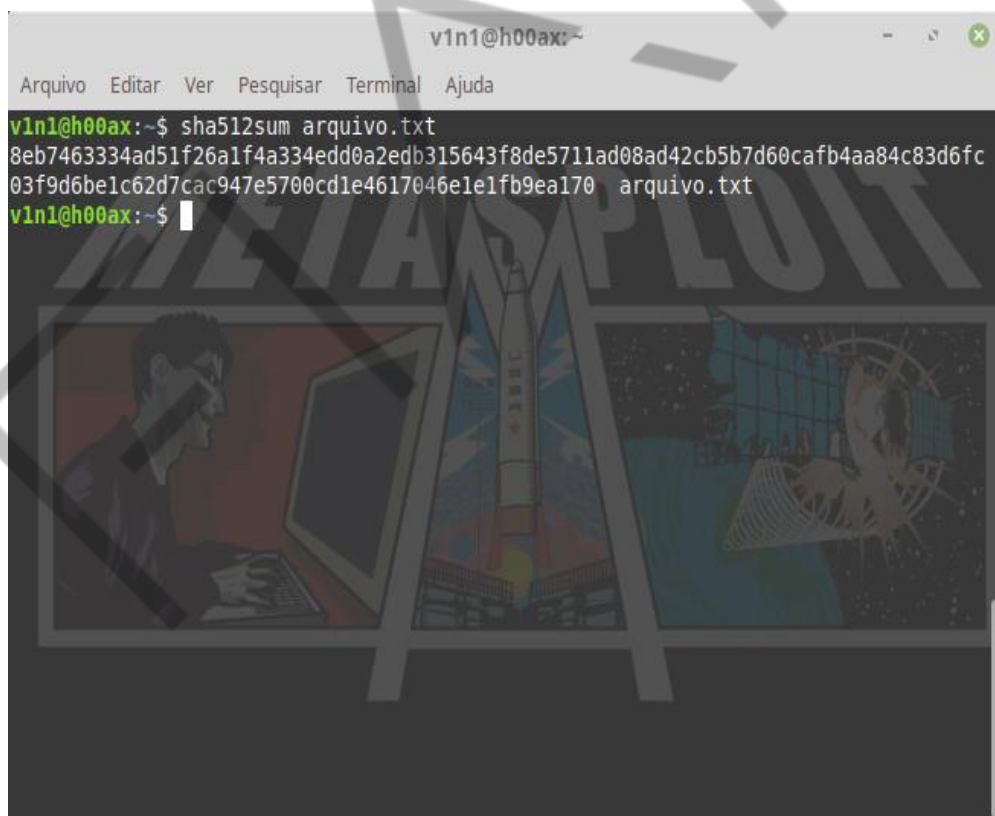


Figura 7.6 - Gerando hash SHA-512 no Linux  
Fonte: autor (2020)

Da mesma forma que o SHA-512, você pode comparar o *hash* alfanumérico que aparecerá no terminal com o *hash* SHA-512 disponibilizado no site em que baixou seu arquivo. Se forem iguais, indica que seu arquivo está íntegro, mas se o



resultado obtido no terminal for diferente, o arquivo está corrompido, neste caso você deverá fazer o download do arquivo novamente.

Para verificar "automaticamente" se o arquivo está ou não corrompido você deverá baixar e salvar o arquivo de verificação SHA-512 na mesma pasta onde baixou o seu pacote/software/ISO, execute o comando sha512sum com a opção -c (checar):

```
sha512sum -c [arquivo-de-verificação]
```

## REFERÊNCIAS

SERAFIM, Vinícius da Silveira. **Introdução à Criptografia: Funções Criptográficas de Hash.** 2012. Disponível em: <[http://www.serafim.eti.br/academia/recursos/Roteiro\\_08-Funcoes\\_de\\_Hash.pdf](http://www.serafim.eti.br/academia/recursos/Roteiro_08-Funcoes_de_Hash.pdf)>. Acesso em: 16 jun. 2020.

TAVARES, Paulo Henrique Nóbrega. **Estudo e implementação de algoritmos de resumo (hash) criptográfico na plataforma Intel® XScale®.** 2006. Disponível em: <[http://repositorio.unicamp.br/bitstream/REPOSIP/276231/1/Tavares\\_PauloHenrique\\_M.pdf](http://repositorio.unicamp.br/bitstream/REPOSIP/276231/1/Tavares_PauloHenrique_M.pdf)>. Acesso em: 16 jun. 2020.