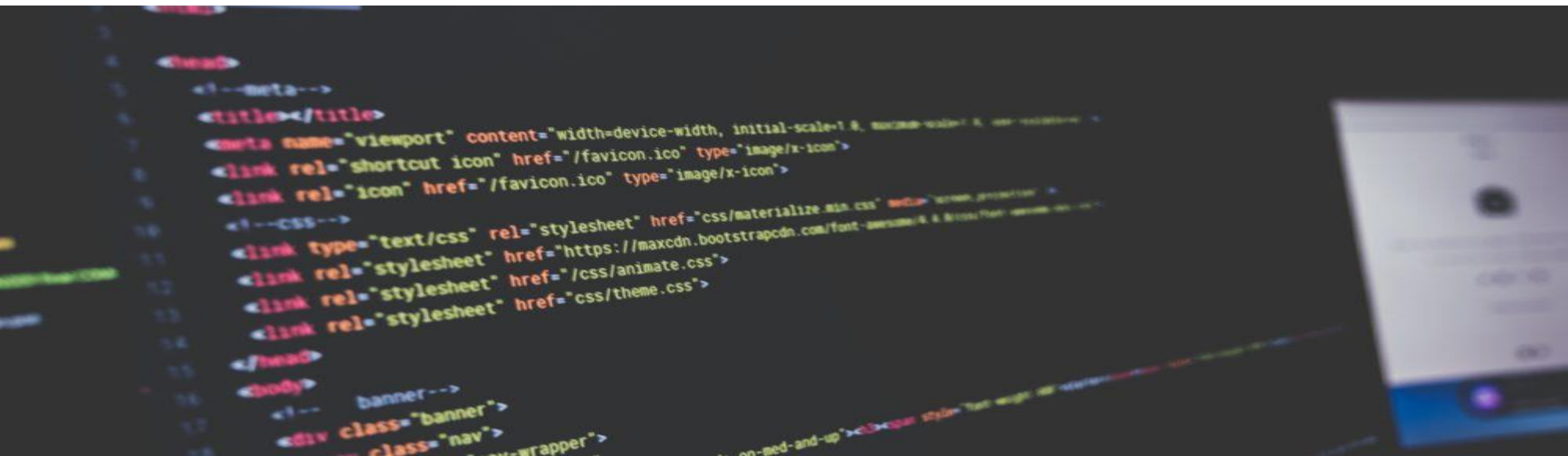




Faculdade de Tecnologia Senac Pelotas
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Programação Web

Prof. Edécio Fernando Iepsen



Autenticação do Usuário

Processo que visa garantir em um sistema que um usuário realmente é quem ele afirma ser.



Criptografia de Senhas com BCrypt



bcrypt - npm

npmjs.com/package/bcrypt

DT

5.0.1 • Public • Published 2 years ago

Readme

Explore BETA

2 Dependencies

3.480 Dependents

52 Versions

node.bcrypt.js

build failing

Dependency Status

A library to help you hash passwords.

You can read about [bcrypt in Wikipedia](#) as well as in the following article: [How To Safely Store A Password](#)

If You Are Submitting Bugs or Issues

Verify that the node version you are using is a *stable* version; it has an even major release number. Unstable versions are currently not supported and issues created while using an unstable version will be closed.

If you are on a stable version of node, please provide a sufficient code snippet or log files for installation issues. The code snippet does not require you to include confidential information. However, it must provide enough information such that the problem can be replicable. Issues which are closed without resolution often lack required information for replication.

Version Compatibility

Install

```
> npm i bcrypt
```

Repository


[github.com/kelektiv/node.bcrypt.js](#)

Homepage

[github.com/kelektiv/node.bcrypt.js#rea...](#)

Weekly Downloads

824.281



Version

5.0.1

License

MIT

Unpacked Size

119 kB

Total Files

25

Issues

11

Pull Requests

1



sync

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const myPlaintextPassword = 's0/\\P4$$w0rd';
const someOtherPlaintextPassword = 'not_bacon';
```

To hash a password:

Technique 1 (generate a salt and hash on separate function calls):

```
const salt = bcrypt.genSaltSync(saltRounds);
const hash = bcrypt.hashSync(myPlaintextPassword, salt);
// Store hash in your password DB.
```

Technique 2 (auto-gen a salt and hash):

```
const hash = bcrypt.hashSync(myPlaintextPassword, saltRounds);
// Store hash in your password DB.
```

As with async, both techniques achieve the same end-result.

To check a password:

```
// Load hash from your password DB.
bcrypt.compareSync(myPlaintextPassword, hash); // true
bcrypt.compareSync(someOtherPlaintextPassword, hash); // false
```

Uma breve introdução sobre BCrypt

Atualmente, o mercado de tecnologia possui uma gama de informações sensíveis e suscetíveis a qualquer usuário, que são essenciais para os negócios de uma organização.

Para evitar acessos não autorizados aos sistemas, utilizam-se alguns métodos de proteção de dados, como a criptografia de senhas, que dificulta o acesso a banco de dados.

| O que é criptografia e algoritmo de hashing?

A palavra criptografia origina-se do grego *kryptós* (“escondido”) e *gráphein* (“escrita”), que significa “tornar informações originais em informações ilegíveis”. Esse processo de “esconder a escrita” é realizado através da aplicação de algoritmos matemáticos que modificam qualquer bloco de dados em caracteres de comprimento fixos, transformando-os em *hashes*. O comprimento dos dados de entrada podem ter qualquer tamanho, porém, os dados de saída sempre terão valores de *hash* de mesmo comprimento. Um dos problemas encontrados no hash é a vulnerabilidade quanto aos ataques de dicionário (força bruta). Um

Tabela arco-íris

Origem: Wikipédia, a enciclopédia livre.

Uma **tabela arco-íris** (do inglês **rainbow table**) é uma **tabela pré-computada** para reverter **funções hash criptográficas**, geralmente para quebrar hashes de senhas. As tabelas são geralmente usadas na recuperação de uma **senha** (ou números de cartão de crédito, etc.) até um determinado comprimento, que consiste em um conjunto limitado de caracteres. É um exemplo prático de uma **troca de espaço-tempo**, usando menos tempo de processamento do computador e mais armazenamento do que um **ataque de força bruta**, que calcula um hash a cada tentativa, mas mais tempo de processamento e menos armazenamento do que uma **tabela de pesquisa** simples com uma entrada por hash. O uso de uma **função de derivação de chave** que emprega um **sal** torna esse ataque inviável.

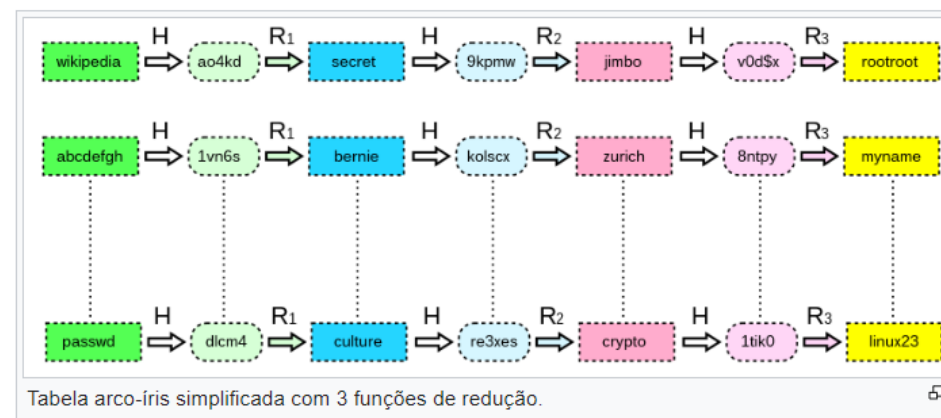
As tabelas do arco-íris foram inventadas por Philippe Oechslin^[1] como uma aplicação de um algoritmo anterior mais simples de **Martin Hellman**.^[2]

É utilizada normalmente para que um ataque contra senhas digeridas seja possível.^[3]

Pano de fundo

Qualquer sistema de computador que exija **autenticação** por senha deve conter um banco de dados de senhas, ofuscadas ou em **texto puro**, e existem vários métodos de **armazenamento de senhas**. Como as tabelas são vulneráveis a roubo, o armazenamento da senha em texto simples é perigoso. Portanto, a maioria dos bancos de dados armazena um **hash criptográfico** da senha de um usuário no banco de dados. Nesse sistema, ninguém - incluindo o sistema de autenticação - pode determinar qual é a senha de um usuário apenas olhando o valor armazenado no banco de dados. Em vez disso, quando um usuário digita uma senha para autenticação, o sistema calcula o valor do hash para a senha fornecida e esse valor é comparado ao hash armazenado para esse usuário. A autenticação é bem-sucedida se os dois hashes corresponderem. Depois de coletar um hash de senha, o uso do referido hash como senha falharia, pois o sistema de autenticação calcularia o hash uma segunda vez. Para aprender a senha de um usuário, uma senha que produz o mesmo valor de hash deve ser encontrada, geralmente por meio de um ataque de força bruta ou de dicionário.

As tabelas arco-íris são um tipo de ferramenta desenvolvida para derivar uma senha, observando apenas um valor de hash. Elas nem sempre são necessárias, pois existem métodos mais simples de reversão de hash disponíveis. **Ataques de força bruta** e **ataques de dicionário** são os métodos mais diretos disponíveis. No entanto, eles não são adequados para sistemas que usam senhas longas devido à dificuldade de armazenar todas as opções



Um sal é uma camada extra de segurança que é adicionada no início do hash para manter as senhas seguras no caso de um ataque de hash pré-computado. O sal é um número que denota a complexidade do hash. Quanto maior o valor, mais tempo leva para a senha ser hash.

O método *hash()* também possui uma função de retorno de chamada (o retorno de chamada terá parâmetros como erro e resultado) e retorna um Promise que precisa ser tratado. Abaixo está o código para usar a função hash. É pegar dados de um formulário. Inicie seu servidor e execute este código. Estou usando Nodejs e MongoDB para meu banco de dados.

As senhas com hash se parecem um pouco com isso.

```
$2a$12$cEzeMCfft3esD42IWS1eu.gSRa1BYuViQEON51Q41ZI3ooqjHnLw21234
```

A primeira é uma senha com hash. A segunda é a senha original. Os primeiros 7 caracteres são o sal gerado pelo *SALT_ROUND*. Aqui seu valor é 12.

O segundo método é o método *hashSync()*, que é bastante simples, pois requer apenas 2 parâmetros. O primeiro é o dado ou string que deve ser hash e o segundo é o saltround. Ao contrário de *hash()*, esta função não retorna uma promessa e, em vez disso, retorna a saída com hash. Eu prefiro este método porque não envolve Promises e é mais simples. Se quiser saber mais sobre

<https://levelup.gitconnected.com/using-bcrypt-to-hash-and-compare-passwords-with-nodejs-and-mongodb-366ff80138b7>

[Casa](#)

PÚBLICO

Questões

Tag

Comercial

Empresas

COLETIVOS

Explorar Coletivos

EQUIPES

Stack Overflow for Teams – Comece a colaborar e

Não há uma resposta? [Pergunte em Stack Overflow em Português.](#)



86



1. Com "rodada de sal" eles realmente significam o *fator de custo* . O fator de custo controla quanto tempo é necessário para calcular um único hash BCrypt. Quanto maior o fator de custo, mais rodadas de hash são feitas. Aumentar o fator de custo em 1 dobra o tempo necessário. Quanto mais tempo for necessário, mais difícil será a força bruta.
2. O salt é um valor aleatório e deve ser diferente para cada cálculo, portanto, o resultado dificilmente deve ser o mesmo, mesmo para senhas iguais.
3. O sal geralmente é incluído na sequência de hash resultante em forma legível. Assim, ao armazenar a hash-string, você também armazena o sal. Dê uma olhada nesta [resposta](#) para mais detalhes.

Compartilhar Seguir

editado em 25 de novembro de 2018 às 12:00

resposta em 12 de outubro de 2017 às 12:00



Alexis Wilke

17k ● 10 ● 72 ● 125



martinstoeckli

22,1 mil ● 4 ● 54 ● 83

```
$2y$10$nOUIs5kJ7naTuTFkBy1veuK0kSxUFxfuaOKdOKf9xYT0KKIGSJwFa
| | | |
| | | | hash-value = K0kSxUFxfuaOKdOKf9xYT0KKIGSJwFa
| | | |
| | salt = nOUIs5kJ7naTuTFkBy1veu
| |
| cost-factor = 10 = 2^10 iterations
|
hash-algorithm = 2y = BCrypt
```

O sal que você encontra após o terceiro \$, ele é gerado automaticamente pelo `password_hash()` usando a fonte aleatória do sistema operacional. Como o sal está incluído na string resultante, a função `password_verify()`, ou na verdade a função `crypt` encapsulada, pode extraí-lo de lá e calcular um hash com o mesmo sal (e o mesmo fator de custo). Esses dois hashes são então comparáveis.

Todos os anos, a desenvolvedora de softwares de segurança [NordPass](#) divulga as senhas mais utilizadas pelos usuários em todo o mundo. Em 2021, **a lista reúne 200 senhas que mostram que as sequências mais comuns são também as mais simples e óbvias** – o que é altamente preocupante e revela sua vulnerabilidade.

A lista global inclui variações das sequências numéricas mais comuns, assim como a famosa “qwerty” (nada mais do que a sequência da primeira linha de letras do teclado). Confira:

Ranking Global 2021: as 10 senhas mais usadas no mundo

1. 123456
2. 123456789
3. 12345
4. qwerty
5. password
6. 12345678
7. 111111
8. 123123
9. 1234567890
10. 1234567

Ranking Brasileiro 2021: as 10 senhas mais usadas no país

1. 123456
2. 123456789