**Basic Encryption**

Up to this point, we haven't really discussed the need to secure our data. Internet hackers are notorious for trying to guess passwords and generally just make a mess of things. As web developers, we are responsible for keeping our users' data safe.  That means storing the data safely as well. Storing passwords as-is is never a good way to keep data secure. Imagine if a user's password was stored as-is on a database somewhere and a hacker got a hold of it. Now imagine if **all**of your users' passwords were stored as-is in a database. Can you say bedlam?

This is why we encrypt users' passwords **before**we put them into our database. By encrypting, we mean to mask the value of an input by using an algorithm that changes its value. Encryption and cryptography are a complicated and fascinating field of study that is heavily mathematical in nature. That being said, you don't need a Ph.D. to protect your data; you just need to know a few handy tools! This tab will show you basic md5 encryption, take it to the next level by generating a salt, and finally using the powerful bcrypt engine to encrypt passwords.

**Basic md5 encryption**

The basic way to encrypt something is to use md5 encryption.  Md5 encryption is a technique that takes an input (most likely a password) and returns an encrypted string. To see how this works, type the following into a sublime document:

import md5 # imports the md5 module to generate a hash

password = 'password';

# encrypt the password we provided as 32 character string

encrypted\_password = md5.new(password).hexdigest();

print encrypted\_password; #this will show you the encrypted value

# 5f4dcc3b5aa765d61d8327deb882cf99 -> nice!

Copy

What you will see is the crazy string above. The md5 function returns an encrypted version of the parameter it is passed. The md5 function yields the same value every time it is run with the same argument. That means the value it spits back is not random -- it is encrypted. You may ask: why don't hackers just reverse engineer the values? Answer: the md5 algorithm is well known and solvable but it is very strong in the sense that it is not easily reverse engineered. Md5 is not the industry standard but it is a good starting point.

**How to use md5**

When you add your users to the database upon registration, you should save their passwords as an encrypted md5 string. Similarly, when they log in, you should encrypt the inputted password to make sure it matches with the one saved in the database. Here's the idea:

The user being put into your database:

import md5 # do this at the top of your file where you import modules

@app.route('/users/create', methods=['POST'])

def create\_user():

username = request.form['username']

email = request.form['email']

password = md5.new(request.form['password']).hexdigest();

insert\_query = "INSERT INTO users (username, email, password, created\_at, updated\_at) VALUES (:username,

:email, :password, NOW(), NOW())"

query\_data = { 'username': username, 'email': email, 'password': password }

mysql.query\_db(insert\_query, query\_data)

When your user is trying to log in:

password = md5.new(request.form['password']).hexdigest()

email = request.form['email']

user\_query = "SELECT \* FROM users where users.email = :email AND users.password = :password"

query\_data = { 'email': email, 'password': password}

user = mysql.query\_db(user\_query, query\_data)

# do the necessary logic to login if the user exists, otherwise redirect to login page with error message<br>

**Salted encryption**

While md5 is an easy encryption method and safe for small-time projects, it is not secure enough for enterprise-level applications. To really secure your data, you need to make the encryption method more random. By random, we mean more unpredictable. As we said before, md5 encryption is the same no matter what computer runs the code. So to make our encryption more powerful, we will encrypt our data by using what is called a salt. A salt is a "unique" key used to generate a unique password. Example:

salt = '123'; //where the value 123 changes randomly

encrypted\_password = md5(password . '' . salt);

A salt is a string of random characters that will be passed to an encryption method (an *md5())* along with the string we are trying to encrypt (the submitted password) via concatenation. The encryption method that uses the salt is designed in such a way that it takes the salt to compute the encrypted string, using the salt as an 'ingredient' in the encryption 'recipe'.

**Generating a salt**

To generate a salt, you just need to generate a random string of characters. The code below actually uses two different functions to render a random string:

import os, binascii # include this at the top of your file

salt = binascii.b2a\_hex(os.urandom(15))

The function called os.urandom() returns a string of bytes. The number of bytes is equal to the parameter provided.This string isn't a normal alphanumeric string, so we turn it into a string using the function b2a\_hex(), which will turn the value returned from the openSSL function into a normal alphanumeric string. This new random string will be our salt. The idea is to store this salt during the registration process. Example:

username = request.form['username']

email = request.form['email']

password = request.form['password']

salt = binascii.b2a\_hex(os.urandom(15))

encrypted\_pw = md5.new(password + salt).hexdigest()

insert\_query = "INSERT INTO users (username, email, password, salt, created\_at, updated\_at)

VALUES (:username, :email, :encrypted\_pw, :salt, NOW(), NOW())"

query\_data = { 'username': username, 'email': email, 'encrypted\_pw': encrypted\_pw, 'salt': salt}

mysql.query\_db(insert\_query, query\_data)

Now, when we are trying to authenticate a user's login, we do some pretty nifty stuff:

email = request.form['email']

password = request.form['password']

user\_query = "SELECT \* FROM users WHERE users.email = :email LIMIT 1"

query\_data = {'email': email}

user = mysql.query\_db(user\_query, query\_data)

if user[0]:

encrypted\_password = md5.new(password + user[0]['salt']).hexdigest();

if user[0]['password'] == encrypted\_password:

# this means we have a successful login!

else:

# invalid password!

else:

# invalid email!

Below are encrypted passwords generated by concatenating a password ('password' as value) with the salt ( *str\_shuffle('hello')* which we know is very insecure) within the md5 hashing method. Even though we provided very simple values for both password and salt, we still get 5 random encrypted values for password in the database.

**Bcrypt Encryption**

Now that we understand the basics of salted encryption, we can use bcrypt which is a powerful module that helps us to encrypt our passwords in a secure way! To get started, let's install the bcrypt module by opening up our terminal:

pip install flask\_bcrypt

Now to use bcrypt in our Flask Apps we can do this:

from flask import Flask, request, render\_template

from mysqlconnection import MySQLConnector

# imports the Bcrypt module

from flask\_bcrypt import Bcrypt

app = Flask(\_\_name\_\_)

bcrypt = Bcrypt(app)

mysql = MySQLConnector(app, 'my\_database\_here')

# this will load a page that has 2 forms one for registration and login

@app.route('/', methods=['GET'])

def index():

return render\_template('index.html')

# we are going to add functions to create new users and login users

Now we can use the provided bcrypt functions to create encrypted passwords and authenticate when a user is signing in

**generate\_password\_hash**

Bcrypt has a handy password hash generator function that we can use when creating new users. We can use it this way in the create user function:

@app.route('/create\_user', methods=['POST'])

def create\_user():

email = request.form['email']

username = request.form['username']

password = request.form['password']

# run validations and if they are successful we can create the password hash with bcrypt

pw\_hash = bcrypt.generate\_password\_hash(password)

# now we insert the new user into the database

insert\_query = "INSERT INTO users (email, username, pw\_hash, created\_at) VALUES (:email, :username, :pw\_hash, NOW())"

query\_data = { 'email': email, 'username': username, 'pw\_hash': pw\_hash }

mysql.query\_db(insert\_query, query\_data)

# redirect to success page

**check\_password\_hash**

The check\_password\_hash function will return true if the hashed value of password provided at login is equal to the password hash in the database. Otherwise, it will return false. The function works like this:

password = 'password'

pw\_hash = bcrypt.generate\_password\_hash(password)

test\_password\_1 = 'thisiswrong'

bcrypt.check\_password\_hash(pw\_hash, test\_password\_1) # this will return false

test\_password\_2 = 'password'

bcrypt.check\_password\_hash(pw\_hash, test\_password\_2) # this will return true

Now to use this when trying to login we might do it like so:

@app.route('/login', methods=['POST'])

def login():

email = request.form['email']

password = request.form['password']

user\_query = "SELECT \* FROM users WHERE email = :email LIMIT 1"

query\_data = { 'email': email }

user = mysql.query\_db(user\_query, query\_data) # user will be returned in a list

if bcrypt.check\_password\_hash(user[0]['pw\_hash'], password):

# login user

else:

# set flash error message and redirect to login page

**Things to Remember**

While Bcrypt does have its benefits, you should remember a few things when using it:

1. When you create your database, make sure your column *pw\_hash* is of type **varchar(255)**, as opposed to something shorter. Bcrypt hashes are quite long, so this prevents any errors from the hashed password being truncated.
2. Do not store the user-entered (human-readable or 'clear-text') password in the database.