Pratical Security - Passwords

# Overview

One of the most valuable targets in hacking is client passwords since they can provide the key for fraud and forgery on the host webapp and elsewhere on the web. While clients should be separate secure passwords for each site they sign up to, currently this is generally not the case. Therefore, a rule of thumb is that passwords should always be asymettrically encrypted in a database.

# Password Storage

## What to Store

As mentioned above, passwords should always be asymmetically encrypted in a database, this can be done via a one way hashing function such as SHA256 on client registration. A common method of hashing passwords is by using the library ‘BCrypt’ which is kept upto date and provide a secure hashing function with integrated salt.

## Where to Store

Another security aspect of password databases is where they are kept. To prevent comprimise under a attackers sucessful sql injection, the passwords database should be kept in a separate table to the general application data.

# Brute Force

The simplest way for an attacker to figure out a users password is to attempt to login mutliple times, guessing the password by brute force. There are several ways to prevent an attackers ability to perform a brute force attack:

## Limit Number of Requests

The number of requests per ip address, user account, or many other factors can be limited by throttling to a certain amount per time period. Additional measures of banning IP address and locking accounts can also be implemented.

A common way to do this in a rack based application is using the rack-attack middleware. rack-attack can be included in ruby based projects including the gem ‘rack-attack’ in the project Gemfile. In Ruby on Rails the middleware will then be enabled by default, and in other applications it can be enabled using the following code in the config.ru file:

require "rack/attack"

use Rack::Attack

Throttleing can then be setup in the rails file config/initializers/rack\_attack.rb

# Throttle requests to login to 1 per second per ip

Rack::Attack.throttle("requests by ip", limit: 60, period: 1) do |request|

request.ip

end

# Using 503 because it may make attacker think that they have successfully

# DOSed the site. Rack::Attack returns 429 for throttling by default

Rack::Attack.throttled\_callback = lambda do |request|

[ 503, {}, ["Server Error\n"]]

end

See further documentation on implementation here: <https://github.com/kickstarter/rack-attack>

Fail2Ban can also be used on the server level to provide similar protection.

## Incorrect Attempts (Timing Attacks)

Comparing submitted passwords against those in the password database using string evaluation opens the authentication system up to timing attacks. These attacks submit many passwords with different charaters and then measure the time for a response to be received. When comparing passwords using string evaluation, the password is iterated through comparing each charater of both strings and returning false if they are not identical, otherwise moving onto the next charater for comparision. Therefore, passwords which have the correct starting charaters take longer to compare than those which do not, allowing an attacker to try many passwords with different charaters and calculate which are the the correct charaters.

To prevent this kind of attack passwords should not be compared, instead they should be hashed with a salt which the attacker will not have access to. Since any change in the submitted password will vastly change the hash with salt, it will make timing attacks against the authentication system useless. Bcrypt and other similar authentication systems implement this type of protection and can be included in almost all languages.

## Fake Requests (DoS)

Denial of Service (DoS) is a attack method used to stop genuine users from access a site, potentially reducing site revenue. As mentioned above, request throttling can be used to limit the effect of this issue, however since DoS attacks are commonly performed by unintelligent bots, Turing tests can also be used to prevent access to pages or login attempts.

Turing tests, such as reCaptcha, by definitiaon are used to distingish between bots and humans, preventing access if they are failed.

Implementing Google reCaptcha v2 (tick box reCaptcha) in Ruby on Rails v6 without using gem:

Sign site up to google reCaptcha Site: <https://www.google.com/recaptcha/admin/create> to get site standard and secret key. Then add keys to credentials using ‘EDITOR=vim rails credentials:edit’

To allow credentials to be used in javascript packs which are compiled with webpacker, add the rails webpacker erb loader to the project using 'rails webpacker:install:erb' on the commandline.

Create javascript erb pack for loading the reCaptcha code on page load in the relevant location (app/javascript/packs/recaptcha/recaptcha.js.erb):

document.addEventListener('turbolinks:load', function(){

var reCaptchaScript = document.createElement('script')

window.reCaptchaOnload = function() {

grecaptcha.render('recaptcha', {'sitekey' : '<%= Rails.application.credentials.recaptcha[:site\_key] %>'})

}

reCaptchaScript.src = 'https://www.google.com/recaptcha/api.js?onload=reCaptchaOnload&render=explic it'

document.head.appendChild(reCaptchaScript)

})

Then add the reCaptcha and javascript pack tag div into the relevant html file :

<div id='recaptcha'></div>

<% unless ENV['RAILS\_ENV'] == 'TEST' %>

<%= javascript\_pack\_tag 'recaptcha/recaptcha' %>

<% end %>

When a user sends a login attempt, it will now also send the parameter 'g-recaptcha-response'.

Next add a method in the controller to verify the captcha and return a boolean if google deems the captcha response valid and use this to allow or disallow user action:

def recaptcha\_confirmation(recaptcha\_response)

response = Faraday.post('https://www.google.com/recaptcha/api/siteverify') do |request|

request.params['secret'] = Rails.application.credentials.recaptcha[:site\_secret]

request.params['response'] = recaptcha\_response

end

JSON.parse(response.body)['success'] == true

end

## SQL Injection