Rails - Security

# Overview

Locking down Ruby on Rail applications.

# SQL Injection

## The Vunerability

SQL injection is most common type of attack on servers, with a sucessful attack potentially giving an attacker unlimited access to the server database. The attack occurs when mailcious code is submitted in a field for execution and the server does not correctly filter inputs for string literal escape charaters embedded in SQL commands. Therefore, to prevent SQL injection, all input parameters which a user can submit to the server must be filtered and whitelisted for string literal escape charaters and mailcious code.

## ActiveRecord

Since Rails uses the ORM Active Record to manipulate database records, many methods are already parameterised and much less vunerable to SQL injection. However, there are still some methods which allow for raw SQL to be written, or simply insert arguments passed into SQL commands

## Commands

Never use untrusted inputs to create SQL parameters such as column names, table names, or pass inputs in ActiveRecord methods which accept column names, table names, or raw SQL.

## Strings As Arguments

Since some ActiveRecord methods take passed arugments and insert them into SQL as fragments, user inputs passed directly into methods leave the system vunerable to SQL injection. Therefore, strings should only be passed as arugments to ActiveRecord methods if they are the only type option for argument input (i.e. the argument could have been a integer) and they have be sanitised for SQL injection using some whitelisitng method.

A general rule of thumb is, if you have to add surrounding quotes to an input, it will generally be vunerable to SQL attacks if the input is not sanitised. To avoid enclosing inputs in string, parameterising can be performed instead of string interpolation:

User.where("email LIKE '%#{partial\_email}%'") # Unsafe

User.where("email LIKE ?", "%#{partial\_email}%") # Safe

For methods such as #where, inputs should not be added as pure strings, instead they should be added as a hash, or array to fix the columns being searched:

User.where("name = '#{params[:name]'") # SQL Injection!

User.where({ name: params[:name] }) # Fixed columns

Defining the query parameters by name, not only fixes the search column, it also takes advantage of the two ActiveRecord::Sanitization::ClassMethods:

* sanitize\_sql\_for\_conditions - sanitizes parameters into a valid SQL fragment for a WHERE clause
* sanitize\_sql\_for\_assignment - sanitizes parameters into a valid SQL fragment for a SET clause

which are included on all ActiveRecord::Base models by default, accepting hash, array, or string parameters.

## Strong Parameters

The strong parameters plugin in ActionController prevents parameters being used in ActionModel mass assignments until they have been whitelisted. Therefore, any updates to columns which have not been permitted, such as those from SQL injection attacks or coding mistakes will raise exception, preventing exposing vunerable data.

For example, if a set of parameters is required from the params hash, it must then be permitted or it will raise an ActiveModel::ForbiddenAttributes exception:

def create

Person.create(params[:person]) # Exception raised

end

def update

person = current\_account.people.find(params[:id]) # No exception since no require

person.update\_attributes!(person\_params) # No exception since permitted

redirect\_to person

end

private

def person\_params

params.require(:person).permit(:name, :age)

end

Outside of controllers, such as with API calls, data can be split and parameterised for require and permitting, limiting the risk of accidental mass assignment:

raw\_params = { :email => "john@example.com", :name => "John", :admin => true }

parameters = ActionController::Parameters.new(raw\_params)

user = User.create(parameters.permit(:name, :email))