Rails Security - First part

Author: Boik
Email: boik@tdohacker.org

0x00 Introduction

Rails is a software library that extends the Ruby programming language. It's often promoted as an MVC web framework, which stands for Model, View, and Controller respectively.

De facto, Web developers are attracted by its simplicity and the priciple of Convention over Configuration, and it has become more popular in recent days.

Despite Rails is a mature framework being used today, Web Security issues are still there. Therefore, this paper will briefly address and give introduction to those discovered vulnerabilities of Rails.

0x01 Mass assignment

- The toxic feature we are deeply in love
- We can pass a Hash object to assign multiple attributes at once
- If we don't limit to what attributes can be assigned through a Hash object, some attributes will be modified unexpectedly

```ruby
def create
  # 假設表單的input欄位送出params[:user]，參數如下
  # {:name => "Lobsiinvok", :email => "xxx@gmail.com", :isadmin => 1}
  @user = User.build(params[:user])
  @user.save
end

def update
  @user = User.update(params[:user])
end
```


- Public Key Security Vulnerability and Mitigation
- After Rails 3.2.3, `config.active_record.whitelist_attributes = true`

- After Rails 4, another security enhancement `strong_parameters` has been added, which allows you to filter attributes easily in Controller layer.

0x02 Unsafe Query Generation

- It's possible for Rake to generate some unsafe queries when dealing with params

```ruby
unless params[:token].nil?
  user = User.find_by_token(params[:token])
  user.reset_password!
end
```

drops.wooyun.org

- We can bypass the check of `.nil?` through forging `params[:token]` to `[]`, `[nil]`, `[nil, nil, ...]` or `['foo', nil]` so as to insert `IS NULL or IN ('foo', NULL)` into SQL query, which might cause the application to behave unexpectedly.
- After Rails 3.2.8, Rails has added a method called `deep_munge` to eliminate `nil`s in Hash object

```ruby
+ test "perform_deep_munge" do
+   ActionDispatch::Request::Utils.perform_deep_munge = false
+   begin
+     assert_parses({"action" => nil}, "action")
+     assert_parses({"action" => {"foo" => nil}}, "action[foo]")
+     assert_parses({"action" => {"foo" => {"bar" => nil}}}, "action[foo][bar]")
+     assert_parses({"action" => {"foo" => {"bar" => [nil]}}}, "action[foo][bar][]")
+     assert_parses({"action" => {"foo" => [nil]}}, "action[foo][]")
+     assert_parses({"action" => {"foo" => [{"bar" => nil}]}}, "action[foo][][bar]")
+     assert_parses({"action" => ['1',nil]}, "action[]=1&action[]")
+   ensure
+     ActionDispatch::Request::Utils.perform_deep_munge = true
+   end
+ end
+
```

drops.wooyun.org

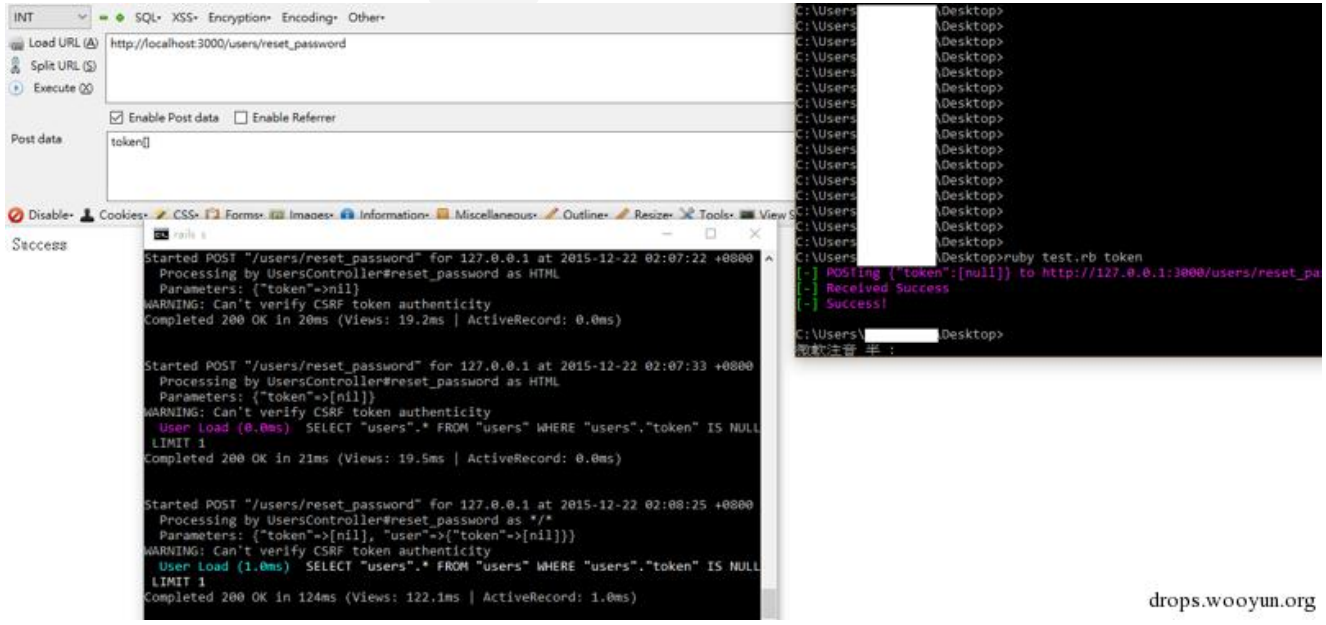
- A small PoC:

- Say we have this code snippet

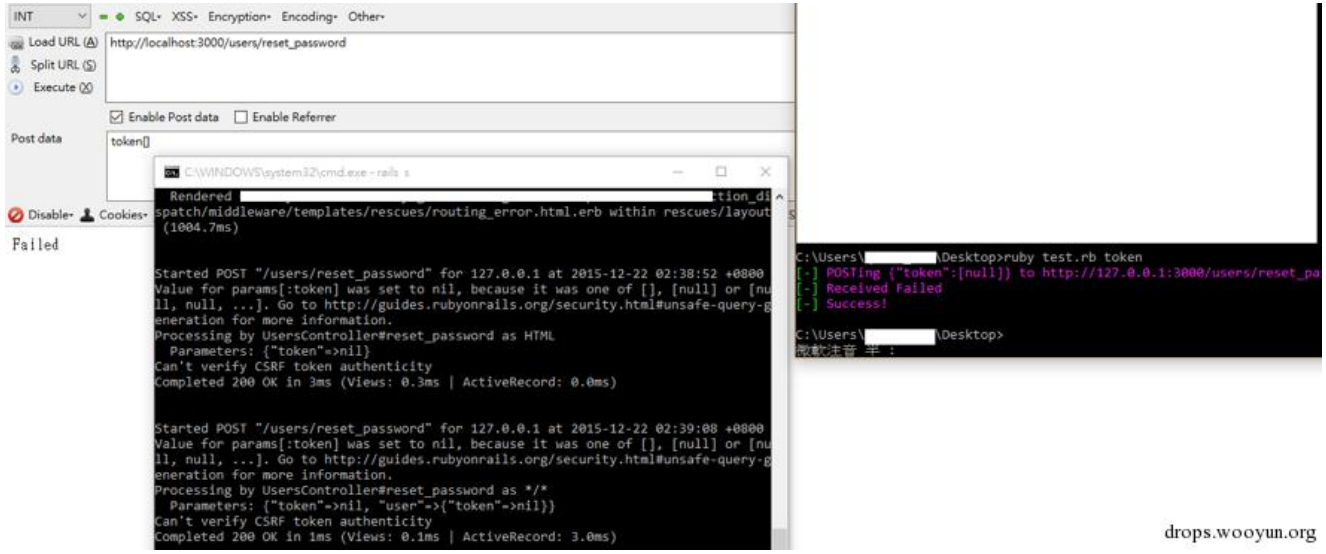
```ruby
class UsersController < ApplicationController

  def reset_password
    unless params[:token].nil?
      @user = User.find_by_token(params[:token])
      # @user.reset_password!
      render :json => 'Success'
      return
    end
    render :json => 'Failed'
  end
end
```


- We can bypass the check of `.nil?` in **Rails 3.1.0**



- However, the attack has been mitigated in **Rails 4.2.5**

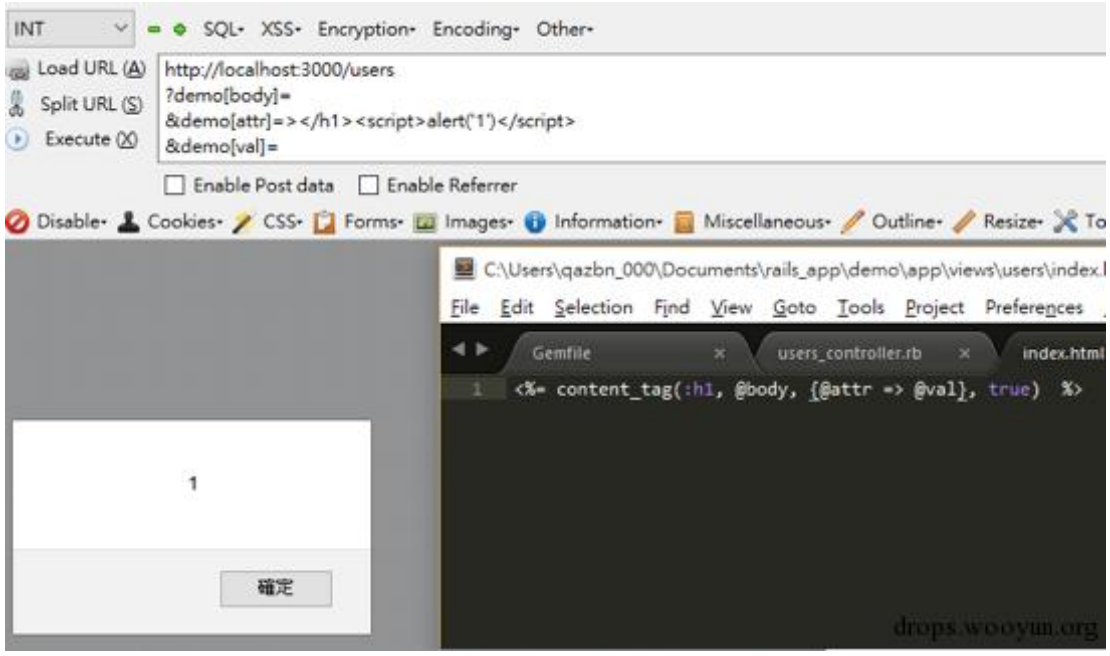


0x03 Content_tag

- `Content_tag` is a helper for developers to generate HTML elements more quickly
- It can also generate some unsafe HTML sometimes (ref: brakeman)

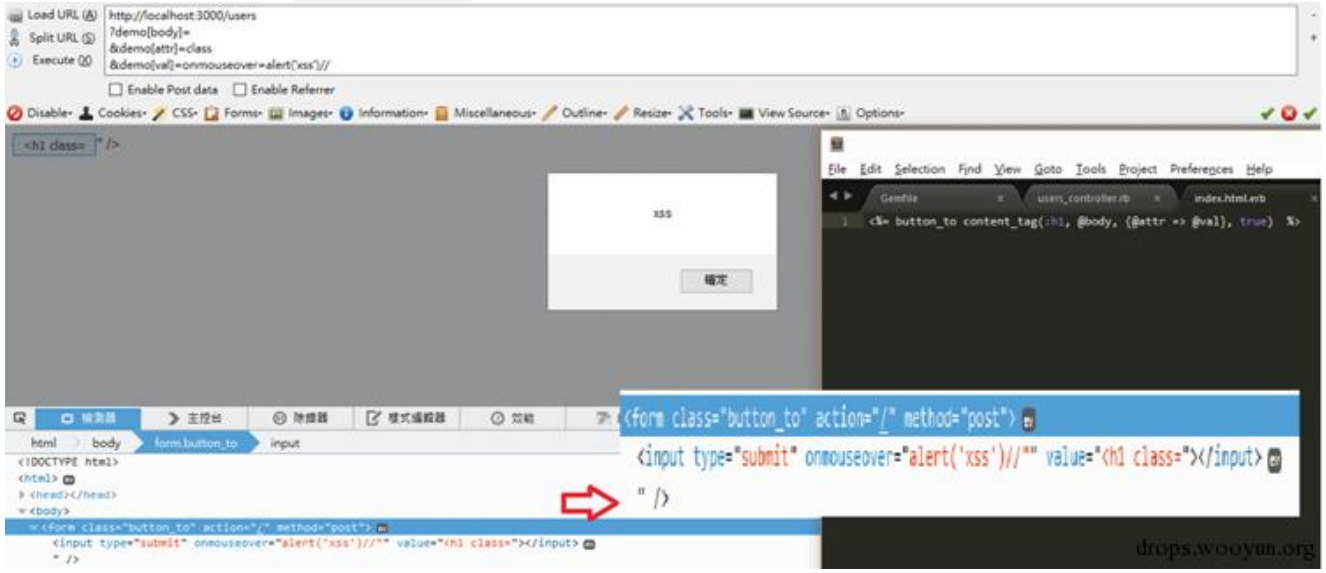
```
#Checks for unescaped values in `content_tag`
#
#    content_tag :tag, body
#                      ^-- Unescaped in Rails 2.x
#
#    content_tag, :tag, body, attribute => value
#                               ^-- Unescaped in all versions
#
#    content_tag, :tag, body, attribute => value
#                                          ^
#                                          |
#           Escaped by default, can be explicitly escaped
#           or not by passing in (true|false) as
```

- In rails 4.2.5, attributes still can be injected with any HTML data



- Though the values of attributes get escaped, they are still subject to XSS attack sometimes

- For instance, when `button_to` gets involved



- Why?
  - `Content_tag` will return strings with `html_safe` attribute, and `button_to` won't escape those strings since it considers them `html_safe`

0x04 YAML.load

- CVE-2013-0156
  - Remote Code Execution Vulnerability
  - Due to the support of parsing yaml in XML parser, we can craft a special XML payload to instantiate a remote object, which in turn can be used to execute any ruby code remotely in the context of the application
  - After Rails 3, the parsing of nodes with yaml type have been disallowed by default

```
DISALLOWED_TYPES = %w(symbol yaml)
```

```
def initialize(xml, disallowed_types = nil)
  @xml = normalize_keys(XmlMini.parse(xml))
  @disallowed_types = disallowed_types || DISALLOWED_TYPES
end
```

- CVE-2013-0333
  - Remote Code Execution Vulnerability
  - Before Rails 3.0.19, the default decoder used by default JSON parser is YAML
  - Details: http://ronin-ruby.github.io/blog/2013/01/28/new-rails-poc.html

0x05 Dynamic Render Paths

- When a call to render uses a dynamically generated path, template name, file name, or action, there is the possibility that a user can access templates that should be restricted (ref: brakeman)
- Before Rails 5, files without a template handler in their extension will be rended using the ERB handler, which might cause remote code execution
- Rails 5 has changed the default template handler from `ERB` to `Raw` (ref: commit)
- Details: http://devco.re/blog/2015/07/24/the-vulnerability-of-dynamic-render-paths-in-rails/

0x06 Reference

- The Ruby/GitHub hack: translated
- How Does Rack Parse Query Params? With Parse_nested_query
- Cross Site Scripting (Content Tag)
- Bad coding style can lead to XSS in Ruby on Rails

- 分析下难得一见的ROR的RCE（CVE－2013－0156）
- Rails PoC exploit for CVE-2013-0333
- Dynamic Render Paths
- Rails 動態樣板路徑的風險