Topic on Ruby on Rails Security

Poly Pentest 2010 final project:
Te-Chun Chao
0416545

1.Abstracts

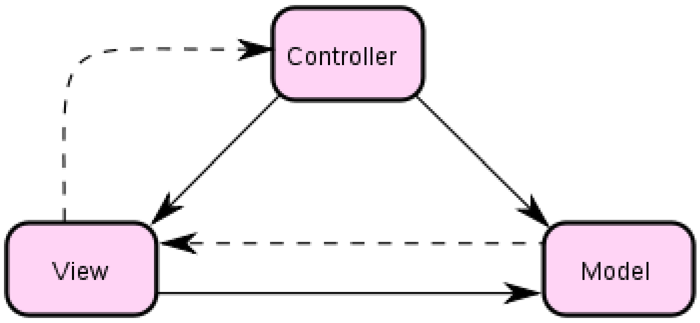
Ruby on Rails is an innovative web application framework based on Ruby. It was created by DHH(David Heinemeier Hansson) since 2004. It have a large amounts of users who build modern and highly demanded web application on it, like Twitter, Github, Groupon. Because it's widely used and have a thrive community sharing experience and improving it. How the Ruby on Rails framework handle web attacks is a interesting topic that this project will like to explore.

2.Source Structure

Source: https://github.com/rails/rails

Ruby on Rails is Open Sourced under MIT license(which means "use it whatever you want if you won't sue me") on Github. This project is using the 3.1.0beta.

MVC: Ruby on Rails is a MVC framework



see http://en.wikipedia.org/wiki/Model-View-Controller

Projects:

The source code of Ruby on Rails include 6 projects and railties(glue the project together) and gem(package installer for dependency), continuous integration config and rake(ruby's make) file.

The projects include:

ActionPack	Main project of Ruby on Rails, It provides Controller and Views Layer, Handling and responding request and the routing mechanisms.
ActionModel	The Model layer of Rails, define object that can interact with action

	pack.
ActionResource	Provide RESTful web service, can interact with other web service or implement service.
ActionSupport	The common utility classes and library for Rails.
ActionMailer	The email service of Rails.
ActionRecord	The Object Relation Mapping of Rails, adapt different database's data to Rails Model.

Some ruby magic for understanding code:

?: at the end of method means the method return boolean

x ||= y : means "x = x || y ",or "if(x==null) x=y; "

:name , use ":" before name means symbol, it is like a string but only stand for key or function name.

3. Security Policy of Rails:

Rails provides their security policy on http://rubyonrails.org/security.

For reporting a bug, you need to send e-mail to security@rubyonrails.org , the email will deliver to a subset of the core team who handle security issues, and will be acknowledge within 24 hours, and respond how they will handle in 48 hours.

Rails also have disclosure policy to protect the information until the fixed is ready, and send the security announcement and patches by mailing list.
http://groups.google.com/group/rubyonrails-security

User can also update the framework using gem or from Github.

4. Web Attack and how Ruby on Rails handle:

0. How to fingerprint application using Rails?

First step of a hacker is to recon what language and framework the server is running, Ruby on Rails have some obvious fingerprint to identify.

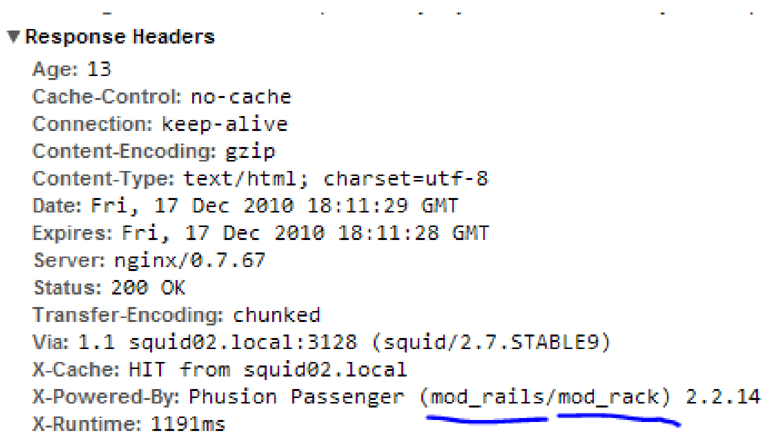
Folder Name:

The Rails application structure is pre-generate by rails, which is one of the Rails philosophy "Convention over Configuration". However, some the folder name is rarely used by the others, so we can know it's possible this website is using Rails by see these naming in source: javascripts/, stylesheet/

Server Header:

By seeing the response header, we can see some rail module is used, like mod_rails/mod_rack. As we know that the server is running Rails.

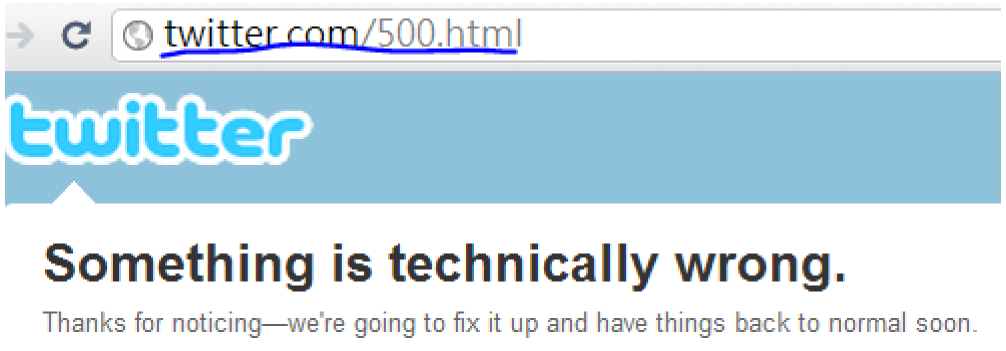
ex: www.scribd.com



```
▼Response Headers
   Age: 13
   Cache-Control: no-cache
   Connection: keep-alive
   Content-Encoding: gzip
   Content-Type: text/html; charset=utf-8
   Date: Fri, 17 Dec 2010 18:11:29 GMT
   Expires: Fri, 17 Dec 2010 18:11:28 GMT
   Server: nginx/0.7.67
   Status: 200 OK
   Transfer-Encoding: chunked
   Via: 1.1 squid02.local:3128 (squid/2.7.STABLE9)
   X-Cache: HIT from squid02.local
   X-Powered-By: Phusion Passenger (mod_rails/mod_rack) 2.2.14
   X-Runtime: 1191ms
```

Error Page:

By default, the error page of Rails is named as [responseCode].html, like 500.html, so we can enter the name to see if we can find the error page.



twitter.com/500.html

Something is technically wrong.

Thanks for noticing—we're going to fix it up and have things back to normal soon.

1.XSS

The Cross Site Script attack is a huge threat of websites, In Rails 2, you can call the "h" which is alias of html_escape to protect your output:

```
<%= h @output %>
```

However, Rails 3 have been setting the HTML encoding by default, so we don't need to remember escaping output.

The view will be handle by the appropriate view handler and render to Html. During the rendering, string data will be parsed out special character and encoded to the escaped HTML character.

In ERB:Util Module, we can see how Rails parse the string:

```
In ActiveSupport\lib\active_support\core_ext\string\output_safety.rb
6:   HTML_ESCAPE = { '&' => '&amp;',   '>' => '&gt;',    '<' => '&lt;', '"' => '&quot;' }
7:   JSON_ESCAPE = { '&' => '\u0026', '>' => '\u003E', '<' => '\u003C' }
```

```
...
def html_escape(s)
   s = s.to_s
   if s.html_safe?
      s
   else
      s.gsub(/[&"><]/) { |special| HTML_ESCAPE[special] }.html_safe
   end
end
```

From the source, we can know that Rails escape the special character to HTML escaped character. Another interesting part is that every string in Rails have html_safe? method, which will return boolean that the string is html safe or not. This is a extension defined in ActionSupport, In script language, you can easily extend the basic type when it's needed. This expansion provide every string to be escaped before output.

On the other side, if we want to allow HTML tag in output, we can call "raw" method in html helper to output raw string.

```
<%= raw @html_content %>
```

but these string might also infected by XSS string, so we should call "sanitize" method before output Html string.

```
<%= sanitize @html_content %>
```

sanitize is can strip out the dangerous tag, it can be set as white-list or black-list based, default tag is showed in the source code:

```
In actionpack\lib\action_controller\vendor\html-scanner\html\sanitizer.rb
   # Specifies the default Set of tags that the #sanitize helper will allow unscathed.
   self.allowed_tags = Set.new(%w(strong em b i p code pre tt samp kbd var sub
   sup dfn cite big small address hr br div span h1 h2 h3 h4 h5 h6 ul ol li dl dt dd abbr
   acronym a img blockquote del ins))
```

2.CSRF

CSRF is another threat that a website might have, in Rails, it provide a build-in CSRF protection function. We can activate it by calling "protect_from_forgery" in controller:

```
class ApplicationController < ActionController::Base
      protect_from_forgery
```

```
end
```

Every form in the controller will be added a validation token , this token is stored as a random string in the session, which an attacker does not have access. When a request reaches the application, Rails then verifies the received token with the token in the session. The layout also need to including "csrf_meta_tags" in html head

```
In a sample application's form:
<meta name="csrf-param" content="authenticity_token"/>
<meta name="csrf-token"
content="b3wvDdaaA9t0W0jgdP0nW0T12jhCohYJ67oBTQ/P6K0="/></head>
...
<form accept-charset="UTF-8" action="/en/orders/298486374" class="edit_order"
id="edit_order_298486374" method="post">
    <div style="margin:0;padding:0;display:inline">
        <input name="utf8" type="hidden" value="&#x2713;" />
        <input name="_method" type="hidden" value="put" />
        <input name="authenticity_token" type="hidden"
             value="b3wvDdaaA9t0W0jgdP0nW0T12jhCohYJ67oBTQ/P6K0="/>
    </div>
......
</form>
```

the token is generated and saved in session:

```
In actionpack\lib\action_controller\metal:
  def verified_request?
    !protect_against_forgery? || request.forgery_whitelisted? ||
      form_authenticity_token == params[request_forgery_protection_token]
  end

  # Sets the token value for the current session.
  def form_authenticity_token
    session[:_csrf_token] ||= ActiveSupport::SecureRandom.base64(32)
  end
```

3.Session

Http is a stateless protocol, so server is using session id in cookie to distinguish different connections. When a user connection to server, it will assign a session id to the user, than when the user connect again, server will load the exist session for the user. And therefore, attacker might sniff the session id to duplicate user authorization.

Rails session :

In Rails, session id is a random Hex number string:

```
In /actionpack/lib/action_dispatch/middleware/session/abstract_store.rb
def generate_sid
    ActiveSupport::SecureRandom.hex(16)
end
```

So it is not feasible to brute-force Rails' session ids.

Session Values :

By default, ruby save session values in cookie, because get value from request header is much faster and require less memory than getting it from hash. But It also introduce security problem that some important session value, such as user_id, also saved in the client with plain code.

To prevent client modified session values in cookie, Rails encrypt the cookie with SHA1, HMAC and encode to 64base with a secret key:

save the cookie encryption key in /config/initializer.rb. encoded it with HMAC method and AES and encode to 64 base.

```
In activesupport/lib/active_support/message_encryptor:
def initialize(secret, cipher = 'aes-256-cbc')
  @secret = secret
  @cipher = cipher
end

def encrypt(value)
  cipher = new_cipher
  # Rely on OpenSSL for the initialization vector
  iv = cipher.random_iv

  cipher.encrypt
  cipher.key = @secret
  cipher.iv    = iv

```

```
    encrypted_data = cipher.update(Marshal.dump(value))
    encrypted_data << cipher.final

    [encrypted_data, iv].map {|v|
ActiveSupport::Base64.encode64s(v)}.join("--")
  end
```

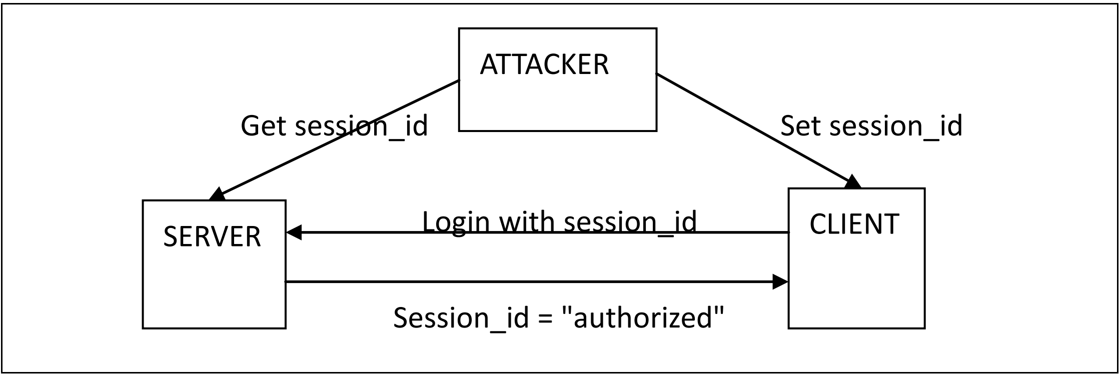
and the secure key is saved in Rails application, in config/initializer/secret_token.rb:

```
Rails.application.config.secret_token =
'21d0fd931f5c004bf7c6c1dbadeb0271123557a574f21d744399d91c6a47c1ee14eeab
d5eddd2d5ef8a07d4cf2b705fd00b48d6f4a3ae22c9b86b0822b563e7a'
```

and the length is restricted to 30 Char length. So it is hard to brute force decrypt. But it could also be dangerous if attacker get the encryption key under your application folder. So for the security, do not save important data in session values.

Session fixation:

Session fixation is a attack method by fixing user's session id to a known id, and thereby the attacker can assess the server as user using the same session Id.



First, the attacker get a connection and maintain a session id. And than assign the session id to other user by JavaScript or query string.

For example, a Session fixation XSS might like this:

```
<script>document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9"; </
script>
```

after user login, attacker can also login by using the same session_id because the authorize information is saved in the session.

Rails with session fixation:

The solution to keep session save, is remember to call reset session after user login or set the expiration time on session.

```
In Sample App, LoginController:
  def login
    reset_session
    ...
  end
```

By default, Rails saves session data in cookie, thereby the authorize data is saved in the client side, therefore even if the attacker fixed the session id, they still can't access the data. But in the counterpart, the attacker can also get the authorize data and try to decrypt and fake it.

The storage setting is in Rails application: config/initializer/session_store.rb

```
SampleApp::Application.config.session_store :cookie_store, :key => '_ sampleApp
_session'
```

4. SQL injection:

In Rails, all database access is going through ActiveRecords, which is the Rails's ORM model. The ORM code is automatically generated by Rails according to the database schema and mapping the database Table. So Most of the database interaction is pre-defined in the ActiveRecords. For example:

```
# "SELECT id,name FROM User where id=?" is mapped in ActiveRecord by:
@user = User.find(param[:id])
```

So SQL injection is not a big problem since it's hard to inject SQL code.

However, there are still some condition that might use the SQL code like search:

```
User.find(:first, :condition => "login = '#{params[:name]}' AND password =
'#{params[:password]}'")
```

and if user enter username as "user ' OR 1=1" and password as "password = '' OR '1' = '1'". the result SQL will become:

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1' LIMIT 1
```

And it will return the first record in the database and get access to that user.

But Rails also has a built-in filter for special SQL character like quote, comment and line break. And it will apply to the method like find , find_by_parameter. but need to applied manually in :condition block:

```
User.find(:first, :conditions =>[ "login = ? AND password = ?", username,password ])
```

5.Mass assignment

The mass-assignment is a specific problem in Ruby on Rails, it allows an attacker to set any model's attribute by manipulating the hash passed to a model's value:
In the Html form, the parameter is like:

```
<form...>
  <label for="user_name">Name</label>:
  <input id="user_name" name="user[name]" size="40" type="text" value="dave" />

  <label for="user_password">Password</label>:
  <input id="user_password" name="user[password]" size="40" type="password" />
</form>
```

And the ActiveRecord will automatically mapping the attribute to database model

```
def signup
   @user = User.new(params[:user])
End
```

Therefore by setting other attributes, attacker can manipulate the value in database:

```
HTTP 1.1 POST/
User[name]="user"&user[password]="password"&user[admin]='true';
```

The counterpart of this problem, is to set the restricted list:

```
In model "User":
   Attr_protected :admin
```

or we can also set the white list:

```
In model "User":
   Attr_accessible :name, :password
```

6.Javascript Hijacking:

JavaScript hijacking is a problem that when the server use ajax to send request to server, if the page is XDD vulnerability, attacker can set script to overwrite the Native Object like "String" , steal or change the data between client and server

```
//With the script of:
<script src="http://my.bank.evil/transactions.json"></script>

//And JSON response:
[[account:"richman",amount:1000000,to:"someone"]]
```

The JSON respond might send back to the evil site when loading the script.

The solution of this problem is:

First, make sure the page is XSS protected, because XSS is the root of all evil.

Second, make sure the connection code is writting in closure, and pass the critical object to prevent overwrite

```
(Function(var window){
        //...some action
        // var str = new String();
})(window)
```

Or we can also add some garbage data between return value to disable the hijack function:

```
===Some garbage padding ====
[[account:"richman",amount:1000000,to:"someone"]]
```

5.Conclusion

By default, Rails has many security features enabled:

- SQL quoting
- HTML escape
- CSRF protection

The config can do the adjustments, but require the knowledge to get things right.

With those features enabled, Rails can help us to take care some basic security requirement, make our application more safety. However, it's not a overall projection, developer need to stay aware in the vulnerabilities.

6.Reference

Ruby on Rails Security: http://vimeo.com/1323092
Ruby on Rails Security Guide: http://guides.rubyonrails.org/security.html
Agile Web Development with Rails, 3rd Edition.
Ruby on Rails Security Best Practice: http://ihower.tw/blog/archives/4096