Some Best Practices in Rails Security

Josh Ma, 3/21/2013
How do you break into a system?

- System authenticates and authorizes users
- Break authentication
  - Pretend to be another user
  - Usually privileged user or a victim
- Break authorization
  - Skip the checks!
  - Find a bug
  - Find an oversight (e.g. leftover Rails scaffolding)
Breaking Authentication

- Trick system into thinking you're someone else
- Trick someone else into letting you be them!
Sessions

- Dependency on client-side data (cookies) opens up huge hole for vulnerabilities
- Out of your control! (Sort of.)
- Hard to tell between real user with cookie C and a malicious user with cookie C
  - IPs, locations can change
- Tough if we can't trust the cookie
  - Avoid storing sensitive information if possible
  - Won't always be possible (sessions)
Session Hijacking

- **Steal a user's cookie**
  - Can "easily" sniff on the same WiFi network
  - **Use SSL (https):** config.force_ssl = true
  - (Don't worry for 6.170)

- **Attackers modify their own cookie**
  - is_admin = true;
  - **Rails uses checksum to prevent tampering**
  - "cookie_value=x; checksum=H(x,secret)"
Session Hijacking (cont)

- Playback
  - cookie: bank_amount=100; hash=a3bc
  - withdraw(50);
  - cookie: bank_amount=50; hash=bc8d
  - playback old cookie:
    - bank_amount=100; hash=a3bc
  - Not protected by checksum
  - Don't store sensitive information in cookie!
  - Rails: use SessionStore instead of CookieStore
Securing (Against) Cookies

- SSL to prevent sniffing
- Avoid storing sensitive information in general (not worth the risk)
- In general, ask for password before highly sensitive operations
- If practical, expire sessions earlier (hacker can only impersonate for session duration)
  - config/initializers/session_store.rb:
    ```ruby
    Myapp::Application.config.session_store = 
    cookie_store,
    :key => "_myapp_session",
    :expire_after => 10.minutes
    ```
CSRF

<img src="http://www.example.com/blog/1/destroy">

- GET request sent to URL even if on a different domain
- Cross Site Request Forgery dangerous because client's cookies are sent along with request
- Rails requires authenticity token = H(sessionid|serversecret) with non-GET requests (make sure GET requests are safe!)
<%= form_tag %>

```html
<form accept-charset="UTF-8" action="/polls" class="new_poll" id="new_poll" method="post">
  <div style="margin:0;padding:0;display:inline">
    <input name="utf8" type="hidden" value="✓" />
    <input name="authenticity_token" type="hidden" value="DGuJCBlonkLoWiRJu5hAfF4LbvecGEDTPLsn2WfcPT4=" />
  </div>
  ...
</form>
```
CSRF (cont)

- Token in AJAX can be clumsy
- Need to set up all requests to include it in header:

```ruby
<%= javascript_tag
"var AUTH_TOKEN = #{form_authenticity_token.inspect};"
if protect_against_forgery?
%

... 

$(document).ajaxSend(function(e, xhr, settings) {
    xhr.setRequestHeader("X-CSRF-Token", AUTH_TOKEN);
});
```
What happens without the token?

- Rails still allows you to make the request
- But session data is no longer available from controller

```ruby
session == nil
```
SQL Injection

- As covered in class, SQL queries with user input can be dangerous
- Use model methods:
  - Model.find(id)
  - Model.find_by_field(field_value)
- Use SQL fragments:
  - Model.where("login = ? AND password = ?", user_input_name, user_input_password)
- Even better, use hashes:
  - Model.where(:login => user_input_name, :password => user_input_password)
XSS Injection

- Rails HTML-escapes data by default!
  ```ruby
  <%= user.html_profile %>
  <script>alert("BAD");</script>
  ```
- Be careful about including raw user input
  ```ruby
  <%= raw user.html_profile %>
  <script>alert("BAD");</script>
  ```
XSS Injection

- Rogue scripts can do all sorts of harm
  - Read `$('#routing_number').html();`
  - `location.href = "http://www.paypal.ph.com/signin";`

- To defend against reading cookies, set "httpOnly" field on cookies
  - Doesn't allow scripts access
  - Rails already does this for session cookies!
File Serving

- Rail's file upload support is pretty basic
- Exact settings depends on the plugin you use
- Upload best practices
  - Limit file upload size
  - Don't trust user filenames, use a sanitizer
  - **Whitelist** allowed extension names
  - Upload to outside application directory
- Download best practices
  - **Absolutely escape filename**
  - `File.open('/var/www/uploads' + params[:pathname])` => `params[:pathname] = ' ../../../etc/passwd`
- Easiest option: use a service like Filepicker.
Mass Assignment

- A Rails Problem (TM)
  user = User.new(params[:user])
- "My form only has a name and password field!"

http://mysite.com/signup?user[admin]=1
params[:user] # => {:admin => true}
user.admin # => true
Mass Assignment (cont)

Better:
attr_protected: admin

Best:
attr_accessible :name

@user = User.new(params[:user])
@user.admin = true
@user.save
How do I keep my app secure?

- Follow best practices (don't reinvent the wheel, cleverness is clever until it's broken)
- Minimize attack surface (whitelist vs blacklist)
- Keep your Rails and gem versions up to date!
  - 12, 10, and 7 major vulnerabilities in last three years
  - Once vulnerability is in the wild, will be surge of hackers who take advantage of it
What should I look out for in 6.170?

- We want to see best practices and consideration of "obvious" attacks
- Don't disable CSRF because it's inconvenient
- Check for authorization for sensitive actions
- Strip / clean any user-input HTML
- Avoid SQL injections, use ORM methods
- Don't store sensitive data in cookies
- Just because there's no link to /show_all_passwords doesn't mean nobody will find it! Never trust security through obscurity
Ruby on Rails Security Guide

http://guides.rubyonrails.org/security.html#executable-code-in-file-uploads

(Most of this presentation based on the guide.)