Ruby On Rails Security

Heiko Webers
42@rorsecurity.info

Heiko Webers

- Ruby On Rails Security Project: www.RoRsecurity.info
  - E-Book „Ruby On Rails Security“
  - Ruby On Rails Security Audits
- Software company

Attack trends

- Cracker‘s motivation: Make money, it’s a multi-billion dollar business
- Recently: Attacks on trusted news or sports sites
  - Break into the server: Apache, cPanel, CMS holes
  - Advertisement with malware
  - Inject specific exploits or entire attack frameworks: MPack, 500-1,000$, available in the Russian underground, guaranteed 40-50% success rate

Injection

- The wrong way
  - Directly use external input
- The right way
  - Consider external input malicious, until proven otherwise
  - Sanitize or (preferably) escape it according to the context it‘s being used in

Injection - Contexts

- SQL – remember SQLi in `find_by_sql`
- SGML (HTML, XML, RSS…) – Escape against XSS, SafeErb plug-in
- JavaScript – Escape possible code in a RJS context

```
escape_javascript()
```

Injection - Contexts

- CSS – This is how the Samy worm brought down MySpace

```
<div id=mycode style="BACKGROUND: url('javascript:
eval(document.all.mycode.expr)')" expr="..." />
```

- Command line parameter: No | or ; allowed, use `system()` instead of `` `command` ``

```
`ls #{dir}` # dir #= "whatever | rm *"
```

Injection - Contexts

- Textile – Definitely sanitize the result with Rails' `sanitize()`

```
!bunny.gif(Bunny" onclick="alert(1))!
```

```
<p><img src="bunny.gif" title="Bunny"
onclick="alert(1)" alt="Bunny"
onclick="alert(1)" /></p>
```

Whitelists vs. Blacklists

- Use `before_filter :only` instead of `:except`
- `Attr_accessible` instead of `attr_protected`
- Against XSS: Allow `<b>` instead of removing `<script>`
- Don't try to "correct" user input by blacklists:
  - `"<sc<script>ript>".gsub("<script>", "")`
  - But reject malformed input

Cross-Site Reference Forgery (CSRF)

- Browser sends domain cookie for every request to that domain
- Client is logged in, authentication information in the cookie
- Victim clicks a link or views a page with a special image:

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

CSRF in Rails

- The wrong way

```
GET /project/1/destroy
```

- The right way

```
GET /project/1/show
POST /project/1/destroy
verify :method => :post, :only => [:destroy]
```

CSRF & POST

```
<a href="http://www.harmless.com/" onclick="var f =
  document.createElement('form'); f.style.display =
  'none'; this.parentNode.appendChild(f); f.method =
  'POST'; f.action =
  'http://www.example.com/account/destroy';
  f.submit();return false;">To the survey</a>

<img src="http://www.harmless.com/img" width="400"
  height="400" onmouseover="..." />
```

CSRF in Rails

- The right way

```
protect_from_forgery :secret => "very_long_secret"
```

- Includes a security token in non-GET requests, automatically generated in form_for() and others

```
<input name="authenticity_token" type="hidden"
 value="3a1e11299eff1fa5cbc724ca32978448098af0" />
```

Administration Interface

- The wrong way
  - Vulnerable to XSS (steal a privileged cookie)
  - Vulnerable to CSRF
    ```
    <img src="/admin/user/destroy/1" />
    ```
  - The same cookies used for the application and the admin interface

Administration Interface

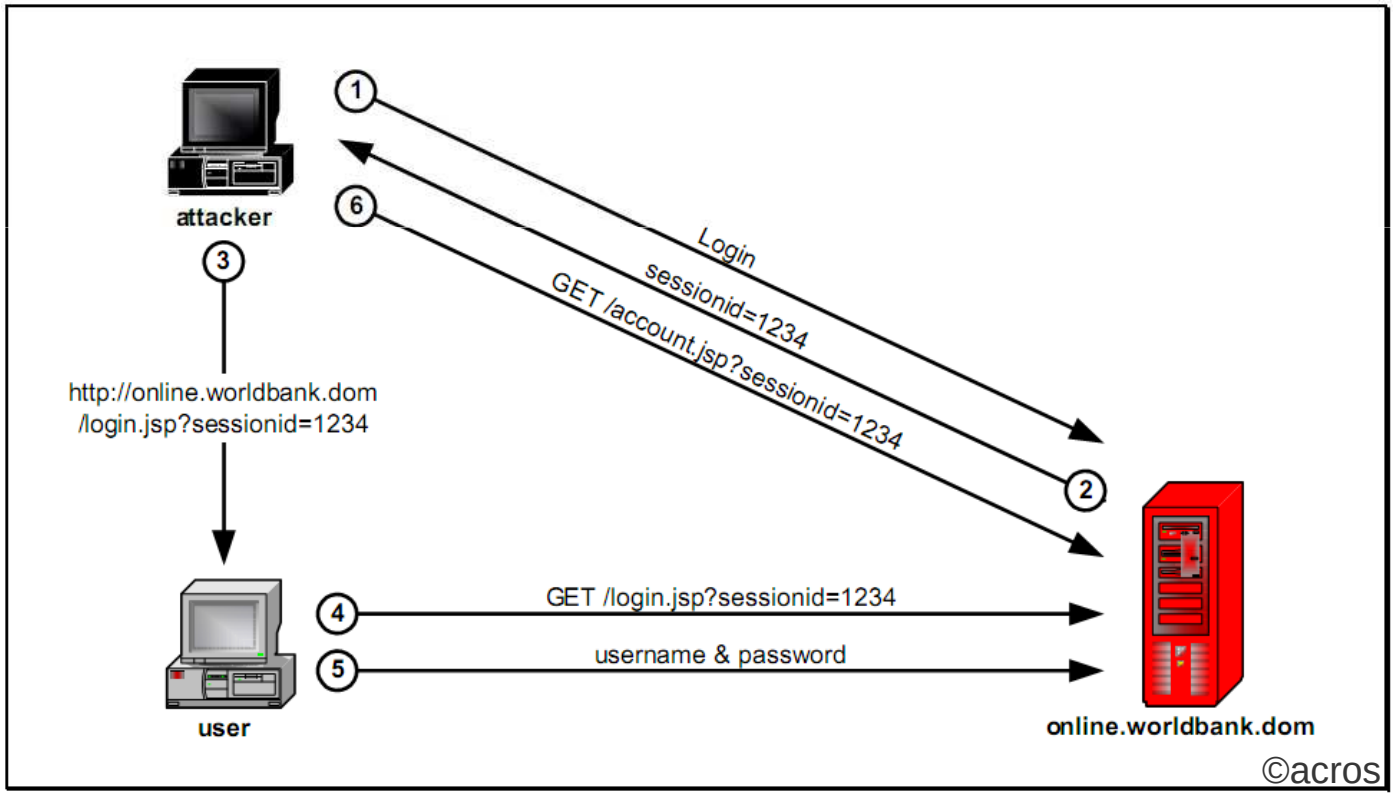
- The right way
  - Take security even more serious (especially XSS & CSRF)
  - Take precautions for the worst case: Someone else takes control of the administration interface
  - Require to log in to the interface despite of a valid session, might even be special admin login credentials
  - Introduce user roles: Different permissions for different admins

Administration Interface

- The right way
  - Put it to admin.example.com instead of www.example.com/ admin: A (stolen admin) cookie from www.example.com doesn‘t work on admin.example.com
  - Check the remote IP: Administration allowed from a certain IP (check `request.remote_ip`)

Session Fixation

- Instead of stealing a cookie, an attacker fixes a user‘s session identifier known to him



Session Fixation

- Change the victim‘s cookie, for example with HTML/JS:
  ```
  document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9";
  ```
- The wrong way
  - Vulnerable to XSS (the most obvious way to fixate sessions)
  - Allow users to log in with the same session ID for years

Session Fixation

- The right way
  - If the application is non-public: Turn off cookies for the public parts, so an attacker may not obtain a valid session ID
  - Issue a new session ID after a succesful login

```
class SessionsController < ApplicationController
  def create
    reset_session
    …
  end
end
```

Session Fixation

- The right way
  - Expire sessions, frequency based on how critical the application is
  - An attacker may write an automated script to keep a session alive: Check the session‘s created_at, as well (for ActiveRecordStore)

Login

- The wrong way
  - Not updating plug-ins (e.g. restful_authentication)
- The right way
  - Check for updates: There was a security hole in it, you could log in without login credentials

```
GET /activate?id=
      User.find_by_activation_code(params[:id])
SELECT * FROM users WHERE (users.`activation_code` IS NULL)
LIMIT 1
```

  - See http://www.rorsecurity.info/2007/10/28/restful_authentication-login-security/

User Management

- The wrong way
  - Make changing password and e-mail address easy
- The right way
  - Make it harder to seize an account
  - Require to enter the old password when changing
  - Or the answer to a security question

User Management

- The wrong way
  - Specific error messages enable username enumeration (for login and “send-forgotten-password” pages)
- The right way
  - Armed with a list of usernames and a dictionary for the passwords, a bot might brute-force accounts
  - First step: Possibly disable an account or require to enter a CAPTCHA after a certain amount of failed logins

CookieStore

- What you store in the session can be seen by the client

```
<base64 encoded session>-<digest>
```

- The wrong way
  - Store secrets, more than 4K of data, entire objects
  - Use a trivial secret

```
config.action_controller.session = {
  :secret      => 'trivial'
}
```

CookieStore

- The right way
  - Ok if you store a user_id and flash message only
  - Make the secret at least 30 characters long

```
config.action_controller.session = {
 :session_key => '_app_session',
 :secret      => '0x0dkfj3927dkc7djdh36rkckdfzsg'
}
```

CookieStore

- Be aware of replay attacks
  1. User receives credits, stored in his session
  2. User buys something
  3. User gets his new, lower credits stored in his session
  4. Cracker takes his saved cookie from step #1 and pastes it back in his browser's cookie. Now he's gotten his credits back
  - Normally solved using a nonce, but that's very hard for multiple app servers (mongrels)

Files

- The wrong way
  - `send_file '/var/www/uploads/' + params[:filename]`
  - `GET /download?filename=../../../etc/passwd`
  - `PUT /upload?filename=../../../etc/passwd`
- The right way
  - Store filenames in the DB, name the files after the record ID, just as the attachment_fu plug-in does
  - Verify the downloaded file to be in the correct directory

```
raise if DOWNLOAD_DIR =!
         File.dirname(filename)
```

Files

- The wrong way
  - Store file uploads in Rails‘ public directory
  - Upload: /public/uploads/file.fcgi
- The right way
  - Do not store it in Apache's DocumentRoot directory tree

Hate CAPTCHAS?

- Say Hello to negative CAPTCHAs
  - Don‘t ask the user to proof he‘s human, but reveal that the spam/login bot is a bot
  - Include a honeypot field that is hidden from the user by CSS
  - If this field contains any text, it must be a bot
  - Or make it more sophisticated

- http://nedbatchelder.com/text/stopbots.html

Thanks