
Intro
According to The Gartner Group, 97% percent of web applications are vulnerable to attack. This tutorial will help you be in the 3%. Fortunately for us, Rails includes several mechanisms that help with making your web applications secure. We will go over each in detail.

Fun fact: According to the Symantec Global Internet Security Threat Report, the underground prices for stolen bank login accounts range from $10–$1000 (depending on the available amount of funds), $0.40–$20 for credit card numbers, $1–$8 for online auction site accounts and $4–$30 for email passwords [1].

Sessions
By default, Rails stores all session information in a cookie. It is important that you know the implications of this. The cookie is signed but unencrypted. This means that anyone with the cookie can read its contents, but not modify them.

By the way, the signature is the SHA512 of the session data with a server secret. You should set the server secret in environment.rb like this:

```ruby
config.action_dispatch.session = {
  :key => '_app_session',
  :secret => '0x0dkfj3927dkc7djdh36rkckdfzsg...'
}
```

Make the key long and random (e.g. 50 random characters). This should not look like a typical user password, e.g. 1<3LadyGaga1991, because you don’t have to memorize it. If you open-source your application, remember not to put this secret in the repo!

If read access to session data is not something your users should have, then consider using ActiveRecord::SessionStore, which stores session data on the server rather than in a cookie.

Session Replay Attacks
Consider an application which awards credits to users, and the credits can be used to make purchases. Suppose the application stores the number of credits in the session (in a cookie). Is there anything wrong with this?

A user could accumulate some credits, then backup her cookie. After making a purchase, she can restore her cookie from the backup and now she has her credits back. This is called a
**replay attack.** One solution might be to include a nonce (random value) in the session, which can only be used once. The server has to manage these nonces in the database, and this defeats the whole purpose of storing the session in a cookie (avoiding database latency).

The best solution is simply to not store this kind of information (e.g. credits) in the session at all. Instead, store the user ID in the session, and look up the credits for that user ID in the database.

**Session Fixation Attacks**

A session fixation attack works like this. First, an attacker creates a session by visiting the web application. The attacker writes down the session ID stored in the cookie. Next, through a JavaScript injection, the attacker sets the victim’s session ID to the one he wrote down. This must happen before the victim logs in. The victim is now sharing a session with the attacker, so when the victim logs in, the attacker is automatically logged in (as the victim) also.

The solution to this kind of attack is relatively simple: give the user a new session after they log in. This can be accomplished with the `reset_session` method. Note that this will completely wipe the session, so you will need to copy over relevant information from the old session such as the user ID.

**Tips**

- Set the server secret for signing session cookies to a long, random string.
- Don’t store information in the session that makes session cookies vulnerable to replay attacks (e.g. number of credits).
- Give the user a new session when they log in.
- Make sessions expire after some period of inactivity to reduce the time-frame for attack. This can be accomplished by setting a timestamp in the session and forcing the user to log in again if the timestamp is too old. Typically this interval is on the order of minutes or hours.
- Make sessions expire after some (larger) interval regardless of whether they have been active or not. This can also be done by storing a timestamp in the session. Typically this interval is on the order of several days.

**Cross-Site Request Forgery (CSRF)**

This is when an attacker creates a link to an action in a web application which the user is logged into. For example, don’t click here if you are logged into the fictional website `myblog.com`. Attackers can be sneakier than that by including such a link in, for example, an image tag, which is not bound by SOP limitations. E.g. `<img src="http://myblog.com/destroy/page/1" />

A GET resource (e.g. to a page that displays a user’s bank account information) is normally not at risk from CSRF attacks, because GET requests aren’t supposed to modify the state of the server, and attackers can’t read the result of a CSRF request (e.g. there is no way to get the raw “image” data from the above image tag).
POST requests are the main concern. In Rails, protecting against CSRF is easy. Just put a secret key in your application controller like:

```
protect_from_forgery :secret => "123456789012345678901234567890..."
```

If you use CookieStore (the default), you don't need the key:

```
protect_from_forgery
```

This automatically includes a hidden field with a security token in all forms (generated with `form_for`), and if the security token doesn't match what the server expected, the session is reset.

**Tips**

- Use POST for requests which can make stateful changes to the server (not GET)
- Use `form_for` to generate form tags (don't just write `<form> ... </form>`)
- Use `protect_from_forgery` in the application controller

### Injection and Cross-Site Scripting (XSS)

This is when an attacker gets malicious code to run within the security context of an application. Two prominent examples are cross-site scripting and SQL-injection.

**Cross-Site Scripting**

Cross-site scripting is the dual to CSRF; it happens when a malicious user injects client-executable code (i.e. JavaScript) into a web application. For example, an attacker could post `<script>alert('Hello');</script>` in a comment, which would cause a message box to pop up for anyone who visits the page with that comment if the application did not sanitize its comments before displaying them. MySpace has been vulnerable to this kind of attack in the past because it allows users to insert custom code onto their pages. Fortunately for us, the Rails templating engine escapes all strings by default before inserting them onto the page.

**SQL-Injection**

SQL-injection happens when a user passes a malicious snippet of SQL to the web application, and the application does not properly escape all user-generated strings. This can result in all kinds of badness, which was mostly discussed in lecture (refer to the lecture slides). Fortunately, if you use ActiveRecord properly, you are safe against these kinds of attacks. In particular, **DO** make queries like this:

```
Client.where("orders_count = ?", params[:orders])
```
And **DO NOT** make queries like this:

```ruby
Client.where("orders_count = #{params[:orders]}")
```

### Files

**Tips**

- Don't allow redirection to arbitrary URLs
- **Whitelist** uploaded file names (e.g. don't let users upload a file called "../../../etc/passwd")
- Similarly, don't let users download arbitrary files. Don't do this:
  - ○ `send_file('/var/www/uploads/' + params[:filename])`
- If you are using Apache, don't allow users to upload files to DocumentRoot (e.g. the Rails /public folder); this allows users to execute arbitrary code by uploading a file with e.g. a .php extension

### Mass Assignment

Rails will let you do this:

```ruby
User.new(params[:user])
```

This could be bad if, for example, the URL looked like:

```
http://www.example.com/user/signup?user[name]=ow3ned&user[admin]=1
```

This could allow an attacker to create an admin account. To prevent this, use `attr_accessible` in your models. This is a whitelist of mass-assignable attributes. For example, you might have `attr_accessible :name, :email` (notice how :is_admin is not listed as a mass-assignable attribute).

### Don't Log Sensitive Information

By default, Rails logs all requests with their request parameters (GET query arguments and POST parameters). Some parameters, such as the user's password, should NOT be logged in plaintext. Tell Rails not to log this field by adding this to the application configuration:

```ruby
config.filter_parameters << :password
```
Miscellaneous Guidelines

- CSS injection is also a thing (some browsers allow JavaScript to be embedded in CSS)
- Put your admin interface on a separate domain (e.g. admin.mysite.com rather than mysite.com/admin) so that if your site is vulnerable to XSS, the SOP comes to the rescue
- Use CAPTCHAs (reCAPTCHA is a popular implementation)
- Hash and salt passwords, don't store them in plaintext!
- Know how regular expressions work. This is not a good regex to filter filenames: /^[\w\.]+$/.
  Why? The ^ and $ symbols refer to the beginning and end of the line, not the string. Better to use /\A[\w\.]+\z/.
- Prefer whitelists over blacklists.
- For system commands, use `system(program, arg1, arg2, ...)`
- Don't write raw SQL unless you have to -- for Rails, just use ActiveRecord. Do this: `Model.where("x = ? AND y = ?", x, y)` or `Model.where(:x => x, :y => y)`

References

Ruby On Rails Security Guide