6.170 Tutorial 7 - Rails Security

- Introduction
- Sessions
  - Session Hijacking
    - Countermeasures
  - Replay Attacks
    - Countermeasures
  - Session Fixation
    - Countermeasures
- CSRF
- Hackers Love Mass Assignment
  - Countermeasures
- Injection
  - SQL Injection
  - Cross Site Scripting (XSS)
    - Countermeasures
- Logging
- Authorizing Ownership

Prerequisites

Basic understanding of how websites work.
Understanding of how rails implements sessions (CookieStore, SessionStore).

Goals of this tutorial

Understand common security issues and learn how to prevent them in the Rails web framework.

Resources

Ruby on Rails Security Guide: http://guides.rubyonrails.org/security.html
Security Railscast: http://railscasts.com/episodes/178-seven-security-tips?view=asciicast
HTML Escapes: http://api.rubyonrails.org/classes/ERB/Util.html
MySpace Samy Worm (CSS Injection): http://namb.la/popular/tech.html

Introduction

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

Most applications need to keep track of certain state of a particular user. This could be the contents of a shopping basket or the user id of the currently logged in user. Without the idea of sessions, the user would have to identify, and probably authenticate, on every request. Rails will create a new session automatically if a new user accesses the application. It will load an existing session if the user has already used the application.

A session usually consists of a hash of values and a session id, usually a 32-character string, to identify the hash. Every cookie sent to the client's browser includes the session id. And the other way round: the browser will send it to the server on every request from the client.

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.

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 Hijacking

— Stealing a user's session id lets an attacker use the web application in the victim's name.

Many web applications have an authentication system: a user provides a user name and password, the web application checks them and stores the corresponding user id in the session hash. From now on, the session is valid. On every request the application will load the user, identified by the user id in the session, without the need for new authentication. The session id in the cookie identifies the session.

Hence, the cookie serves as temporary authentication for the web application. Everyone who seizes a cookie from someone else, may use the web application as this user – with possibly severe consequences.

Countermeasures

For the web application builder this means to provide a secure connection over SSL. In Rails 3.1 and later, this could be accomplished by always forcing SSL connection in your application config file:

```
config.force_ssl = true
```

Replay Attacks

— Another sort of attack you have to be aware of when using CookieStore is the replay attack.

Imagine a site that uses credit to reward users for being awesome. The credits are stored in the session (which is a bad idea already). A rogue user realizes this and decides to hack the system. Replay attacks happen like this:

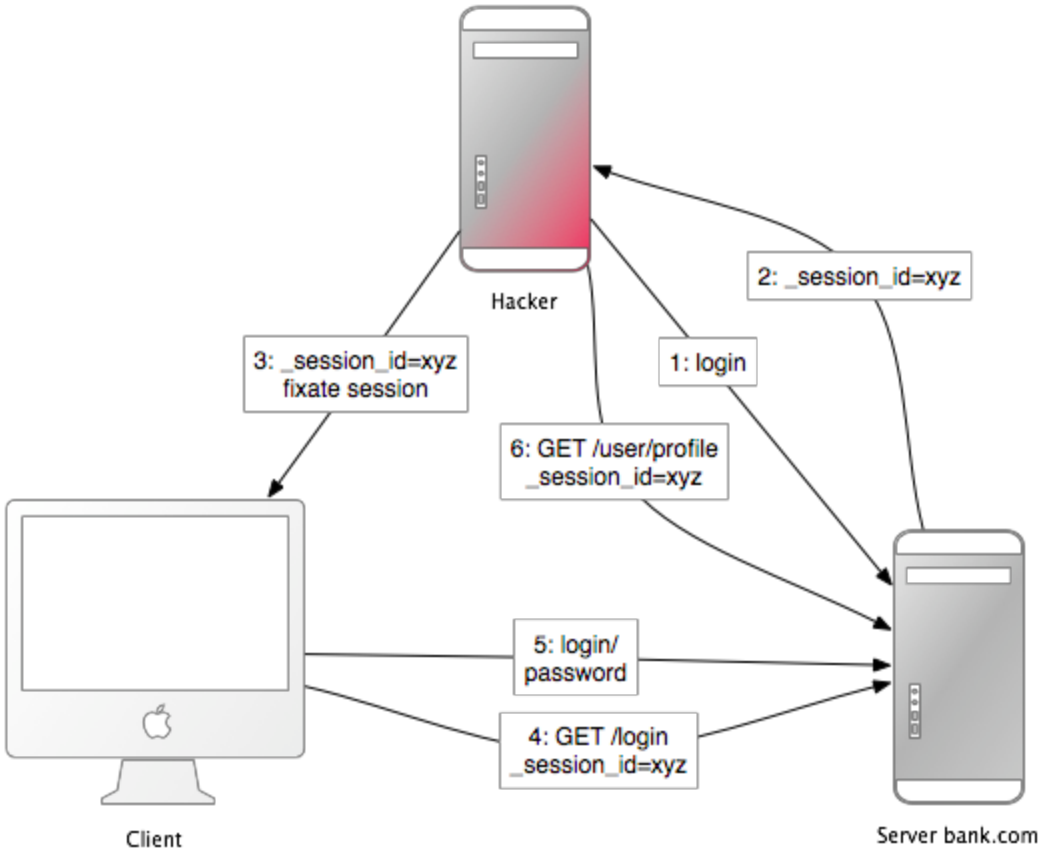
1. A user receives credits, the amount is stored in a session.
2. The user makes a copy of his cookie.
3. The user makes a purchase using his credits that are stored in his session (cookies).
4. The user takes the cookie from step 2 and replaces the current cookie in the browser.
5. The user has all of his credit again!

Countermeasures

Including a nonce (a random value) in the session solves replay attacks. A nonce is valid only once, and the server has to keep track of all the valid nonces. It gets even more complicated if you have several application servers (mongrels). Storing nonces in a database table would defeat the entire purpose of CookieStore (avoiding accessing the database).

The best solution against it is not to store this kind of data in a session, but in the database. In this case store the credit in the database and the `logged_in_user_id` in the session.

Session Fixation



This attack focuses on fixing a user's session id known to the attacker, and forcing the user's browser into using this id. It is therefore not necessary for the attacker to steal the session id afterwards. Here is how this attack works:

1. The attacker creates a valid session id: He loads the login page of the web application where he wants to fix the session, and takes the session id in the cookie from the response (see number 1 and 2 in the image).
2. He possibly maintains the session. Expiring sessions, for example every 20 minutes, greatly reduces the time-frame for attack. Therefore he accesses the web application from time to time in order to keep the session alive.
3. Now the attacker will force the user's browser into using this session id (see number 3 in the image). As you may not change a cookie of another domain (because of the same origin policy), the attacker has to run a JavaScript from the domain of the target web application. Injecting the JavaScript code into the application by XSS accomplishes this attack. Here is an example:
   `<script>document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9";</script>.`
4. The attacker lures the victim to the infected page with the JavaScript code. By viewing the page, the victim's browser will change the session id to the trap session id.
5. As the new trap session is unused, the web application will require the user to

authenticate.

6. From now on, the victim and the attacker will co-use the web application with the same session: The session became valid and the victim didn't notice the attack.

Countermeasures

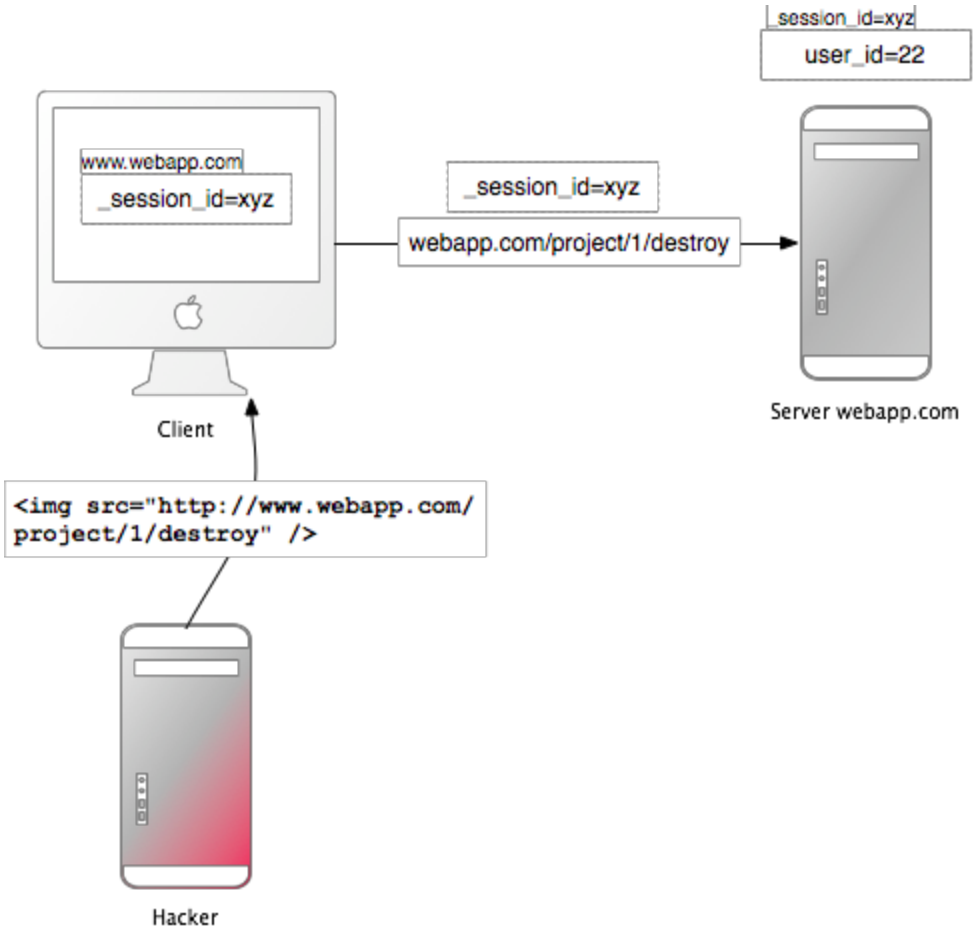
— *One line of code will protect you from session fixation.*

The most effective countermeasure is to *issue a new session identifier* and declare the old one invalid after a successful login. That way, an attacker cannot use the fixed session identifier. This is a good countermeasure against session hijacking, as well. Here is how to create a new session in Rails:

```
reset_session
```

CSRF

— This attack method works by including malicious code or a link in a page that accesses a web application that the user is believed to have authenticated. If the session for that web application has not timed out, an attacker may execute unauthorized commands.



Most Rails applications use cookie-based sessions. Either they store the session id in the cookie and have a server-side session hash, or the entire session hash is on the client-side. In either case the browser will automatically send along the cookie on every request to a domain, if it can find a cookie for that domain. The controversial point is, that it will also send the cookie, if the request comes from a site of a different domain. Let's start with an example:

- Bob browses a message board and views a post from a hacker where there is a crafted HTML image element. The element references a command in Bob's project management application, rather than an image file.
- `<img src="http://www.webapp.com/project/1/destroy">`
- Bob's session at www.webapp.com is still alive, because he didn't log out a few minutes ago.
- By viewing the post, the browser finds an image tag. It tries to load the suspected image from www.webapp.com. As explained before, it will also send along the cookie

with the valid session id.

- The web application at www.webapp.com verifies the user information in the corresponding session hash and destroys the project with the ID 1. It then returns a result page which is an unexpected result for the browser, so it will not display the image.
- Bob doesn't notice the attack — but a few days later he finds out that project number one is gone.

Countermeasures

```
protect_from_forgery :secret => "123456789012345678903456789O..."
```

This will automatically include a security token, calculated from the current session and the server-side secret, in all forms and Ajax requests generated by Rails. You won't need the secret, if you use CookieStorage as session storage. If the security token doesn't match what was expected, the session will be reset.

Hackers Love Mass Assignment

— Without any precautions Model.new(params[:model]) allows attackers to set any database column's value.

The mass-assignment feature may become a problem, as it allows an attacker to set any model's attributes by manipulating the hash passed to a model's `new()` method:

```
def signup
  params[:user] # => {:name => "ow3ned", :admin => true}
  @user = User.new(params[:user])
end
```

Mass-assignment saves you much work, because you don't have to set each value individually. Simply pass a hash to the `new` method, or `assign_attributes=` a hash value, to set the model's attributes to the values in the hash. The problem is that it is often used in conjunction with the parameters (params) hash available in the controller, which may be manipulated by an attacker. He may do so by changing the URL like this:

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

This will set the following parameters in the controller:

```
params[:user] # => {:name => "ow3ned", :admin => true}
```

Note that this vulnerability is not restricted to database columns. Any setter method, unless explicitly protected, is accessible via the `attributes=` method. In fact, this vulnerability is

extended even further with the introduction of nested mass assignment (and nested object forms) in Rails 2.3. The `accepts_nested_attributes_for` declaration provides us the ability to extend mass assignment to model associations (`has_many, has_one, has_and_belongs_to_many`). For example:

```
class Person < ActiveRecord::Base
  has_many :children
  accepts_nested_attributes_for :children
end

class Child < ActiveRecord::Base
  belongs_to :person
end
```

As a result, the vulnerability is extended beyond simply exposing column assignment, allowing attackers the ability to create entirely new records in referenced tables (children in this case).

Countermeasures

To avoid this, Rails provides two class methods in your Active Record class to control access to your attributes. The `attr_protected` method takes a list of attributes that will not be accessible for mass-assignment. For example:

```
attr_protected :admin
```

`attr_protected` also optionally takes a role option using :as which allows you to define multiple mass-assignment groupings. If no role is defined then attributes will be added to the :default role.

```
attr_protected :last_login, :as => :admin
```

A much better way, because it follows the whitelist-principle, is the `attr_accessible` method. It is the exact opposite of `attr_protected`, because *it takes a list of attributes that will be accessible*. All other attributes will be protected. This way you won't forget to protect attributes when adding new ones in the course of development. Here is an example:

```
attr_accessible :name
attr_accessible :name, :is_admin, :as => :admin
```

If you want to set a protected attribute, you will to have to assign it individually:

```
params[:user] # => {:name => "ow3ned", :admin => true}
@user = User.new(params[:user])
@user.admin # => false # not mass-assigned
@user.admin = true
```

```
@user.admin # => true
```

When assigning attributes in Active Record using `attributes=` the :default role will be used. To assign attributes using different roles you should use `assign_attributes` which accepts an optional :as options parameter. If no :as option is provided then the :default role will be used. You can also bypass mass-assignment security by using the `:without_protection` option. Here is an example:

```
@user = User.new

@user.assign_attributes({ :name => 'Josh', :is_admin => true })
@user.name # => Josh
@user.is_admin # => false

@user.assign_attributes({ :name => 'Josh', :is_admin => true }, :as => :admin)
@user.name # => Josh
@user.is_admin # => true

@user.assign_attributes({ :name => 'Josh', :is_admin => true },
:without_protection => true)
@user.name # => Josh
@user.is_admin # => true
```

In a similar way, `new`, `create`, `create!`, `update_attributes`, and `update_attributes!` methods all respect mass-assignment security and accept either `:as` or `:without_protection` options. For example:

```
@user = User.new({ :name => 'Sebastian', :is_admin => true }, :as => :admin)
@user.name # => Sebastian
@user.is_admin # => true

@user = User.create({ :name => 'Sebastian', :is_admin => true },
:without_protection => true)
@user.name # => Sebastian
@user.is_admin # => true
```

A more paranoid technique to protect your whole project would be to enforce that all models define their accessible attributes. This can be easily achieved with a very simple application config option of:

```
config.active_record.whitelist_attributes = true
```

This will create an empty whitelist of attributes available for mass-assignment for all models in

your app. As such, your models will need to explicitly whitelist or blacklist accessible parameters by using an `attr_accessible` or `attr_protected` declaration. This technique is best applied at the start of a new project. However, for an existing project with a thorough set of functional tests, it should be straightforward and relatively quick to use this application config option; run your tests, and expose each attribute (via `attr_accessible` or `attr_protected`) as dictated by your failing tests.

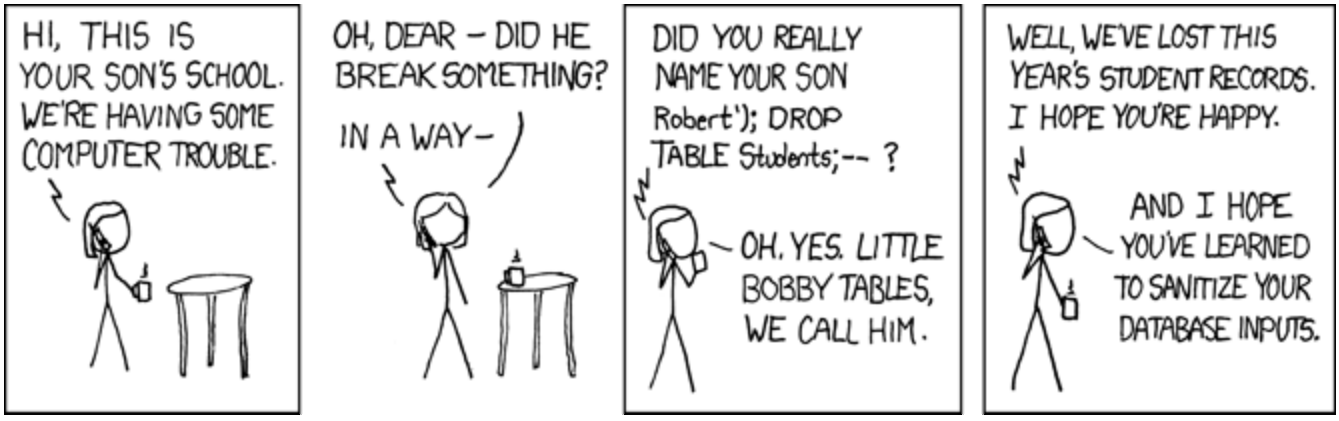
Injection

— Injection is a class of attacks that introduce malicious code or parameters into a web application in order to run it within its security context. Prominent examples of injection are cross-site scripting (XSS) and SQL injection.

Injection is very tricky, because the same code or parameter can be malicious in one context, but totally harmless in another. A context can be a scripting, query or programming language, the shell or a Ruby/Rails method.

SQL Injection

One of the first rules of security is to never trust input from users. In Rails this means taking care of the items in the params hash. The user can control both the keys and the values in the params hash, so all input must be considered suspicious.



One of the most common security issues is known as SQL injection. This happens when user input is placed directly into a SQL query. If a user knows that their input is being directly inserted into a query they can deliberately write input that can retrieve data that they should not see or even alter or destroy data in the database.

Countermeasures

Instead of passing a string to the conditions option, you can pass an array to sanitize tainted strings like this:

```
Model.where("login = ? AND password = ?", user_name, password).first

Model.where(:login => user_name, :password => password).first
```

Cross Site Scripting (XSS)

Cross-site scripting is another common security issue to consider when developing web applications. It happens when you allow the users of your site to enter HTML or JavaScript directly. If you're not escaping the input that's entered by users, your site will be vulnerable to attack.

Assume you're working on a question, answer site (like Quora). If we enter JavaScript into a new question, surrounded by <script> tags, that script will be executed when the page reloads and every time it it viewed afterwards. For example if we entered <script>alert('hello')</script> into a new question and submitted it, every subsequent time the page is viewed the user would see an alert box.

Causing an alert box to be shown on a page is annoying, but cross-site scripting can be used for much more malicious purposes. For example it could be used to read the cookies for the site's other users. The cookie could easily be sent to a remote server where the session id information in the cookie could be used to hijack another user's session.

Countermeasures

Before rails 3, in order to stop these attacks you needed to escape any user input before you display it on the screen. Instead of taking the question's content directly from the database and outputting it into the HTML stream, Rails provides a method simply called h to escape the content before it is output.

```
<% @task.comments.each do |comment| %>
  <p><%= h(comment.content) %></p>
<% end %>
```

In Rails 3, however, output is escaped automatically so there's no need to put the `h` method in your views. The following snippets will result in the same output in Rails 3 (notice the difference is the `h` method).

```
<div class="question">
  <strong><%= link_to question.name, question.url %></strong>
  <p><%= question.content %></p>
</div>

<div class="question">
  <strong><%= link_to h(question.name), question.url %></strong>
  <p><%= h question.content %></p>
</div>
```

The html for both snippets looks the same and the output hasn't been double-escaped. Rails is

clever here; even if we use the h method it will escape the script tag only the once.

If we trust the content that the user enters, say they're an administrative user, and we want to display exactly what they enter then we can use the new `raw` method to do that.

Rails 3 has a concept of HTML-safe strings. This means that we can check if any string is safe to output as HTML by calling the new method `html_safe?` on it. We can mark a string as being HTML-safe by calling the `html_safe` method on it. No actual escaping goes on here. All that happens is that a boolean property is set against a string to determine whether it should be escaped before being output.
So how does this apply in our view template? Well, Rails looks at each piece of output and sees if it's marked as HTML-safe. If it's not then it will be automatically escaped when it is output to the view. If it is safe then it is passed through without being escaped. If we use the `h` method to escape a string it will perform the escaping and mark the string as HTML-safe. This means that Rails 3 will see that the string is safe and not escape it again.

When the `raw` method is used on a string it will be marked as HTML-safe but not escaped, ensuring that the string's content is passed to the output without being changed.

Logging

— Tell Rails not to put passwords in the log files.

By default, Rails logs all requests being made to the web application. But log files can be a huge security issue, as they may contain login credentials, credit card numbers et cetera. When designing a web application security concept, you should also think about what will happen if an attacker got (full) access to the web server. Encrypting secrets and passwords in the database will be quite useless, if the log files list them in clear text. You can *filter certain request parameters from your log files* by appending them to `config.filter_parameters` in the application configuration. These parameters will be marked [FILTERED] in the log.

```
config.filter_parameters << :password
```

Consider the following. Your application has signup and login forms in which users can enter their username and password. By default, Rails will store all form parameters as plain text which means that when we log in, our username and password are stored in the log file.

While we've gone to the effort of encrypting our users' passwords in the database they are still clearly visible in the application's log file.

Other field names can be added to the list of parameters if there are other fields that need to be filtered.

Authorizing Ownership

For project 1, you were making a web analytics engine which allowed a site owner to view analytics about his or her site. If we look at the page for a specific site in our application we can see that the site's id is in the URL. Each user has their own collection of sites and we want to make sure that one user cannot view another's sites.

The URL might be something like: http://localhost:3000/sites/1

Site with ID 1 might belong to us, but what happens if we start altering the id? We now can guess other site IDs and see analytics. To fix this we need to look in our SitesController, in particular at the code that gets the site from the database.

```
def show
  @project = Project.find(params[:id])
end
```

This code will fetch any site by its `id`, with no authorisation to check that the site belongs to the currently logged-in user. There are a few ways we could do this, but one easy way is to use ActiveRecord associations to get the site in the scope of the current user. As a site belongs_to a user we can do this by changing the code above to:

```
def show
  @project = current_user.projects.find(params[:id])
end
```

This will now scope the search to the sites that belong to the currently logged-in user. If we try to view another user's project now we'll see an error.

```
ActiveRecord::RecordNotFound in SitesController#show
Couldn't find Site with ID=2 AND ("sites".user_id = 1)
```

Note that the SQL condition in the error message has the current user's id in the select condition. A RecordNotFound error means that when this application runs in production mode the user will see a 404 page.

6.170 Software Studio
Spring 2013