

# Ruby on Rails Security Guide

Ruby on Rails

# Ruby on Rails Security Guide

## [Ruby on Rails Security Guide](#)

### [1 Introduction](#)

### [2 Sessions](#)

#### [2.1 What are Sessions?](#)

#### [2.2 Session id](#)

#### [2.3 Session Hijacking](#)

#### [2.4 Session Guidelines](#)

#### [2.5 Session Storage](#)

#### [2.6 Replay Attacks for CookieStore Sessions](#)

#### [2.7 Session Fixation](#)

#### [2.8 Session Fixation - Countermeasures](#)

#### [2.9 Session Expiry](#)

### [3 Cross-Site Request Forgery \(CSRF\)](#)

#### [3.1 CSRF Countermeasures](#)

### [4 Redirection and Files](#)

#### [4.1 Redirection](#)

##### [4.1.1 Self-contained XSS](#)

#### [4.2 File Uploads](#)

#### [4.3 Executable Code in File Uploads](#)

#### [4.4 File Downloads](#)

### [5 Intranet and Admin Security](#)

#### [5.1 Additional Precautions](#)

### [6 User Management](#)

#### [6.1 Brute-Forcing Accounts](#)

#### [6.2 Account Hijacking](#)

##### [6.2.1 Passwords](#)

##### [6.2.2 E-Mail](#)

##### [6.2.3 Other](#)

#### [6.3 CAPTCHAs](#)

#### [6.4 Logging](#)

#### [6.5 Good Passwords](#)

#### [6.6 Regular Expressions](#)

#### [6.7 Privilege Escalation](#)

### [7 Injection](#)

#### [7.1 Whitelists versus Blacklists](#)

#### [7.2 SQL Injection](#)

##### [7.2.1 Introduction](#)

##### [7.2.2 Bypassing Authorization](#)

##### [7.2.3 Unauthorized Reading](#)

##### [7.2.4 Countermeasures](#)

## [7.3 Cross-Site Scripting \(XSS\)](#)

### [7.3.1 Entry Points](#)

### [7.3.2 HTML/JavaScript Injection](#)

### [7.3.3 Examples from the Underground](#)

## [7.4 CSS Injection](#)

### [7.4.1 Countermeasures](#)

## [7.5 Textile Injection](#)

### [7.5.1 Countermeasures](#)

## [7.6 Ajax Injection](#)

## [7.7 Command Line Injection](#)

## [7.8 Header Injection](#)

### [7.8.1 Response Splitting](#)

## [8 Unsafe Query Generation](#)

## [9 Default Headers](#)

## [10 Environmental Security](#)

## [11 Additional Resources](#)

## [Feedback](#)

# Ruby on Rails Security Guide

This manual describes common security problems in web applications and how to avoid them with Rails. After reading this guide, you will know:

- All countermeasures that are highlighted.
- The concept of sessions in Rails, what to put in there and popular attack methods.
- How just visiting a site can be a security problem (with CSRF).
- What you have to pay attention to when working with files or providing an administration interface.
- How to manage users: Logging in and out and attack methods on all layers.
- And the most popular injection attack methods.

# 1 Introduction

Web application frameworks are made to help developers build web applications. Some of them also help you with securing the web application. In fact one framework is not more secure than another: If you use it correctly, you will be able to build secure apps with many frameworks. Ruby on Rails has some clever helper methods, for example against SQL injection, so that this is hardly a problem.

In general there is no such thing as plug-n-play security. Security depends on the people using the framework, and sometimes on the development method. And it depends on all layers of a web application environment: The back-end storage, the web server and the web application itself (and possibly other layers or applications).

The Gartner Group however estimates that 75% of attacks are at the web application layer, and found out "that out of 300 audited sites, 97% are vulnerable to attack". This is because web applications are relatively easy to attack, as they are simple to understand and manipulate, even by the lay person.

The threats against web applications include user account hijacking, bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. Or an attacker might be able to install a Trojan horse program or unsolicited e-mail sending software, aim at financial enrichment or cause brand name damage by modifying company resources. In order to prevent attacks, minimize their impact and remove points of attack, first of all, you have to fully understand the attack methods in order to find the correct countermeasures. That is what this guide aims at.

In order to develop secure web applications you have to keep up to date on all layers and know your enemies. To keep up to date subscribe to security mailing lists, read security blogs and make updating and security checks a habit (check the [Additional Resources](#) chapter). It is done manually because that's how you find the nasty logical security problems.

# 2 Sessions

A good place to start looking at security is with sessions, which can be vulnerable to particular attacks.

## 2.1 What are Sessions?

### Note:

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. In Rails you can save and retrieve values using the session method:

```
session[:user_id] = @current_user.id
User.find(session[:user_id])
```

## 2.2 Session id

### Note:

A session id consists of the hash value of a random string. The random string is the current time, a random number between 0 and 1, the process id number of the Ruby interpreter (also basically a random number) and a constant string. Currently it is not feasible to brute-force Rails' session ids. To date MD5 is uncompromised, but there have been collisions, so it is theoretically possible to create another input text with the same hash value. But this has had no security impact to date.

## 2.3 Session Hijacking

### Warning:

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. Anyone who seizes a cookie from someone else, may use the web application as this user - with possibly severe consequences. Here are some ways to hijack a session, and their countermeasures:

- Sniff the cookie in an insecure network. A wireless LAN can be an example of such a network. In an unencrypted wireless LAN it is especially easy to listen to the traffic of all connected clients. For the web application builder this means to . In Rails 3.1 and later, this could be accomplished by always forcing SSL connection in your application config file:

```
config.force_ssl = true
```

- Most people don't clear out the cookies after working at a public terminal. So if the last user didn't log out of a web application, you would be able to use it as this user. Provide the user with a in the web application, and .
- Many cross-site scripting (XSS) exploits aim at obtaining the user's cookie. You'll read [more about XSS](#) later.
- Instead of stealing a cookie unknown to the attacker, they fix a user's session identifier (in the cookie) known to them. Read more about this so-called session fixation later.

The main objective of most attackers is to make money. 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, according to the [Symantec Global Internet Security Threat Report](#).

## 2.4 Session Guidelines

Here are some general guidelines on sessions.

- . Instead you should store them in the database and save their id in the session. This will eliminate synchronization headaches and it won't fill up your session storage space (depending on what session storage you chose, see below). This will also be a good idea, if you modify the structure of an object and old versions of it are still in some user's cookies. With server-side session storages you can clear out the sessions, but with client-side storages, this is hard to mitigate.
- . If the user clears their cookies or closes the browser, they will be lost. And with a client-side session storage, the user can read the data.

## 2.5 Session Storage

### Note:

Rails 2 introduced a new default session storage, CookieStore. CookieStore saves the session hash directly in a cookie on the client-side. The server retrieves the session hash from the cookie and eliminates the need for a session id. That will greatly increase the speed of the application, but it is a controversial storage option and you have to think about the security implications of it:

- Cookies imply a strict size limit of 4kB. This is fine as you should not store large amounts of data in a session anyway, as described before. .
- The client can see everything you store in a session, because it is stored in clear-text (actually Base64-encoded, so not encrypted). So, of course, . To prevent session hash tampering, a digest is calculated from the session with a server-side secret and inserted into the end of the cookie.

That means the security of this storage depends on this secret (and on the digest algorithm, which defaults to SHA1, for compatibility). So .

`secrets.secret_key_base` is used for specifying a key which allows sessions for the application to be verified against a known secure key to prevent tampering. Applications get `secrets.secret_key_base` initialized to a random key present in `config/secrets.yml`, e.g.:

```
development:
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Older versions of Rails use `CookieStore`, which uses `secret_token` instead of `secret_key_base` that is used by `EncryptedCookieStore`. Read the upgrade documentation for more information.

If you have received an application where the secret was exposed (e.g. an application whose source was shared), strongly consider changing the secret.

## 2.6 Replay Attacks for CookieStore Sessions

### Info:

It works like this:

- A user receives credits, the amount is stored in a session (which is a bad idea anyway, but we'll do this for demonstration purposes).
- The user buys something.
- The new adjusted credit value is stored in the session.
- The user takes the cookie from the first step (which they previously copied) and replaces the current cookie in the browser.
- The user has their original credit back.

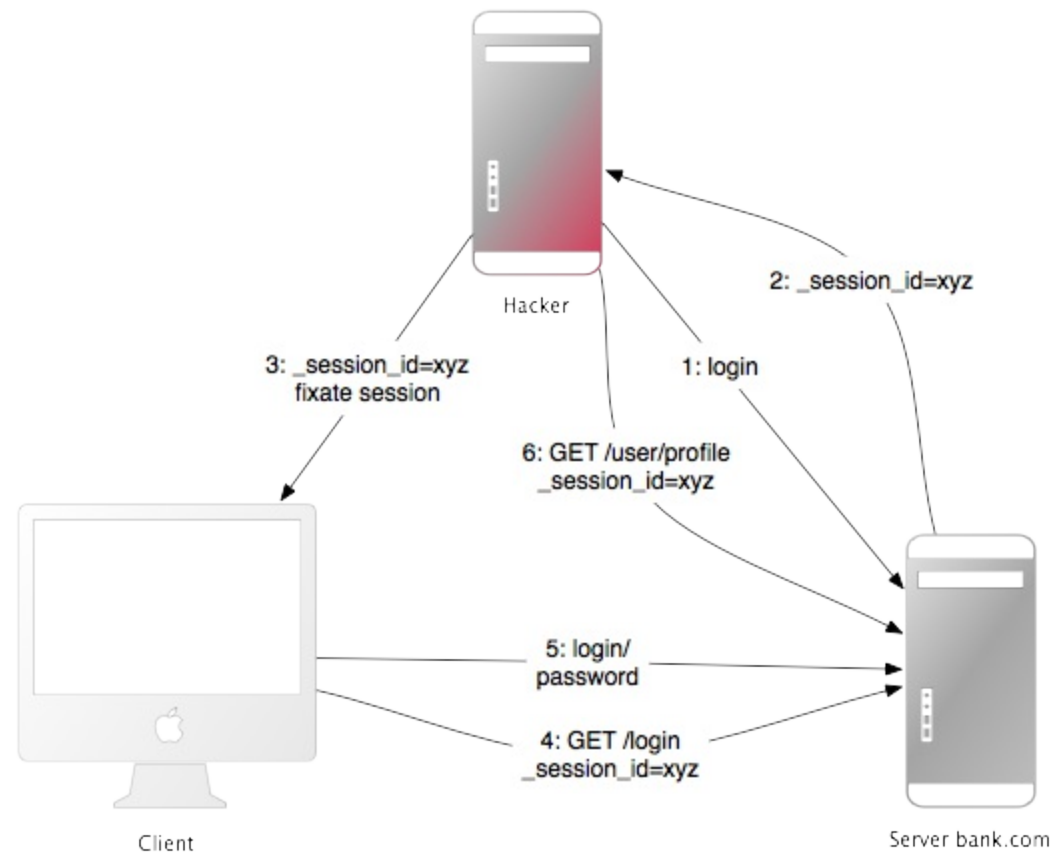
Including a nonce (a random value) in the session solves replay attacks. A nonce is valid only once, and the server has to keep track of all the valid nonces. It gets even more complicated if you have several application servers (mongrels). Storing nonces in a database table would defeat the entire purpose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



The best . In this case store the credit in the database and the logged\_in\_user\_id in the session.

## 2.7 Session Fixation

**Note:**



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:

- The attacker creates a valid session id: They load the login page of the web application where they want to fix the session, and take the session id in the cookie from the response (see number 1 and 2 in the image).
- They maintain the session by accessing the web application periodically in order to keep an expiring session alive.
- The attacker forces 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>.
```

 Read more about XSS and injection later on.
- 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.
- As the new trap session is unused, the web application will require the user to authenticate.
- 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.

## 2.8 Session Fixation - Countermeasures

### Info:

The most effective countermeasure is to 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
```

If you use the popular `RestfulAuthentication` plugin for user management, add `reset_session` to the `SessionsController#create` action. Note that this removes any value from the session, .

Another countermeasure is to , verify them every time a request comes in, and deny access, if the information does not match. Such properties could be the remote IP address or the user agent (the web browser name), though the latter is less user-specific. When saving the IP address, you have to bear in mind that there are Internet service providers or large organizations that put their users behind proxies. , so these users will not be able to use your application, or only in a limited way.

## 2.9 Session Expiry

### Note:

One possibility is to set the expiry time-stamp of the cookie with the session id. However the client can edit cookies that are stored in the web browser so expiring sessions on the server is safer. Here is an example of how to . Call `Session.sweep("20 minutes")` to expire sessions that were used longer than 20 minutes ago.

```
class Session < ActiveRecord::Base
  def self.sweep(time = 1.hour)
    if time.is_a?(String)
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

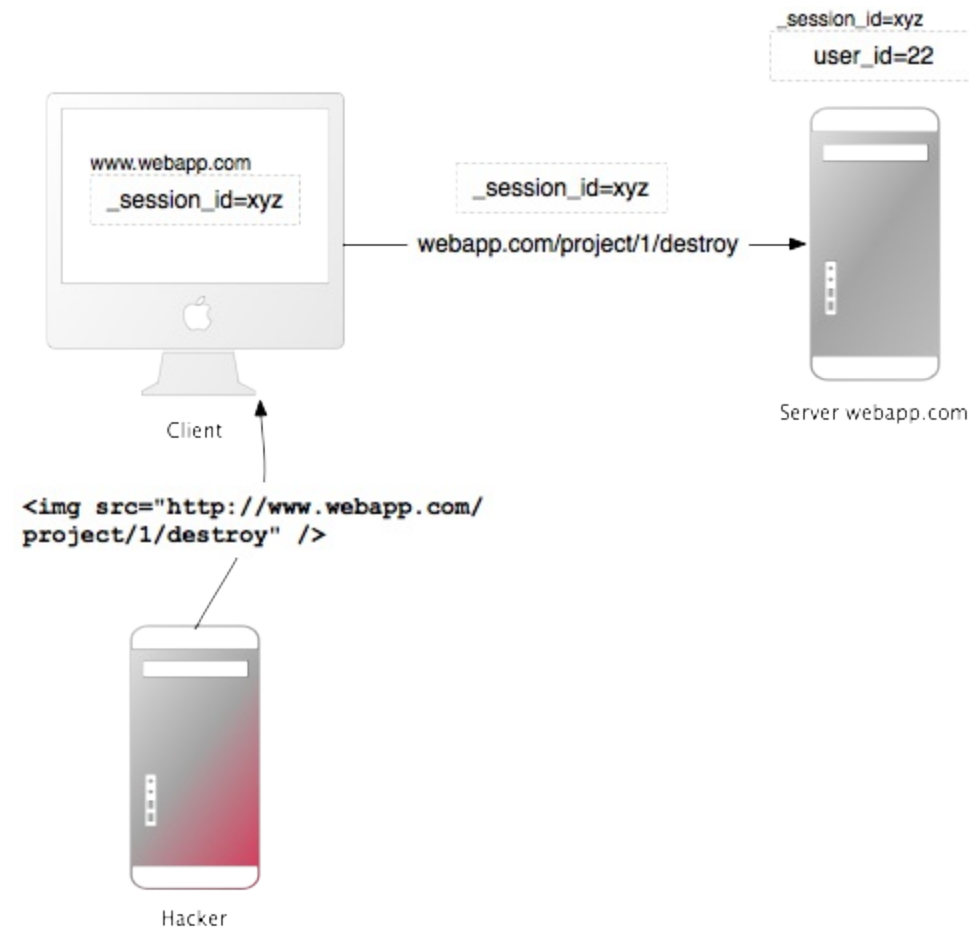
    delete_all "updated_at < '#{time.ago.to_s(:db)}'"
  end
end
```

The section about session fixation introduced the problem of maintained sessions. An attacker maintaining a session every five minutes can keep the session alive forever, although you are expiring sessions. A simple solution for this would be to add a `created_at` column to the sessions table. Now you can delete sessions that were created a long time ago. Use this line in the sweep method above:

```
delete_all "updated_at < '#{time.ago.to_s(:db)}' OR
  created_at < '#{2.days.ago.to_s(:db)}'"
```

# 3 Cross-Site Request Forgery (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.



In the [session chapter](#) you have learned that 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.
- ``
- 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.

It is important to notice that the actual crafted image or link doesn't necessarily have to be situated in the web application's domain, it can be anywhere - in a forum, blog post or email.

CSRF appears very rarely in CVE (Common Vulnerabilities and Exposures) - less than 0.1% in 2006 - but it really is a 'sleeping giant' [Grossman]. This is in stark contrast to the results in many security contract works - .

## 3.1 CSRF Countermeasures

### Note:

The HTTP protocol basically provides two main types of requests - GET and POST (and more, but they are not supported by most browsers). The World Wide Web Consortium (W3C) provides a checklist for choosing HTTP GET or POST:

- The interaction is more (i.e., it is a safe operation such as a query, read operation, or lookup).
- The interaction is more , or
- The interaction of the resource in a way that the user would perceive (e.g., a subscription to a service), or
- The user is of the interaction.

If your web application is RESTful, you might be used to additional HTTP verbs, such as PATCH, PUT or DELETE. Most of today's web browsers, however do not support them - only GET and POST. Rails uses a hidden `_method` field to handle this barrier.

. Here is an example for a link which displays [www.harmless.com](http://www.harmless.com) as destination in the browser's status bar. In fact it dynamically creates a new form that sends a POST request.

```
<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 harmless survey</a>
```

Or the attacker places the code into the onmouseover event handler of an image:

```
` tag to make a cross-site request to a URL with a JSONP or JavaScript response. The response is executable code that the attacker can find a

way to run, possibly extracting sensitive data. To protect against this data leakage, we disallow cross-site `<script>` tags. Only Ajax requests may have JavaScript responses since `XmlHttpRequest` is subject to the browser Same-Origin policy - meaning only your site can initiate the request.

To protect against all other forged requests, we introduce a that our site knows but other sites don't know. We include the security token in requests and verify it on the server. This is a one-liner in your application controller, and is the default for newly created rails applications:

```
protect_from_forgery with: :exception
```

This will automatically include a security token in all forms and Ajax requests generated by Rails. If the security token doesn't match what was expected, an exception will be thrown.

It is common to use persistent cookies to store user information, with `cookies.permanent` for example. In this case, the cookies will not be cleared and the out of the box CSRF protection will not be effective. If you are using a different cookie store than the session for this information, you must handle what to do with it yourself:

```
rescue_from ActionController::InvalidAuthenticityToken do |exception|
  sign_out_user # Example method that will destroy the user cookies
end
```

The above method can be placed in the `ApplicationController` and will be called when a CSRF token is not present or is incorrect on a non-GET request.

Note that . XSS gives the attacker access to all elements on a page, so they can read the CSRF security token from a form or directly submit the form. Read [more about XSS](#) later.

# 4 Redirection and Files

Another class of security vulnerabilities surrounds the use of redirection and files in web applications.

## 4.1 Redirection

### Warning:

Whenever the user is allowed to pass (parts of) the URL for redirection, it is possibly vulnerable. The most obvious attack would be to redirect users to a fake web application which looks and feels exactly as the original one. This so-called phishing attack works by sending an unsuspecting link in an email to the users, injecting the link by XSS in the web application or putting the link into an external site. It is unsuspecting, because the link starts with the URL to the web application and the URL to the malicious site is hidden in the redirection parameter: <http://www.example.com/site/redirect?to=www.attacker.com>. Here is an example of a legacy action:

```
def legacy
  redirect_to(params.update(action: 'main'))
end
```

This will redirect the user to the main action if they tried to access a legacy action. The intention was to preserve the URL parameters to the legacy action and pass them to the main action. However, it can be exploited by attacker if they included a host key in the URL:

```
http://www.example.com/site/legacy?param1=xy&param2=23&host=www.attacker.com
```

If it is at the end of the URL it will hardly be noticed and redirects the user to the attacker.com host. A simple countermeasure would be to (again a whitelist approach, as opposed to removing unexpected parameters). .

### 4.1.1 Self-contained XSS

Another redirection and self-contained XSS attack works in Firefox and Opera by the use of the data protocol. This protocol displays its contents directly in the browser and can be anything from HTML or JavaScript to entire images:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K
```

This example is a Base64 encoded JavaScript which displays a simple message box. In a redirection URL, an attacker could redirect to this URL with the malicious code in it. As a countermeasure, .

## 4.2 File Uploads

## Note:

Many web applications allow users to upload files. as an attacker could use a malicious file name to overwrite any file on the server. If you store file uploads at /var/www/uploads, and the user enters a file name like ".././../etc/passwd", it may overwrite an important file. Of course, the Ruby interpreter would need the appropriate permissions to do so - one more reason to run web servers, database servers and other programs as a less privileged Unix user.

When filtering user input file names, . Think of a situation where the web application removes all "../" in a file name and an attacker uses a string such as "....//" - the result will be "../". It is best to use a whitelist approach, which . This is opposed to a blacklist approach which attempts to remove not allowed characters. In case it isn't a valid file name, reject it (or replace not accepted characters), but don't remove them. Here is the file name sanitizer from the [attachment\\_fu plugin](#):

```
def sanitize_filename(filename)
  filename.strip.tap do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.sub! /\A.*(\\|\/)/, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^w\.\-]/, '_'
  end
end
```

A significant disadvantage of synchronous processing of file uploads (as the attachment\_fu plugin may do with images), is its . An attacker can synchronously start image file uploads from many computers which increases the server load and may eventually crash or stall the server.

The solution to this is best to : Save the media file and schedule a processing request in the database. A second process will handle the processing of the file in the background.

## 4.3 Executable Code in File Uploads

### Warning:

The popular Apache web server has an option called DocumentRoot. This is the home directory of the web site, everything in this directory tree will be served by the web server. If there are files with a certain file name extension, the code in it will be executed when requested (might require some options to be set). Examples for this are PHP and CGI files. Now think of a situation where an attacker uploads a file "file.cgi" with code in it, which will be executed when someone downloads the file.

, store files at least one level downwards.

## 4.4 File Downloads

## Note:

Just as you have to filter file names for uploads, you have to do so for downloads. The `send_file()` method sends files from the server to the client. If you use a file name, that the user entered, without filtering, any file can be downloaded:

```
send_file('/var/www/uploads/' + params[:filename])
```

Simply pass a file name like `"../../etc/passwd"` to download the server's login information. A simple solution against this, is to :

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '../..files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename !=
      File.expand_path(File.join(File.dirname(filename), '../..../'))
send_file filename, disposition: 'inline'
```

Another (additional) approach is to store the file names in the database and name the files on the disk after the ids in the database. This is also a good approach to avoid possible code in an uploaded file to be executed. The `attachment_fu` plugin does this in a similar way.



# 5 Intranet and Admin Security

Intranet and administration interfaces are popular attack targets, because they allow privileged access. Although this would require several extra-security measures, the opposite is the case in the real world.

In 2007 there was the first tailor-made trojan which stole information from an Intranet, namely the "Monster for employers" web site of Monster.com, an online recruitment web application. Tailor-made Trojans are very rare, so far, and the risk is quite low, but it is certainly a possibility and an example of how the security of the client host is important, too. However, the highest threat to Intranet and Admin applications are XSS and CSRF.

If your application re-displays malicious user input from the extranet, the application will be vulnerable to XSS. User names, comments, spam reports, order addresses are just a few uncommon examples, where there can be XSS.

Having one single place in the admin interface or Intranet, where the input has not been sanitized, makes the entire application vulnerable. Possible exploits include stealing the privileged administrator's cookie, injecting an iframe to steal the administrator's password or installing malicious software through browser security holes to take over the administrator's computer.

Refer to the Injection section for countermeasures against XSS. It is also in an Intranet or administration interface.

Cross-Site Request Forgery (CSRF), also known as Cross-Site Reference Forgery (XSRF), is a gigantic attack method, it allows the attacker to do everything the administrator or Intranet user may do. As you have already seen above how CSRF works, here are a few examples of what attackers can do in the Intranet or admin interface.

A real-world example is a [router reconfiguration by CSRF](#). The attackers sent a malicious e-mail, with CSRF in it, to Mexican users. The e-mail claimed there was an e-card waiting for them, but it also contained an image tag that resulted in a HTTP-GET request to reconfigure the user's router (which is a popular model in Mexico). The request changed the DNS-settings so that requests to a Mexico-based banking site would be mapped to the attacker's site. Everyone who accessed the banking site through that router saw the attacker's fake web site and had their credentials stolen.

Another example changed Google AdSense's e-mail address and password by. If the victim was logged into Google AdSense, the administration interface for Google advertisements campaigns, an attacker could change their credentials.

Another popular attack is to spam your web application, your blog or forum to propagate malicious XSS. Of course, the attacker has to know the URL structure, but most Rails URLs are quite straightforward or they will be easy to find out, if it is an open-source application's admin interface. The attacker may even do 1,000 lucky guesses by just including malicious IMG-tags which try every

possible combination.

For .

## 5.1 Additional Precautions

The common admin interface works like this: it's located at [www.example.com/admin](http://www.example.com/admin), may be accessed only if the admin flag is set in the User model, re-displays user input and allows the admin to delete/add/edit whatever data desired. Here are some thoughts about this:

- It is very important to : What if someone really got hold of your cookies or user credentials. You could for the admin interface to limit the possibilities of the attacker. Or how about for the admin interface, other than the ones used for the public part of the application. Or a ?
- Does the admin really have to access the interface from everywhere in the world? Think about . Examine `request.remote_ip` to find out about the user's IP address. This is not bullet-proof, but a great barrier. Remember that there might be a proxy in use, though.
- such as `admin.application.com` and make it a separate application with its own user management. This makes stealing an admin cookie from the usual domain, [www.application.com](http://www.application.com), impossible. This is because of the same origin policy in your browser: An injected (XSS) script on [www.application.com](http://www.application.com) may not read the cookie for `admin.application.com` and vice-versa.

# 6 User Management

## Note:

There are a number of authentication plug-ins for Rails available. Good ones, such as the popular [devise](#) and [authlogic](#), store only encrypted passwords, not plain-text passwords. In Rails 3.1 you can use the built-in `has_secure_password` method which has similar features.

Every new user gets an activation code to activate their account when they get an e-mail with a link in it. After activating the account, the `activation_code` columns will be set to NULL in the database. If someone requested an URL like these, they would be logged in as the first activated user found in the database (and chances are that this is the administrator):

```
http://localhost:3006/user/activate
http://localhost:3006/user/activate?id=
```

This is possible because on some servers, this way the parameter `id`, as in `params[:id]`, would be `nil`. However, here is the finder from the activation action:

```
User.find_by_activation_code(params[:id])
```

If the parameter was `nil`, the resulting SQL query will be

```
SELECT * FROM users WHERE (users.activation_code IS NULL) LIMIT 1
```

And thus it found the first user in the database, returned it and logged them in. You can find out more about it in [this blog post](#). Moreover, you can review your application to find more flaws like this.

## 6.1 Brute-Forcing Accounts

## Note:

A list of user names for your web application may be misused to brute-force the corresponding passwords, because most people don't use sophisticated passwords. Most passwords are a combination of dictionary words and possibly numbers. So armed with a list of user names and a dictionary, an automatic program may find the correct password in a matter of minutes.

Because of this, most web applications will display a generic error message "user name or password not correct", if one of these are not correct. If it said "the user name you entered has not been found", an attacker could automatically compile a list of user names.

However, what most web application designers neglect, are the forgot-password pages. These pages often admit that the entered user name or e-mail address has (not) been found. This allows an attacker to compile a list of user names and brute-force the accounts.

In order to mitigate such attacks, . Moreover, you can . Note, however, that this is not a bullet-proof solution against automatic programs, because these programs may change their IP address exactly as often. However, it raises the barrier of an attack.

## 6.2 Account Hijacking

Many web applications make it easy to hijack user accounts. Why not be different and make it more difficult?.

### 6.2.1 Passwords

Think of a situation where an attacker has stolen a user's session cookie and thus may co-use the application. If it is easy to change the password, the attacker will hijack the account with a few clicks. Or if the change-password form is vulnerable to CSRF, the attacker will be able to change the victim's password by luring them to a web page where there is a crafted IMG-tag which does the CSRF. As a countermeasure, , of course. And .

### 6.2.2 E-Mail

However, the attacker may also take over the account by changing the e-mail address. After they change it, they will go to the forgotten-password page and the (possibly new) password will be mailed to the attacker's e-mail address. As a countermeasure .

### 6.2.3 Other

Depending on your web application, there may be more ways to hijack the user's account. In many cases CSRF and XSS will help to do so. For example, as in a CSRF vulnerability in [Google Mail](#). In this proof-of-concept attack, the victim would have been lured to a web site controlled by the attacker. On that site is a crafted IMG-tag which results in a HTTP GET request that changes the filter settings of Google Mail. If the victim was logged in to Google Mail, the attacker would change the filters to forward all e-mails to their e-mail address. This is nearly as harmful as hijacking the entire account. As a countermeasure, .

## 6.3 CAPTCHAs

### Info:

But not only spam robots (bots) are a problem, but also automatic login bots. A popular CAPTCHA API is [reCAPTCHA](#) which displays two distorted images of words from old books. It also adds an angled line, rather than a distorted background and high levels of warping on the text as earlier CAPTCHAs did, because the latter were broken. As a bonus, using reCAPTCHA helps to digitize old books. [ReCAPTCHA](#) is also a Rails plug-in with the same name as the API.

You will get two keys from the API, a public and a private key, which you have to put into your Rails

environment. After that you can use the `recaptcha_tags` method in the view, and the `verify_recaptcha` method in the controller. `Verify_recaptcha` will return false if the validation fails. The problem with CAPTCHAs is, they are annoying. Additionally, some visually impaired users have found certain kinds of distorted CAPTCHAs difficult to read. The idea of negative CAPTCHAs is not to ask a user to prove that they are human, but reveal that a spam robot is a bot.

Most bots are really dumb, they crawl the web and put their spam into every form's field they can find. Negative CAPTCHAs take advantage of that and include a "honeypot" field in the form which will be hidden from the human user by CSS or JavaScript.

Here are some ideas how to hide honeypot fields by JavaScript and/or CSS:

- position the fields off of the visible area of the page
- make the elements very small or color them the same as the background of the page
- leave the fields displayed, but tell humans to leave them blank

The most simple negative CAPTCHA is one hidden honeypot field. On the server side, you will check the value of the field: If it contains any text, it must be a bot. Then, you can either ignore the post or return a positive result, but not saving the post to the database. This way the bot will be satisfied and moves on. You can do this with annoying users, too.

You can find more sophisticated negative CAPTCHAs in Ned Batchelder's [blog post](#):

- Include a field with the current UTC time-stamp in it and check it on the server. If it is too far in the past, or if it is in the future, the form is invalid.
- Randomize the field names
- Include more than one honeypot field of all types, including submission buttons

Note that this protects you only from automatic bots, targeted tailor-made bots cannot be stopped by this. So .

## 6.4 Logging

### Warning:

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 by appending them to `config.filter_parameters` in the application configuration. These parameters will be marked [FILTERED] in the log.

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

## 6.5 Good Passwords

### Info:

Bruce Schneier, a security technologist, [has analyzed](#) 34,000 real-world user names and passwords from the MySpace phishing attack mentioned [below](#). It turns out that most of the passwords are quite easy to crack. The 20 most common passwords are:

password1, abc123, myspace1, password, blink182, qwerty1, \*\*\*\*you, 123abc, baseball1, football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1, iloveyou1, and monkey.

It is interesting that only 4% of these passwords were dictionary words and the great majority is actually alphanumeric. However, password cracker dictionaries contain a large number of today's passwords, and they try out all kinds of (alphanumerical) combinations. If an attacker knows your user name and you use a weak password, your account will be easily cracked.

A good password is a long alphanumeric combination of mixed cases. As this is quite hard to remember, it is advisable to enter only the . For example "The quick brown fox jumps over the lazy dog" will be "Tqbfjotld". Note that this is just an example, you should not use well known phrases like these, as they might appear in cracker dictionaries, too.

## 6.6 Regular Expressions

### Info:

Ruby uses a slightly different approach than many other languages to match the end and the beginning of a string. That is why even many Ruby and Rails books get this wrong. So how is this a security threat? Say you wanted to loosely validate a URL field and you used a simple regular expression like this:

```
/^https?:\/\/[^\n]+$/i
```

This may work fine in some languages. However, . And thus a URL like this passes the filter without problems:

```
javascript:exploit_code();/*  
http://hi.com  
*/
```

This URL passes the filter because the regular expression matches - the second line, the rest does not matter. Now imagine we had a view that showed the URL like this:

```
link_to "Homepage", @user.homepage
```

The link looks innocent to visitors, but when it's clicked, it will execute the JavaScript function

"exploit\_code" or any other JavaScript the attacker provides.

To fix the regular expression, `^` and `$` should be used instead of `^` and `$`, like so:

```
/\Ahttps?:\/\/[^\n]+\z/i
```

Since this is a frequent mistake, the format validator (`validates_format_of`) now raises an exception if the provided regular expression starts with `^` or ends with `$`. If you do need to use `^` and `$` instead of `^` and `$` (which is rare), you can set the `:multiline` option to `true`, like so:

```
# content should include a line "Meanwhile" anywhere in the string
validates :content, format: { with: /^Meanwhile$/, multiline: true }
```

Note that this only protects you against the most common mistake when using the format validator - you always need to keep in mind that `^` and `$` match the beginning and line end in Ruby, and not the beginning and end of a string.

## 6.7 Privilege Escalation

### Warning:

The most common parameter that a user might tamper with, is the `id` parameter, as in `http://www.domain.com/project/1`, whereas `1` is the `id`. It will be available in `params` in the controller. There, you will most likely do something like this:

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

This is alright for some web applications, but certainly not if the user is not authorized to view all projects. If the user changes the `id` to `42`, and they are not allowed to see that information, they will have access to it anyway. Instead, :

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

Depending on your web application, there will be many more parameters the user can tamper with. As a rule of thumb, .

Don't be fooled by security by obfuscation and JavaScript security. The Web Developer Toolbar for Mozilla Firefox lets you review and change every form's hidden fields. . The Live Http Headers plugin for Mozilla Firefox logs every request and may repeat and change them. That is an easy way to bypass any JavaScript validations. And there are even client-side proxies that allow you to intercept any request and response from and to the Internet.



# 7 Injection

## Info:

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. The following sections will cover all important contexts where injection attacks may happen. The first section, however, covers an architectural decision in connection with Injection.

## 7.1 Whitelists versus Blacklists

### Note:

A blacklist can be a list of bad e-mail addresses, non-public actions or bad HTML tags. This is opposed to a whitelist which lists the good e-mail addresses, public actions, good HTML tags and so on. Although sometimes it is not possible to create a whitelist (in a SPAM filter, for example), :

- Use `before_action` only: [...] instead of `except`: [...]. This way you don't forget to turn it off for newly added actions.
- Allow **instead of removing**

", which makes an attack work. That's why a whitelist approach is better, using the updated Rails 2 method `sanitize()`:

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br c:
s = sanitize(user_input, tags: tags, attributes: %w(href title))
```

This allows only the given tags and does a good job, even against all kinds of tricks and malformed tags.

As a second step, , especially when re-displaying user input, which hasn't been input-filtered (as in the search form example earlier on). to replace the HTML input characters `&`, `"`, `<`, `>` by their uninterpreted representations in HTML (`&amp;`, `&quot;`, `&lt;`, and `&gt;`). However, it can easily happen that the programmer forgets to use it, so `_` it is recommended to use the `SafeErb` gem. `SafeErb` reminds you to escape strings from external sources.

### 7.3.2.4 Obfuscation and Encoding Injection

Network traffic is mostly based on the limited Western alphabet, so new character encodings, such as Unicode, emerged, to transmit characters in other languages. But, this is also a threat to



web applications, as malicious code can be hidden in different encodings that the web browser might be able to process, but the web application might not. Here is an attack vector in UTF-8 encoding:

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

This example pops up a message box. It will be recognized by the above `sanitize()` filter, though. A great tool to obfuscate and encode strings, and thus "get to know your enemy", is the [Hackvertor](#). Rails' `sanitize()` method does a good job to fend off encoding attacks.

### 7.3.3 Examples from the Underground

The following is an excerpt from the [Js.Yamanner@m](#) Yahoo! Mail [worm](#). It appeared on June 11, 2006 and was the first webmail interface worm:

```
<img src='http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'  
target=""onload="var http_request = false;    var Email = '';  
var IDList = '';    var CRumb = '';    function makeRequest(url, Func, Method, Pa
```

The worms exploits a hole in Yahoo's HTML/JavaScript filter, which usually filters all `target` and `onload` attributes from tags (because there can be JavaScript). The filter is applied only once, however, so the `onload` attribute with the worm code stays in place. This is a good example why blacklist filters are never complete and why it is hard to allow HTML/JavaScript in a web application.

Another proof-of-concept webmail worm is Nduja, a cross-domain worm for four Italian webmail services. Find more details on [Rosario Valotta's paper](#). Both webmail worms have the goal to harvest email addresses, something a criminal hacker could make money with.

In December 2006, 34,000 actual user names and passwords were stolen in a [MySpace phishing attack](#). The idea of the attack was to create a profile page named "login\_home\_index\_html", so the URL looked very convincing. Specially-crafted HTML and CSS was used to hide the genuine MySpace content from the page and instead display its own login form.

The MySpace Samy worm will be discussed in the CSS Injection section.

## 7.4 CSS Injection

Info:

CSS Injection is explained best by a well-known worm, the [MySpace Samy worm](#). This worm automatically sent a friend request to Samy (the attacker) simply by visiting his profile. Within several hours he had over 1 million friend requests, but it creates too much traffic on MySpace, so that the site goes offline. The following is a technical explanation of the worm.

MySpace blocks many tags, however it allows CSS. So the worm's author put JavaScript into CSS like this:

```
<div style="background:url('javascript:alert(1)')">
```

So the payload is in the style attribute. But there are no quotes allowed in the payload, because single and double quotes have already been used. But JavaScript has a handy `eval()` function which executes any string as code.

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(docu
```

The `eval()` function is a nightmare for blacklist input filters, as it allows the style attribute to hide the word "innerHTML":

```
alert(eval('document.body.inne' + 'rHTML'));
```

The next problem was MySpace filtering the word "javascript", so the author used "j~~ava~~script" to get around this:

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(do
```

Another problem for the worm's author were CSRF security tokens. Without them he couldn't send a friend request over POST. He got around it by sending a GET to the page right before adding a user and parsing the result for the CSRF token.

In the end, he got a 4 KB worm, which he injected into his profile page.

The [moz-binding](#) CSS property proved to be another way to introduce JavaScript in CSS in Gecko-based browsers (Firefox, for example).

## 7.4.1 Countermeasures

This example, again, showed that a blacklist filter is never complete. However, as custom CSS in web applications is a quite rare feature, it may be hard to find a good whitelist CSS filter. . Use Rails' `sanitize()` method as a model for a whitelist CSS filter, if you really need one.

## 7.5 Textile Injection

If you want to provide text formatting other than HTML (due to security), use a mark-up language which is converted to HTML on the server-side. [RedCloth](#) is such a language for Ruby, but without precautions, it is also vulnerable to XSS.

For example, RedCloth translates `_test_` to *test*, which makes the text italic. However, up to the current version 3.0.4, it is still vulnerable to XSS. Get the [all-new version 4](#) that removed serious bugs. However, even that version has [some security bugs](#), so the countermeasures still apply.

*Here is an example for version 3.0.4:*

```
RedCloth.new('<script>alert(1)</script>').to_html
# => "<script>alert(1)</script>"
```

*Use the :filter\_html option to remove HTML which was not created by the Textile processor.*

```
RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
# => "alert(1)"
```

*However, this does not filter all HTML, a few tags will be left (by design), for example :*

```
RedCloth.new("<a href='javascript:alert(1) '>hello</a>", [:filter_html]).to_html
# => "<p><a href="javascript:alert(1) ">hello</a></p>"
```

## 7.5.1 Countermeasures

*It is recommended to , as described in the countermeasures against XSS section.*

## 7.6 Ajax Injection

*Note:*

*If you use the [in\\_place\\_editor plugin](#), or actions that return a string, rather than rendering a view, . Otherwise, if the return value contains a XSS string, the malicious code will be executed upon return to the browser. Escape any input value using the h() method.*

## 7.7 Command Line Injection

*Note:*

*If your application has to execute commands in the underlying operating system, there are several methods in Ruby: exec(command), syscall(command), system(command) and command. You will have to be especially careful with these functions if the user may enter the whole command, or a part of it. This is because in most shells, you can execute another command at the end of the first one, concatenating them with a semicolon (;) or a vertical bar (|).*

*A countermeasure is to .*

```
system("/bin/echo","hello; rm *")
# prints "hello; rm *" and does not delete files
```

## 7.8 Header Injection

*Warning:*

*HTTP request headers have a Referer, User-Agent (client software), and Cookie field, among others. Response headers for example have a status code, Cookie and Location (redirection target URL) field. All of them are user-supplied and may be manipulated with more or less effort. For example when you display the user agent in an administration area.*

*Besides that, it is For example you want to redirect the user back to a specific page. To do that you introduced a "referer" field in a form to redirect to the given address:*

```
redirect_to params[:referer]
```

*What happens is that Rails puts the string into the Location header field and sends a 302 (redirect) status to the browser. The first thing a malicious user would do, is this:*

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
```

*And due to a bug in (Ruby and) Rails up to version 2.1.2 (excluding it), a hacker may inject arbitrary header fields; for example like this:*

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
http://www.yourapplication.com/controller/action?referer=path/at/your/app%0d%0aL
```

*Note that "%0d%0a" is URL-encoded for "" which is a carriage-return and line-feed (CRLF) in Ruby. So the resulting HTTP header for the second example will be the following because the second Location header field overwrites the first.*

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

*So And what could an attacker do with a false redirection? They could redirect to a phishing site that looks the same as yours, but ask to login again (and sends the login credentials to the attacker). Or they could install malicious software through browser security holes on that site. Rails 2.1.2 escapes these characters for the Location field in the redirect\_to method.*

## **7.8.1 Response Splitting**

*If Header Injection was possible, Response Splitting might be, too. In HTTP, the header block is followed by two CRLFs and the actual data (usually HTML). The idea of Response Splitting is to inject two CRLFs into a header field, followed by another response with malicious HTML. The response will be:*

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location: Content-Type: text/html
```

```
HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html
```

```
<html><font color=red>hey</font></html> [Arbitrary malicious HTML]
Keep-Alive: timeout=15, max=100 shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

*Under certain circumstances this would present the malicious HTML to the victim. However, this only seems to work with Keep-Alive connections (and many browsers are using one-time connections). But you can't rely on this.*

# 8 Unsafe Query Generation

Due to the way Active Record interprets parameters in combination with the way that Rack parses query parameters it was possible to issue unexpected database queries with `IS NULL` where clauses. As a response to that security issue ([CVE-2012-2660](#), [CVE-2012-2694](#) and [CVE-2013-0155](#)) `deep_munge` method was introduced as a solution to keep Rails secure by default.

Example of vulnerable code that could be used by attacker, if `deep_munge` wasn't performed is:

```
unless params[:token].nil?  
  user = User.find_by_token(params[:token])  
  user.reset_password!  
end
```

When `params[:token]` is one of: `[], [nil], [nil, nil, ...]` or `['foo', nil]` it will bypass the test for `nil`, but `IS NULL` or `IN ('foo', NULL)` where clauses still will be added to the SQL query.

To keep rails secure by default, `deep_munge` replaces some of the values with `nil`. Below table shows what the parameters look like based on `JSON` sent in request:

JSON	Parameters
<code>{ "person": null }</code>	<code>{ :person =&gt; nil }</code>
<code>{ "person": [] }</code>	<code>{ :person =&gt; nil }</code>
<code>{ "person": [null] }</code>	<code>{ :person =&gt; nil }</code>
<code>{ "person": [null, null, ...] }</code>	<code>{ :person =&gt; nil }</code>
<code>{ "person": ["foo", null] }</code>	<code>{ :person =&gt; ["foo"] }</code>

It is possible to return to old behaviour and disable `deep_munge` configuring your application if you are aware of the risk and know how to handle it:

```
config.action_dispatch.perform_deep_munge = false
```

# 9 Default Headers

Every HTTP response from your Rails application receives the following default security headers.

```
config.action_dispatch.default_headers = {  
  'X-Frame-Options' => 'SAMEORIGIN',  
  'X-XSS-Protection' => '1; mode=block',  
  'X-Content-Type-Options' => 'nosniff'  
}
```

You can configure default headers in `config/application.rb`.

```
config.action_dispatch.default_headers = {  
  'Header-Name' => 'Header-Value',  
  'X-Frame-Options' => 'DENY'  
}
```

Or you can remove them.

```
config.action_dispatch.default_headers.clear
```

Here is a list of common headers:

- X-Frame-Options
- allow framing on same domain. Set it to 'DENY' to deny framing at all or 'ALLOWALL' if you want to allow framing for all website.
- X-XSS-Protection
- use XSS Auditor and block page if XSS attack is detected. Set it to '0;' if you want to switch XSS Auditor off(useful if response contents scripts from request parameters)
- X-Content-Type-Options
- stops the browser from guessing the MIME type of a file.
- X-Content-Security-Policy [A powerful mechanism for controlling which sites certain content types can be loaded from](#)
- Access-Control-Allow-Origin Used to control which sites are allowed to bypass same origin policies and send cross-origin requests.
- Strict-Transport-Security [Used to control if the browser is allowed to only access a site over a secure connection](#)

# 10 Environmental Security

It is beyond the scope of this guide to inform you on how to secure your application code and environments. However, please secure your database configuration, e.g. `config/database.yml`, and your server-side secret, e.g. stored in `config/secrets.yml`. You may want to further restrict access, using environment-specific versions of these files and any others that may contain sensitive information.



# 11 Additional Resources

The security landscape shifts and it is important to keep up to date, because missing a new vulnerability can be catastrophic. You can find additional resources about (Rails) security here:

- The Ruby on Rails security project posts security news regularly: <http://www.rorsecurity.info>
- Subscribe to the Rails security [mailing list](#)
- [Keep up to date on the other application layers](#) (they have a weekly newsletter, too)
- A [good security blog](#) including the [Cross-Site scripting Cheat Sheet](#)

# Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our [documentation contributions](#) section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check [Edge Guides](#) first to verify if the issues are already fixed or not on the master branch. Check the [Ruby on Rails Guides Guidelines](#) for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please [open an issue](#).

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the [rubyonrails-docs mailing list](#).