Ruby on Rails Secure Coding Recommendations

Introduction

Altius IT's list of Ruby on Rails Secure Coding Recommendations is based upon security best practices. This list may not be complete and Altius IT recommends this list be augmented with additional controls. While not all risk can be eliminated, secure coding practices can include preventive, detective, and corrective safeguards that help reduce risks to acceptable levels.

Ensure the appropriate use of attr_accessible/attr-protected:

- Use attr_accessible in every class, thereby defining what you have access to in the model. Don't pass user data to .new or .create.
- The attr_protected method in Ruby on Rails will prevent the fields from being assigned via mass assignment.
- Don't use mass assignments for user tables. When a user account is created, assign each field individually. One additional way to handle mass-assignment is through strong_parameters. The strong_parameters gem is an improvement over attr_accessible to securely handle mass assignment even when you have complex authorization logic. It is also a whitelist-based approach, but moves the responsibility to the controller making it easier to authorize what is and isn't allowed based on the current user's role.

Use white-list filtering as early as possible. White lists describe what is allowed, rather than what is not allowed.

Using a plugin does not mean that the code is mature, stable, or secure.

- Popular plugins may be vulnerable to security attacks (e.g. restful_authentication) and need to be augmented with secure coding practices.
- Ensure applications (Gems) are up-to-date with latest security patches. Subscribe to Ruby on Rails security mailing list (https://groups.google.com/forum/?fromgroups#!forum/rubyonrails-security).
- Always perform a code review of new plugins for issues.
- Track plugin announcements.
- Consider tracking external sources with Piston (http://piston.rubyforge.org), a wrapper around svn:externals.

Prevent application leaks by overriding defaults. Hackers can check for default configurations to find out more about the target:

- Server header - When Apache HTTPD web server generates web pages or error pages, important information about the version and other details implemented on the system are displayed in the web site server header. Disable the server header.
- Determine if the target is a Rails application – By default, static files include /javascripts/application.js, /stylesheets/application.css, /images/foo.png. Rails

default templates for 404 (404.html, 422.html) and 500 (500.html) status pages. Rails applications deployed with Capistrano / Webistrano push .svn directories to the severs (http://svn.altiusit.com). Do not use svn:externals to track external plugins. If the plugins home page is unavailable, you may not be able to deploy your site.

- Ensure error messages are generated by the application, not the server. Error messages should be short and give user information without providing details to a hacker.
- Where appropriate, filter certain parameters from being shown in plain text in the log file by using filter_parameter_logging.

Protect against Cross Site Request Forgery (CSRF). Use the protect_from_forgery feature in Rails that protects against Cross-site Request Forgery (CSRF) attacks. This feature makes all generated forms have a hidden id field. This id field must match the stored id or the form submission is not accepted. This prevents malicious forms on other sites or forms inserted with XSS from submitting to the Rails application.

Protect against Cross Site Scripting (XSS). Sanitize input when appropriate (e.g. search queries, user name, description).

- Use the Rails 'h()' helper to HTML escape user input. Using 'h()' may be difficult to use everywhere. Consider using safeERB (http://agilewebdevelopment.com/plugins/safe_erb) plugin. SafeERB will raise an exception whenever a tainted string is not escaped. If formatting is allowed, consider using custom tags that will translate to HTML.
- Use the Rails 'sanitize()' helper. Sanitize filters HTML nodes and attributes, removes protocols (e.g. JavaScript), and handles unicode/ascii/hex attacks. For more information see (http://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html).
- Use XSSshield.

Protect against SQL injection attacks. Consider all input to be malicious. Sanitize/escape input using procedures identified in this document. If the code must manually use a user-submitted value, use 'quote()'. Use bind variables and an array for SQL queries that use .find.

Prevent against JavaScript hijacking. Use the following coding techniques to minimize JavaScript hijacking attacks:

- Do not store important data in JavaScript Object Notation (JSON) responses.
- Do not return a straight JSON response, prefix it with extraneous characters. Rails JavaScript helpers may not support pre-fixed JSON responses.
- Use hard to guess URLs.

Ruby on Rails Secure Coding Recommendations

Protect against Denial of Service (DoS) attacks. Rails applications are singled threaded with a limited number of Rails instances. Rails is particularly vulnerable to DoS attacks during file uploads/downloads, image processing, report generation, mass mailing, etc. Minimize DoS attacks by reducing these types of activities and maximizing additional resources such as serving static files directly through the web server (e.g. Apache, Amazon S3).

By using RESTful (Representational State Transfer) resources, the interface to database objects is clear and consistent throughout the application and follows clear rules. Consider popular authentication plugins such as restful_authentication and authlogic.

Cookie sessions. By default, session data is stored in a cookie. This presents several security implications including user viewing session data in plain text, the HMAC can be brute-forced and arbirtray session data created, replay attacks are easier as the client-side session cannot be flushed. Protect against cookie vulnerabilities:

- Do not store important data in the session. Sessions are stored by default in a browser cookie that's cryptographically signed, but unencrypted. This prevents the user from tampering with the session while allowing the contents to be viewed.
- Ensure the use of strong passwords.
- Servers should timeout inactive sessions.
- Ensure session cookies are not persistent (expires-at).

File uploads. All file uploads should be analyzed for possible malware. Files should be saved outside of DocumentRoot. Consider using the hash of a file name instead of the real file name or store the file name in a database and use an ID number. Deny the direct download of files. Deny file name suffixes such as rb, cgi, php, erb, html, htm.

Ensure secure authentication. Ensure the application enforces an acceptable password policy for users:

- Use reset_session at login.
- Ensure the authentication process cannot by bypassed.
- Ensure credentials are encrypted with SSL.
- Salt passwords. Storing a hash of the passwords isn't enough as it would still leave the application vulnerable attack via rainbow tables; where an attacker pre-calculates hashes of millions of passwords, then compares the hashes with the values stolen from your database. To prevent this you need to salt them. This requires storing a small random value against each of your users and adding that to the password before it is hashed.
- Bind the IP address to the session ID.

- Reauthenticate the user before performing security relevant actions such as changing a password.

Limit the use of very dangerous methods/functions including system(), popen(), `` (backticks), eval(), and deserialize().

Limit administrative interfaces to a sub-net or localhost.

Consider CAPTHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) to reduce risks related to scripted attacks. A CAPTCHA is a type of challenge-response test used to ensure that the response is generated by a person.

Ensure sensitive information is protected:

- Use appropriate encryption.
- Keep sensitive data in memory only as long as required.
- Create audit records when sensitive information is created, accessed, modified, or deleted.
- Avoid leaking by not logging sensitive data itself.

Refer to Altius IT's Recommended Software Security Controls for additional safeguards and security best practices.

Resources

Security toolboxes can help developers ensure code is secure. Altius IT provides the following resources as a client convenience. We recommend that the resources be reviewed for applicability and appropriateness in your specific environment.

ActiveRbac (http://active-rbac.rubyforge.org) – a Ruby On Rails plugin that supports Role Based Access Control (RBAC) in the application.

Brakeman (http://www.brakemanscanner.org) - an open source vulnerability scanner specifically designed for Ruby on Rails applications. It statically analyzes Ruby on Rails code to find potential security vulnerabilities.

- Command injection
- Cross site scripting
- Dangerous use of eval() Default routes
- Insufficient model validation
- SQL injection
- Unprotected redirects
- Unrestricted mass assignment
- Unsafe file access
- Version-specific security issues

Rails testing (http://willbryant.net/software/rails/application_testing_top_to_bottom/) – testing contributes to the success of software development efforts.

- What we want to test
- MVC testing
- Model testing
- Controller testing

Ruby on Rails Security Blog (http://www.rorsecurity.info/) – blog posts and information on security related topics.

Ruby on Rails Security Guide (http://guides.rubyonrails.org/security.html) – this guide describes common security problems in web applications and how to avoid them with Ruby on Rails.

The Ruby Toolbox (https://www.ruby-toolbox.com/categories/security_tools) – includes links to plugins including:

- Tarantula – crawls Ruby on Rails applications using fuzzing data to determine if errors are encountered
- Rails xss – replaces the default ERB template handlers with erubis, and escapes by default rather than requiring programmer to escape.
- Find mass assignment – finds mass assignment vulnerabilities
- Audit mass assignment – checks for use of the attr_accessible white list

Xss_terminate (http://github.com/look/xss_terminate) – strips and sanitizes HTML code.

Publication Information

Altius IT is a security audit, security consulting, and risk management firm. We are certified by the Information Systems Audit and Control Association (ISACA) as a Certified Information Systems Auditor (CISA), Certified in Risk and Information Systems Controls (CRISC), and Certified in the Governance of Enterprise Wide IT (CGEIT). For more information, please visit www.AltiusIT.com.