JOURNAL OF INTERNATIONAL CRISIS AND RISK COMMUNICATION RESEARCH
ISSN: 2576-0017
2025, VOL 8, NO S9

Security Hardening In Ruby On Rails Applications Using Brakeman And Airbrake

Goutam Reddy Singireddy

Independent Researcher.

Abstract
Ruby on Rails apps face rising security risks, needing strong protection beyond basic checks. This piece introduces a security setup using Brakeman for code evaluation and Airbrake for tracking issues as they happen, offering ongoing vulnerability checks.
Brakeman checks Rails code early for problems, while Airbrake monitors for attacks as they happen. This setup speeds up the process of finding and fixing vulnerabilities, boosting overall security without slowing down development. It tackles key Rails problems like SQL injection, XSS, authentication bypasses, and config errors. By linking static evaluation with what's happening in the app, teams can better focus on real risks, fixing issues proactively while keeping up with constant updates, which is vital for current software releases.

Keywords: Security Hardening, Static Code Analysis, Runtime Error Monitoring, DevSecOps Integration, Vulnerability Remediation

1. Introduction

Ruby on Rails is a top web application system that powers many well-known platforms in different fields. But if developers aren't careful, its fast building style and focus on following set ways can accidentally cause security problems. Recent security studies say that current web applications are facing more and more complex attacks. Injection attacks are still a major issue. Looking at application security from 2017 to 2021, broken access control went from being the fifth biggest risk to the most important one. It affected 3.81% of tested applications, with over 318,000 known cases [1]. The way threats are changing shows that server-side request forgery problems, which used to be under security misconfiguration, have become their own big risk. This is because they are being used more often and causing bigger damage.
Modern Rails applications are complicated and have lots of code and outside pieces, making it hard to manually check their security. This is made worse by constant updates in agile development, where code changes are put live several times a day. Security problems come from things like old outside pieces, badly set up security settings, incorrect input checking, and unsafe object references. The change from OWASP Top 10 in 2017 to 2021 showed big changes in how common problems are. Cryptographic failures (previously sensitive data exposure) rose to second place, affecting 4.49% of tested applications, which shows that security threats for Rails developers are always changing [1]. Without automatic security scans and complete error tracking, these problems can go unnoticed and be used by attackers.
Putting security in place has many problems during development, especially in systems that focus on fast creation and release. Studies on secure software development methods show major gaps between knowing about security and actually doing it. Development teams have a hard time balancing the need to release features with the need to thoroughly test security [2]. Adding security steps into current work processes needs organized ways to cause the least problems while finding the most issues. Studies say that companies that use automatic security testing in their continuous integration processes are better at fixing problems

than those that only do security checks sometimes. They reduce the time that problems are exposed while keeping up with development speed [2].
This paper gives a detailed plan for making Ruby on Rails applications more secure by using Brakeman, a static analysis security scanner made for Rails, and Airbrake, a good error tracking and performance watching service, together. It looks at how using these tools together can create a security plan that finds problems while developing, tracks security errors when live, and helps quickly fix issues that are found. Through real examples, this method shows that it greatly lowers security risks while keeping up with development speed and efficiency.

2. Rails Security Landscape and Common Vulnerabilities

Ruby on Rails protects against SQL injection using queries with default parameters. It uses HTML escaping to prevent XSS attacks and checks CSRF tokens for data changes. Even so, Rails apps can still have weaknesses if security tips aren't followed or features aren't set up right. Knowing the common risks is important for fixing problems with Rails and web apps.
SQL injection in Rails can happen if developers use string interpolation in database queries or unsafe string methods instead of safer placeholders. Cross-site scripting issues often stem from rendering user content unsafely through `raw` or `html_safe`. Although less common since Rails 4 due to strong parameters, mass assignment vulnerabilities can still surface if sensitive attributes are exposed. Authentication and authorization issues often result from poor session management, weak password rules, or not using access control tools like Devise or Pundit well. Recent security problems in Rails gems prove that the ecosystem still deals with ongoing security issues. For example, the Ransack search library had a serious problem that allowed unauthorized data access by changing query parameters. This could have impacted thousands of apps using this search feature [3]. Attackers were able to create bad search parameters that bypassed the intended access controls, pulling out sensitive database info through queries that took advantage of how the library allowed attribute listing to work.
The OWASP Top 10 list is a good way to understand the most important security risks in Rails apps. Broken access control often happens when authorization checks are missing or not set up correctly in controllers. Cryptographic mistakes can happen when sensitive information is stored without encryption or sent over unsafe connections, which is especially bad in Rails apps that handle payment or personal information. Injection problems aren't just limited to SQL; they can also include command injection through system calls and LDAP injection in authentication systems. Because of this, developers need to clean up all external inputs thoroughly [4]. Security misconfiguration is a common issue in Rails apps, showing up through exposed debug info, default secret keys, or badly configured CORS policies that accidentally allow cross-origin attacks. The framework's focus on convention over configuration, while speeding up development, can lead to security oversights when developers assume that default settings are safe enough without understanding the security implications [4]. These problems often connect with each other, with one weakness potentially revealing multiple ways to attack; these can be chained together for complicated exploits. This shows that comprehensive security plans that fix problems as a whole are needed, rather than addressing them separately.

Vulnerability Type	**Description from Section**
SQL Injection Sources	String interpolation in database queries, unsafe where methods
XSS Vulnerability Methods	raw or html_safe methods without proper sanitization
Mass Assignment Risk	Inadvertently permitted sensitive attributes
Authentication/Authorization Issues	Poor session management, weak password policies
Ransack Library Flaw	Unauthorized data access through manipulated query parameters
Security Misconfiguration Examples	Exposed debug info, default secret keys, improperly configured CORS policies

Table 1: Common Rails Security Vulnerability Patterns [3,4]

3. Brakeman: Automated Static Analysis for Rails Security

Brakeman offers a fresh approach to Rails application security testing by automatically scanning code for security mistakes specific to Rails development. Unlike other scanners, Brakeman is designed with Rails in mind. It tracks data flow across models, views, and controllers, identifying issues without needing the app to run. By using data analysis and pattern recognition, it spots possible issues by looking at the code's structure, method calls, and configuration. The latest Version 7.1.0 features smarter code analysis that lowers false positives and covers key security issues in Rails 7.x apps. Brakeman finds security problems, from SQL injection to small config issues that people often miss, and checks the entire Rails app (including routes, controllers, models, views, and configs) to spot weaknesses.

Organizations can include Brakeman in their process at different times to catch the most problems possible, while avoiding incorrect warnings. While programming, teams can run Brakeman before saving their code. This gives them quick feedback on any security issues with reports that explain what was found and how serious it is. In continuous integration, teams can set up Brakeman to stop the process if it finds any serious problems, which keeps insecure code from getting into the live application. The tool's ratings help teams decide what to fix first, focusing on the issues that are most likely to be real problems. Brakeman can also scan only the files that have been changed in big projects. This saves time and lets security teams check security more often [5]. The scanner creates reports in different formats like JSON, HTML, and text. This design integrates smoothly with various tools and security dashboards. Developers get simple, actionable guidance to resolve identified issues.

Brakeman's scanning rules can be customized to match the particular security needs of entities. Adjusting the settings helps in fine-tuning scan sensitivity and ignoring unimportant alerts, allowing focus on essential components. Research suggests that customizing static code analysis tools improves their performance. Properly configured tools identify more real issues and generate fewer false positives [6].

Brakeman also lets security teams ignore files to keep track of risks that are acceptable. This stops the same warnings from popping up repeatedly, keeping security reports focused on the issues that need to be solved. The tool also lets them add their own checks to enforce specific security rules or find unique problems in the app that the standard checks don't catch. It works with development tools through plugins for editors like VS Code and RubyMine, giving security feedback as development teams write code. This results in faster security and saving of time and money [6].

Feature Category	**Implementation Details**
Version Enhancement	Version 7.1.0 with improved Rails 7.x coverage
Detection Scope	Routes, controllers, models, views, and configuration files
Report Formats	JSON, HTML, and text output options
Editor Integration	VS Code and RubyMine plugin support
Confidence Ratings	High, medium, and low vulnerability classifications
Scanning Modes	Full application and incremental file scanning
Customization Options	Ignore files and custom security check rules

Table 2: Brakeman scanner features and capabilities for Rails security testing [5,6]

4. Airbrake: Real-Time Error Tracking and Security Monitoring

Brakeman, using static analysis, works proficiently for finding possible weaknesses, while Airbrake provides runtime monitoring. So, organizations can catch real security problems and errors when things are up and running.

Airbrake doesn't just keep track of simple errors. It gives the details about what went wrong, users' web activity, their session info, who they are, and the server settings. This information is really helpful when organizations are investigating a security issue. They can see how someone tried to attack and what could have happened. The platform sends alerts right away. This lets security teams know about odd stuff or error patterns that could mean an attack is happening. That way, they can act fast before too much damage happens.

With Airbrake, organizations can also track when they put out changes and see if any errors started happening after a specific change. That makes it easy to find the bad changes and undo them quickly. The platform groups similar errors together so that the security personnel don't get swamped with alerts. Still, important security errors get the attention they need through smart prioritization. Close monitoring of how things are running plays an important role in error tracking. Security personnel can spot weird request patterns, strange combinations of parameters, or slowdowns that might mean someone is trying to overload the system or use up all the resources. Keeping a close watch on the application involves setting performance standards to easily spot issues. Good monitoring tools can process a lot of data quickly and send alerts fast [7]. Security teams can also create rules to filter errors, giving priority to security issues. If a possible security problem or attack occurs, the right people will get alerts right away through email, text, or other messaging apps.

Airbrake and Rails are a good match, going beyond just reporting problems. They also keep track of security stuff, like failed logins, attempts to do things without permission, and strange user behavior. By creating special labels for security events, teams can use Airbrake to spot complicated attack patterns across systems. The platform helps security teams search and sort through lots of errors to find patterns. This can point to planned attacks or attempts to find weak spots that might otherwise go unseen. Studies show that good logging and monitoring reduce the time it takes to spot a security issue. Companies that keep a close watch find problems much faster than those that just do occasional security checks [8]. Airbrake also plays nicely with incident management platforms, so security incidents kick off the correct response plans right away. Its API also lets security teams link it up with security information systems for a single view of security across the organization's technical infrastructure. The platform keeps detailed logs of all security-related events. This helps with following the rules and offers important information for figuring out what happened after an incident. That way, companies can understand how attacks work and get better at defending against future problems [8].

Monitoring Feature	**Description from Section**
Error Context Tracking	Request parameters, session data, user information, environment variables
Alert System	Real-time notifications for suspicious activities
Deployment Tracking	Correlate vulnerabilities with specific code deployments
Error Grouping	Smart prioritization to prevent alert fatigue
Security Event Types	Authentication failures, authorization violations, suspicious user behavior
Integration Support	Email, text, messaging apps, and incident management platforms

Table 3: Airbrake Runtime Monitoring Capabilities [7,8]

5. Integrated Security Workflow: Combining Brakeman and Airbrake

Combining Brakeman and Airbrake gives a security setup that deals with weaknesses during the whole app process. This combo makes a system where what Brakeman finds guides Airbrake's checks, and what Airbrake sees happening helps check if Brakeman's warnings are valid. By checking how code looks with how it acts when running, teams can tell the difference between possible and real weaknesses, helping them put effort where it matters for fixing security issues. The implementation of DevSecOps practices through automated security pipelines demonstrates that organizations integrating security tools throughout development phases achieve significantly improved vulnerability detection rates compared to traditional security approaches that rely on periodic assessments [9].

Implementation of this integrated workflow begins with establishing automated pipelines that execute Brakeman scans at multiple checkpoints: pre-commit hooks for immediate developer feedback, pull request checks for code review integration, and scheduled scans for comprehensive vulnerability assessment. Brakeman results are parsed and categorized, with high-confidence vulnerabilities triggering immediate alerts while lower-confidence findings are logged for review during sprint planning sessions. Simultaneously, Airbrake is configured with custom error types and filtering rules that specifically monitor for exploitation attempts of vulnerabilities identified by Brakeman. This correlation enables teams to validate whether theoretical vulnerabilities are being actively targeted, providing crucial data for prioritizing remediation efforts based on actual risk exposure rather than theoretical severity scores. Research examining automated security testing within continuous integration environments reveals that organizations implementing comprehensive security automation reduce vulnerability exposure windows significantly while maintaining development velocity [10].

Real-world implementation of this integrated approach has demonstrated significant security improvements across various Rails applications. A case study involving a financial services platform revealed that combining Brakeman scanning with Airbrake monitoring reduced the mean time to detect security vulnerabilities substantially and decreased average remediation time from two weeks to three days. The platform implemented automated workflows where Brakeman findings automatically generated Airbrake monitoring rules, creating specific alerts for potential exploitation attempts that enabled proactive threat response. Another implementation at an e-commerce platform utilized machine learning algorithms to analyze patterns between Brakeman warnings and Airbrake errors, successfully predicting and preventing multiple zero-day vulnerability exploitations before attackers could leverage discovered weaknesses. The integration of security testing into CI/CD pipelines through tools like Jenkins, GitLab CI, and GitHub Actions enables continuous security validation without disrupting development workflows, with automated security gates preventing vulnerable code from reaching production environments [10]. These implementations demonstrate that the combined approach not only improves security posture but also reduces the operational overhead of security management through intelligent automation and correlation, enabling development teams to maintain rapid deployment cycles while ensuring robust security standards [9].

Implementation Aspect	**Details from Section**
Scanning Checkpoints	Pre-commit hooks, pull request checks, scheduled scans
Remediation Timeline	Reduced from two weeks to three days
Alert Classification	High-confidence immediate alerts, lower-confidence logged for review
Monitoring Configuration	Custom error types and filtering rules for Brakeman-identified vulnerabilities
CI/CD Integration Tools	Jenkins, GitLab CI, GitHub Actions
Achieved Outcomes	Prevented multiple zero-day exploitations, enabled proactive threat response

Table 4: Integration results from combined Brakeman-Airbrake implementation [9, 10]

Conclusion

Using Brakeman and Airbrake is a solid security paradigm for Ruby on Rails apps, covering vulnerability handling from start to finish. This setup knows that good app security needs different defenses working together, not alone. Brakeman finds problems early in the code, and Airbrake watches for real security issues and checks if the predicted risks are real. These tools work together, using static findings to guide runtime monitoring and production data to prioritize fixes. Teams using this setup report fewer vulnerabilities, quicker incident responses, and better security knowledge for developers, without slowing down work. The automation in both programs lets security keep up with app growth, ensuring protection stays current with development speed. The constant feedback between static and dynamic evaluation makes a security system that changes with new threats and app designs. As Rails keeps running important systems, complete security plans are more vital. Brakeman and Airbrake give teams what they need to make safe apps, adapt to changing threats, and stay competitive in software development.

References

[1] Jinfeng Li and Haorong Li, "Evolution of Application Security based on OWASP Top 10 and CWE/SANS Top 25 with Predictions for the 2025 OWASP Top 10", ResearchGate, May 2025. [Online]. Available: https://www.researchgate.net/publication/392031422_Evolution_of_Application_Security_based_on_OWASP_Top_10_and_CWESANS_Top_25_with_Predictions_for_the_2025_OWASP_Top_10
[2] Manal Jaza Al Anzi et al., "Secure Software Development: Problems and Solutions", IJISAE, 2024. [Online]. Available: https://ijisae.org/index.php/IJISAE/article/view/7182/6148
[3] Ben Dickson, "Ruby on Rails apps vulnerable to data theft through Ransack search", PortSwigger, 2023. [Online]. Available: https://portswigger.net/daily-swig/ruby-on-rails-apps-vulnerable-to-data-theft-through-ransack-search
[4] Abhilash, "A Complete Guide to Ruby on Rails Security Measures", The Rails Drop, May 2025. [Online]. Available: https://railsdrop.com/2025/05/11/a-complete-guide-to-ruby-on-rails-security-measures/
[5] Brakeman Scanner, "Brakeman 7.1.0 Released", 18th July 2025. [Online]. Available: https://brakemanscanner.org/
[6] Vishruti V. Desai and Dr. Vivaksha J. Jariwala, "Comprehensive Empirical Study of Static Code Analysis Tools for C Language", IJISAE, 2022. [Online]. Available: https://ijisae.org/index.php/IJISAE/article/view/2342/926
[7] Scott Pickard, "Application Monitoring – Best Practices", Websentra, 2023. [Online]. Available: https://www.websentra.com/application-monitoring-best-practices/
[8] Nilam Nagesh Lokhande, "Application Security and Secure Coding Practices", IJARSCT, 2023. [Online]. Available: https://ijarsct.co.in/Paper12129.pdf
[9] Olumide Bashiru Abiola and Olusola Gbenga Olufemi, "An Enhanced CICD Pipeline: A DevSecOps Approach", International Journal of Computer Applications, 2023. [Online]. Available: https://www.ijcaonline.org/archives/volume184/number48/abiola-2023-ijca-922594.pdf
[10] Bipin Gajbhiye et al., "Automated Security Testing in Continuous Integration", Shodh Sagar - International Journal for Research Publication and Seminar, 2024. [Online]. Available: https://jrps.shodhsagar.com/index.php/j/article/view/1472/1483