Code Audit on Ruby on Rails
for the Open Source Technology Improvement Fund

Final Report (Not for Release)

2025-06-11

PUBLIC

X41 D-Sec GmbH
Soerser Weg 20
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

In cooperation with GitLab Inc.

Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2024-12-23	Initial Threat Model and Test Plan	E. Sesterhenn, J. Schneeweisz, R. Femmer, M. Vervier
2	2025-03-25	Final Report and Management Summary	E. Sesterhenn, J. Schneeweisz, R. Femmer, M. Vervier
3	2025-06-11	Public Report Release	E. Sesterhenn, J. Schneeweisz, R. Femmer, M. Vervier

Contents

Dashboard

Target	
Customer	Open Source Technology Improvement Fund
Name	Ruby on Rails
Type	Web Framework
Version	v8.0.1
Engagement	
Type	Gray Box Penetration Test
Consultants	5: Eric Sesterhenn, Joern Schneeweisz, J.M., Markus Vervier, and Robert Femmer
Engagement Effort	50 person-days, 2024-12-09 to 2025-03-25
Total issues found	7

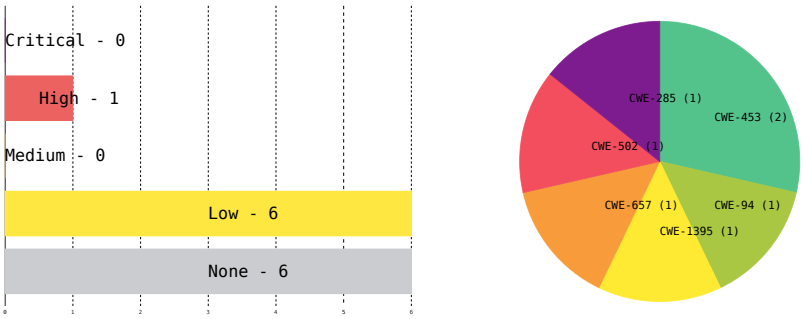

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

The security review of Ruby on Rails v8.0.1 performed by X41 between December 2024 and March 2025 has identified several areas where improvements can be made to ensure robust security. The test was organized by the Open Source Technology -Improvement Fund[1]. GitLab[2] directly supported the assessment by sponsoring participation of the GitLab Security Research Team[3] in the audit.

A total of seven vulnerabilities were discovered during the test by X41. None were rated as having a critical severity, one as high, none as medium, and six as low. Additionally, six issues without a direct security impact were identified.

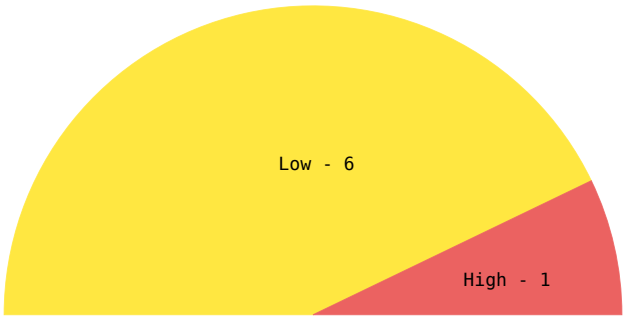

Figure 1.1: Issues and Severity

[1] https://ostif.org
[2] https://about.gitlab.com
[3] https://handbook.gitlab.com/handbook/security/product-security/security-platforms-architecture/security-research/

The test was performed by five experienced security experts between 2024-12-09 and 2025-03-25.

During the security code review, various types of attacks were investigated, including authentication bypasses, authorization flaws, cryptographic issues, information exposure, privilege escalation, client-side injections, SSRF, race conditions, insecure file operations, and concurrency issues.

It shows that over the recent years, the maturity of the Rails code base has grown significantly in regard to security. Issues identified include the lack of comprehensive deauthentication mechanisms in ActionCable, potential parsing vulnerabilities for different formats like JSON and HTTP, and logical bugs in ActiveStorage that could lead to dangerous file accesses. Additionally, the review highlights third-party dependency handling as an area requiring further investigation due to its critical role in security.

To address these findings, X41 recommends performing follow-up tests such as code audits of components used (like the mail library) and assessing popular Gems within the Ruby on Rails ecosystem for potential vulnerabilities that could impact multiple deployments.

The security audit has identified several areas of improvement to enhance the security posture of Ruby on Rails, focusing on components such as ActionMailer, ActionMailbox, and ActiveStorage. The recommendations provided will assist in mitigating identified vulnerabilities and ensuring robust security practices are implemented.

2 Introduction

The review of Ruby on Rails by X41 aims to identify security vulnerabilities in the source code via a manual review.

The threat model described in the chapter 3 forms the basis of this review and the ratings of identified issues. It is based on the assumptions of typical attackers and their capabilities in common usage scenarios.

2.1 Methodology

The review is conducted as a manual code review. X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[1] standards and the *Study - A Penetration Testing Model*[2] of the German Federal Office for Information Security.

The workflow of source code reviews is shown in figure 2.1. In an initial, informal workshop regarding the design and architecture of the application, a basic threat model is created. This is used to explore the source code for interesting attack surfaces and code paths. These are then audited manually and with the help of tools such as static analyzers and fuzzers. The identified issues are documented and can be used in a GAP analysis to highlight changes to previous audits.

[1] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

[2] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

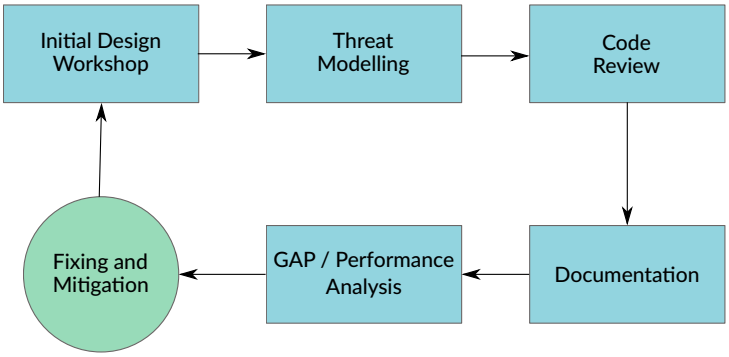

Figure 2.1: Code Review Methodology

2.2 Scope

The source code in scope for this audit was Ruby on Rails v8.0.1[3] which correlates to Git commit cf6ff17e9a3c6c1139040b519a341f55f0be16cf[4].

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

The ActionMailer and ActionMailbox components were inspected for logic and implementation bugs.

Semgrep[5] was used for an initial static analysis of the source code.

- Cookies were checked for `Secure` and `HttpOnly` attributes.
- Parsing issues for different formats (JSON, HTTP).
- WebSocket hijacking was investigated using dynamic and static analysis.
- The code was checked for URL[6] parsing discrepancies

[3] `https://github.com/rails/rails/releases/tag/v8.0.1`
[4] `https://github.com/rails/rails/tree/cf6ff17e9a3c6c1139040b519a341f55f0be16cf`
[5] `https://semgrep.dev/index.html`
[6] Uniform Resource Locator

The ActionCable code has been reviewed with a focus on authentication and deauthentication. The framework has been inspected for availability and default values of web security features such as CSP[7], SRI[8], HSTS[9], `Referrer-Policy`, `X-Content-Type-Options`, `X-Frame-Options`, and cookie attributes such as `HttpOnly`, `Secure`, and `SameSite`. A special focus was put on parameter handling in regard to type tampering, bypassing protection mechanisms, overwriting path and query parameters, and inconsistencies between different rails versions.

The ActiveStorage module was reviewed for dangerous file accesses and logic bugs. Due to the potent attack surface, the dependencies handling image parsing and transformations were audited for security issues. ActiveRecord was audited for undocumented potential SQL injections and design issues. Third party dependencies were checked for obvious security vulnerabilities and potential supply chain issues.

Suggestions for next steps in securing this scope can be found in section 2.3.

2.3 Recommended Further Tests

X41 recommends to perform code audits of the components used, such as the mail[10] library.

Given that many of the security features of the Ruby on Rails framework depend on the proper and secure usage framework functionality, further audits could be conducted on Gems that are popular within the Ruby on Rails ecosystem, where identified vulnerabilities would impact a wide array of Ruby on Rails deployments.

[7] Content Security Policy
[8] Subresource Integrity
[9] HTTP Strict Transport Security
[10] `https://github.com/mikel/mail/`

3 Threat Model and Test Plan

A threat model is a framework that seeks to identify the trust boundaries inherent to a software system. This offers a systematic approach to arrive at a list of attack surfaces, which can be translated into the components that may be targeted by attackers. It describes the outcome of a successful attack and common vulnerabilities, which may enable an attack. From this information, a test plan can be developed that covers a section or all of the identified attack surfaces. The test plan can be used to establish coverage as a metric to be taken into account by further security review in the future. Last but not least, the threat model can be employed to decide whether a bug is to be considered security sensitive.

Ruby on Rails is a web application framework, therefore any vulnerability identified within the framework potentially exposes the application using it.

This test plan and threat model cover the generic aspects of a (web) application framework applied to the specifics of Ruby and Ruby on Rails.

3.1 System Overview

This section provides an overview of the Ruby on Rails framework, in particular its architecture, key components, and typical deployment scenarios. Understanding these aspects is crucial for identifying potential technical security risks and developing an effective test plan.

3.1.1 Ruby on Rails Architecture and Key Components

Ruby on Rails follows the MVC[1][2] design pattern. The framework is split up into several parts for different aspects of the application.

[1] Model-View-Controller
[2] `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`

- ActionCable - WebSocket integration
- ActionMailbox - Inbound email handling
- ActionMailer - Email sending
- ActionPack - Request lifecycle
- ActionText - Rich text handling
- ActionView - View layer
- ActiveJob - Background processing
- ActiveModel - Model layer
- ActiveRecord - ORM[3] Database layer
- ActiveStorage - File upload and storage handling
- ActiveSupport - Utility classes

3.1.2 Typical Deployment Scenarios

Ruby on Rails applications can be deployed in various ways, each with its own security considerations.

1. Single-server deployment:

   - Rails application server (e.g., Puma, Unicorn)
   - Database server (e.g., PostgreSQL, MySQL)
   - Web server (e.g., Nginx, Apache) as a reverse proxy

   This is the simplest deployment scenario, suitable for small to medium-sized applications. All components run on a single server, which can be easier to manage but may have limitations in terms of scalability and fault tolerance.

2. Multi-server deployment:

   - Load balancer (e.g., HAProxy, Nginx)
   - Multiple application servers
   - Database server (possibly with replication)
   - Caching server (e.g., Redis, Memcached)
   - Background job workers

   This setup is more complex but offers better scalability and reliability. It distributes the load across multiple servers and introduces redundancy, making the application more resilient to failures.

3. Cloud-based deployment:

[3] Object-Relational Mapping

- Platform as a Service (PaaS[4]) providers (e.g. Heroku, AWS[5] Elastic Beanstalk)
- Containerized deployment (e.g. Docker with Kubernetes)
- Serverless deployment (e.g. AWS Lambda with API[6] Gateway)

Cloud-based deployments offer flexibility, scalability, and often come with built-in security features. However, they require careful configuration to ensure proper security measures are in place, especially when dealing with sensitive data.

4. Microservices architecture:

   - Multiple applications as separate services
   - API Gateway
   - Service discovery and communication layer

   This advanced architecture breaks down the application into smaller, independent services. While it offers great flexibility and scalability, it also introduces complexity in terms of service communication and security management.

Each deployment scenario has its own security implications and requires specific considerations in terms of network security, access control, and data protection.

3.2 Assets Identification

- Source code
- Database
- User data
- Configuration files
- Session information
- Trust relationship with other entities
- Browser of users

3.3 Entry Points

1. Web interface:

   - HTTP[7]/HTTPS[8] requests to the application's endpoints

[4] Platform as a Service
[5] Amazon Web Services
[6] Application Programming Interface
[7] HyperText Transfer Protocol
[8] HyperText Transfer Protocol Secure

- User-facing forms and interactive elements
- Authentication and login pages
- File upload interfaces

2. API endpoints:

   - RESTful API routes
   - GraphQL endpoints (if implemented)
   - WebSocket connections (via ActionCable)
   - Webhook receivers

3. Database connections:

   - ActiveRecord ORM interactions
   - Raw SQL[9] queries (if used)
   - Database migration scripts

4. File system access:

   - File uploads and downloads (possibly using ActiveStorage)
   - Log file writing and reading
   - Temporary file creation and manipulation

5. External service integrations:

   - Third-party API calls
   - OAuth[10] authentication providers
   - Payment gateways
   - Email services (via ActionMailer)
   - Background job processors (using ActiveJob)

6. Command-line interface:

   - Rails console
   - Rake tasks
   - Custom scripts using the Rails environment

7. Configuration files:

   - Environment variables
   - Database configuration files
   - Initializers and application settings

8. Inbound email handling:

[9] Structured Query Language
[10] Open Authorization

- Email processing via ActionMailbox

9. View rendering:
    - Template rendering (via ActionView)
    - JavaScript execution in the browser
    - CSS[11] styling and processing
10. Caching mechanisms:
    - Page caching
    - Fragment caching
    - Low-level caching

Each of these entry points represents a potential avenue for attackers to interact with the Ruby on Rails application. It's crucial to implement proper security measures, input validation, and access controls at each of these points to maintain the integrity of the trust boundaries identified in the threat model.

3.4 Trust Boundaries

In this section, trust boundaries of a typical Ruby on Rails application are identified and defined. These trust boundaries are a fundamental part of the threat model and guide the efforts of securing the framework. A bug is considered a security vulnerability if and only if it violates one of the trust boundaries of the respective threat model. The section concludes with a list of bug classes that generally violate one or more trust boundaries.

3.4.1 Client-Server Boundary

Applications written using Ruby on Rails are meant to serve resources using the HTTP protocol family to users connecting over a remote network. The interaction typically involves the user sending a request for some resource and the application generating a reply and sending it back to the user.

In the context of this threat model, the interface between the user (client) and the application (server) constitutes a trust boundary. A violation of this boundary means that the user can craft a request or a series of requests, which causes unintended consequences for the state of the server, for example returning data that the user is not authorized to receive or executing code that the user is not authorized to execute, possibly code that the user supplied themselves.

[11] Cascading Style Sheets

An example of a vulnerability undermining this trust boundary is deserialization attacks, where user-supplied data is deserialized into functional Ruby classes, including method definitions that are executed. Another example is a command injection vulnerability, where the user is able to craft input in a way that it is used to execute commands in a shell on the server.

3.4.2 Application-Database Boundary

Applications written using Ruby on Rails (and other similar frameworks) rely on a backend database for persistent storage of data. Typically, these databases are provided by some implementation of a relational database, which can be interacted with via SQL. The ORM database layer provides an abstraction over the variants of compatible database backends. It enables the programmer to let the row of a table in the database be represented by an instance of a Ruby class. The abstraction layer's task is to map the instances of the Ruby classes to the internal representation in the specific SQL database and back. Further, it handles the network connection and authentication with the database.

In the context of this threat model, the interfaces between objects backed by the same database that require differing levels of authorization, constitute trust boundaries. A violation of this trust boundary means that the user can craft a request or a series of requests, which causes unintended consequences in the state of the database, for example returning data that the user is not authorized to receive.

A prominent bug class that violates this trust boundary is SQL injection, where a user of the application can inject SQL statements into data supplied to the application in a way that the application executes the SQL statements on the backend database.

3.4.3 Application-OS Boundary

Ruby on Rails applications can interact with the operating system and local file system. This might lead to files being created with insecure permissions or race conditions when reading data from files. The Ruby on Rails framework should ensure that these kinds of security issues cannot happen easily.

3.4.4 Application-Network Boundary

Ruby on Rails applications can interact with other network applications such as microservices available via REST[12] APIs. These interactions need to be authorized, encrypted, and integrity

[12] Representational State Transfer

protected. Additionally, attack classes such as server-side request forgery (SSRF[13]) should be prevented.

3.4.5 Inter-Component Boundaries within the Application

Applications written using Ruby on Rails often desire to authenticate and authorize a user, such that the application is able to determine which information a user is entitled to receive, or which functions a user is allowed to execute. Ruby on Rails provides various APIs around handling sessions, encryption, and verification. The APIs performing these tasks constitute a trust boundary within this threat model. A Ruby on Rails application shall not authenticate and/or authorize a user to a level of access that they are not entitled to.

Examples of violations to this trust boundary are broadly distributed. Cross-Site Scripting (XSS[14]) and Cross-Site Request Forgery (CSRF[15]) often depend on the concrete application logic, but may also be a vulnerability in the framework.

Vulnerabilities in the session management often stem from errors in the framework and misconfigurations are usually an error by the developer of the application.

3.4.6 Ruby

With Rails being developed in Ruby, security issues in Ruby might affect Rails as well. Ruby functions might be used in an insecure manner or contain vulnerable code paths. These might be in the Ruby code itself or the Ruby C extensions.

3.5 Threat Categories

- Injection attacks (SQL, command, etc.)
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Authentication and session management vulnerabilities
- Insecure Direct Object References
- Security misconfiguration
- Sensitive data exposure
- Broken access control

[13] Server-Side Request Forgery
[14] Cross-site Scripting
[15] Cross-Site Request Forgery

- Insufficient logging and monitoring
- Race conditions
- Denial-of-service issues

3.6 Specific Ruby on Rails Vulnerabilities

- Mass assignment
- Remote code execution via YAML[16] deserialization
- Unsafe reflection methods
- Template injection
- Insecure defaults in older versions

3.7 Conclusions and Proposed Test Plan

Due to the size of Ruby on Rails, it will not be possible to cover all the tests and tasks in the following test plan. The tests performed will be covered in the final report along with suggestions on what to focus on in future audits.

3.7.1 Source Code Auditing

The source code will be audited to various bug classes, such as those highlighted by the OWASP Top 10 2024. Furthermore, the threats listed in section *Threat Categories* will be inspected where sensible. Additionally, security issues specific to Ruby on Rails will be audited for. These include, but are not limited to:

- Mass assignment issues
- Serialization and Deserialization issues
- Unsafe reflection methods
- Template injection
- Insecure defaults

[16] YAML Ain't Markup Language

3.7.2 Business Logic

The various Action-components should be audited for security issues stemming from logic bugs related to the component. These bugs are tightly related to the task of the component. For example, the Action Mailbox will be inspected on how it handles maliciously formatted emails. This might include the handling of malformed headers, headers missing or occurring multiple times, a flood of emails, or the very slow receiving of email data.

3.7.3 Deployment Scenarios

The various deployment scenarios should be compared and the implications on the security of deployments inspected. Configuration options should be audited for easy-to-make mistakes that might affect a subset of the deployment scenarios.

3.7.4 Boundary Trusts

Various boundaries were identified in the Ruby on Rails framework. Each side of these boundaries will have assumptions about the other side embedded in the code base. When these assumptions are violated, there might be ways to abuse these violations in a security-relevant manner. Therefore, these assumptions and interactions should be audited.

3.7.5 Security Mechanisms

Security mechanisms implemented by Ruby on Rails should be audited for security bugs which might allow attackers to circumvent these protections. Additionally, they will be inspected for other forms of abuse, e.g. denial-of-service attacks that are only possible due to these mechanisms.

3.7.6 Improve Tooling

The following actions should be taken to improve the security-relevant tooling of the Ruby on Rails codebase:

- Inspect and increase *ruzzer* fuzzing coverage for Ruby C extensions used
- Attempt OSS-Fuzz integration
- Inspect current and create new SAST rules

4 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Open Source Technology Improvement Fund are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating 'none' are called informational findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration
[2] `https://www.mitre.org`

5 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 5.1. Additionally, findings without a direct security impact are documented in Section 5.2.

5.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

5.1.1 ROR-CR-23-01: Potential Remote Code Execution in *image_processing* Gem

Severity:	HIGH
CWE:	94 – Improper Control of Generation of Code ('Code Injection')
Affected Component:	activestorage/lib/active_storage/transformers/image_processing_transformer.rb

5.1.1.1 Description

Ruby on Rails exposes the *image_processing* Gem as part of ActiveStorage, which enables app developers to perform basic image processing. The fix for a vulnerability reported on March 1st, 2022 [1], is incomplete. If a vulnerable Rails app exposes the type of transformations to be applied to an image to a remote attacker, they may execute arbitrary commands in the context of the Rails app. Note that when *MiniMagick* is used, Ruby on Rails performs validation of transformations, which matches the method against a white list *supported_image_processing_methods*. The required method *send* is not among them, so there is seemingly no way to exploit this issue with

[1] https://github.com/janko/image_processing/security/advisories/GHSA-cxf7-qrc5-9446

MiniMagick. However, if the processing variant is *VIPS*, transformations are not validated. The underlying issue is that the fix attempts to constrain methods to be called to public methods, which would exclude sensitive methods like ***system()***, ***spawn()***, and ***eval()***. However, the ***send()*** method itself is public and thus not excluded. Therefore it is possible to execute code if the processor can be invoked in a way as shown in listing 5.1:

```
1  ImageProcessing::Vips.source(@image).apply(send: ["spawn", "touch foo.txt"])
```

Listing 5.1: Vulnerable Library Call

Other paths to the vulnerable methods in the gem *image_processing* exist, but are not called by Ruby on Rails in a way that would make them exploitable due to an app developer exposing them.

5.1.1.2 Solution Advice

X41 recommends applying white lists unconditionally.

5.1.2 ROR-CR-23-02: Ruby on Rails Ships Vulnerable Version of Trix Editor

Severity:	LOW
CWE:	1395 – Dependency on Vulnerable Third-Party Component
Affected Component:	actiontext/app/assets/javascripts/trix.js

5.1.2.1 Description

Ruby on Rails ships the Trix editor as part of the ActionText module. Trix editor in the shipped version (2.1.10) is vulnerable to an XSS attack [2]. While the impact of the vulnerability is low, the process to keep the dependency up to date when a vulnerability is reported has, failed.

5.1.2.2 Solution Advice

X41 recommends to update the shipped and Trix editor to the latest version and employ processes to keep third party shipped with Ruby on Rails up to date.

[2] `https://github.com/basecamp/trix/security/advisories/GHSA-j386-3444-qgwg`

5.1.3 ROR-CR-23-03: ActionMailer Default Connect Policy Allows Downgrade Attack

Severity:	LOW
CWE:	453 – Insecure Default Variable Initialization
Affected Component:	actionmailer/lib/action_mailer/base.rb

5.1.3.1 Description

Among the default settings for ActionMailer are:

1. `enable_starttls: false`
2. `enable_starttls_auto: true`

The first setting removes the requirement for TLS[3] on a connection to the configure SMTP[4] server. The second setting enables detecting whether a server supports TLS or not. A potential attacker, who can intercept and manipulate traffic, may impersonate the SMTP server and force a plain-text connection, rendering the connection no longer secure.

5.1.3.2 Solution Advice

X41 recommends to change the default settings or warn the user if they are connecting to a non-local SMTP server without enforcing TLS.

[3] Transport Layer Security
[4] Simple Mail Transfer Protocol

5.1.4 ROR-CR-23-04: Insecure Default Value for Same Site Protection of Cookies

Severity:	LOW
CWE:	453 – Insecure Default Variable Initialization
Affected Component:	actionpack

5.1.4.1 Description

The default value for same site protection for cookies is `lax`. This potentially allows cross site request forgery attacks.

5.1.4.2 Solution Advice

X41 recommends changing the default setting for the same site protection for cookies to `strict`.

5.1.5 ROR-CR-23-05: Weak Defenses Against SQL Injections

Severity:	LOW
CWE:	657 – Violation of Secure Design Principles
Affected Component:	ActiveRecord

5.1.5.1 Description

Active Record uses a Builder pattern to construct an AST[5], which will then be translated to an SQL statement tailored for the target database. At various points documented at [6] the app developer can introduce `SqlLiteral` objects as nodes in the AST. `SqlLiterals` represent literal SQL strings, that will be translated to the final SQL statement without escaping special characters. These entry points are responsible for SQL injections to occur.

5.1.5.2 Solution Advice

X41 recommends to disallow the creation of un-escaped `SqlLiteral` objects with user input in favor of a complete model of SQL. Wherever possible, methods should be introduced that model a certain database operation explicitly. The methods should only allow arguments which can be validated against the model (like table or column names, functions, etc.) or can be escaped. To ease the transition on app developers, an optional strict mode setting could be implemented, which disallows un-escaped `SqlLiterals` to be appended to the AST.

[5] Abstract Syntax Tree
[6] `https://rails-sqli.org/`

5.1.6 ROR-CR-23-06: Potential Code Execution via Redis Cache

Severity:	LOW
CWE:	502 – Deserialization of Untrusted Data
Affected Component:	ActiveSupport

5.1.6.1 Description

`ActiveSupport` offers a `RedisCacheStore` to be used for caching purposes in a Rails app. Ruby objects are (optionally) compressed and ***Marshal.load()***-ed and stored as value under some key. If an attacker can store arbitrary values in a Redis instance in use by the Rails app, they may be able to achieve code execution in the context of the Rails app. The impact is low, as a fair bit of control over the infrastructure (or another bug) is prerequisite for the attack.

One possible scenario would be that the Rails app allows arbitrary server-side requests to be directed to the backend Redis cache. The attacker would be able to forge a key value pair with a Ruby object as value, which would execute arbitrary code upon retrieval from the cache.

5.1.6.2 Solution Advice

X41 recommends to replace un-marshaling objects from remote sources with a safe alternative.

5.1.7 ROR-CR-23-07: Cannot Revoke Session's WebSocket Connection

Severity:	LOW
CWE:	285 – Improper Authorization
Affected Component:	Generators::AuthenticationGenerator

5.1.7.1 Description

When using the `rails generate authentication` command, Rails creates the following file.

```
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      set_current_user || reject_unauthorized_connection
    end

    private
      def set_current_user
        if session = Session.find_by(id: cookies.signed[:session_id])
          self.current_user = session.user
        end
      end
  end
end
```

Listing 5.2: /app/channels/application_cable/connection.rb

The generated code ensures that WebSocket connections are only created for authenticated users. It also generates the ***terminate_session()*** method, shown below, which is called when a user logs out.

```
def terminate_session
    Current.session.destroy
    cookies.delete(:session_id)
end
```

Listing 5.3: /app/controllers/concerns/authentication.rb

When using the generated code, the `ActionCable::Connection` is only identified by the user that was logged in during its creation.

This does not allow identifying – and terminating – WebSocket connections based on a particular session. It would be possible to terminate all connections belonging to a user. However, the ***terminate_session()*** method does not terminate any WebSocket connections.

5.1.7.2 Solution Advice

X41 recommends adding the session as an attribute to the `ActionCable::Connection`, and to allow terminating based on this identifier.

As a workaround, all WebSocket connections belonging to a user could be disconnected using ***disconnect(reconnect: true)***, effectively causing all connections to reconnect and only those with a valid session to be accepted again.

5.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

5.2.1 ROR-CR-23-100: Non-Cryptographic Randomness in Mail Library

Affected Component:	lib/mail/mail.rb:random_tag

5.2.1.1 Description

The mail library[7] used by Ruby on Rails ActionMailer and ActionMailbox generates boundary tags that separate different parts of a multipart email [8]. These should be unique and must not appear in other parts of an email. If an attacker would be able to inject data into an email generated by Rails and guess the boundary tag, it could allow the attacker to create additional attachments for that email as seen in listing 5.4:

```
1  RANDOM_TAG='%x%x_%x%x%d%x'
2
3  def self.random_tag
4    t = Time.now
5    sprintf(RANDOM_TAG,
6            t.to_i, t.tv_usec,
7            $$, Thread.current.object_id.abs, self.uniq, rand(255))
8  end
9
10 private
11
12 def self.something_random
13   (Thread.current.object_id * rand(255) / Time.now.to_f).to_s.slice(-3..-1).to_i
14 end
15
16 def self.uniq
17   @@uniq += 1
18 end
```

Listing 5.4: Boundary Tag Creation

[7] https://github.com/mikel/mail/
[8] https://datatracker.ietf.org/doc/html/rfc2046#section-5.1

Various sources are used to create the boundary line, of these ***t.to_i*** offers little randomness, since its the number of seconds since the Epoch[9]. ***t.tv_usec*** has around 6^{10} combinations[10], but might be less if the timer resolution is bad. ***$$*** and ***Thread.current.object_id.abs*** are more or less static for a running Ruby on Rails instance. ***self.uniq*** is a counter that gets increased for each generated tag and can therefore be guessed as well. ***rand(255)*** gets its entropy from a PRNG[11] in the Ruby Kernel[12].

This is considered an informational issue, since the mail library is not in scope for this test and the entropy seems to be sufficient in a typical Ruby on Rails setup.

5.2.1.2 Solution Advice

X41 recommends to increase the entropy in the generated boundary tags and use cryptographically secure sources of randomness for this purpose.

[9] `https://ruby-doc.org/core-3.1.1/Time.html#method-i-to_i`
[10] `https://ruby-doc.org/core-3.1.1/Time.html#method-i-tv_usec`
[11] Pseudo Random Number Generator
[12] `https://ruby-doc.org/core-3.1.0/Kernel.html#method-i-rand`

5.2.2 ROR-CR-23-101: URL Parameter Parsing

Affected Component: ActionDispatch::ParamBuilder

5.2.2.1 Description

While fuzzing the URL parameter parser, X41 noticed an inconsistency between the URL parameter parsing used in Rails 7 and the recently introduced implementation used in Rails 8.

The input query string is parsed differently as shown in the following listings 5.5:

```
1 comments[]id=1&comments[]&comments[]delete=true&comments[]id=2
```

Listing 5.5: Query String for Parsing Test

This inconsistency can be observed between the two versions as shown in listing 5.6:

```
1 # In Rails 7.2.2.1
2 {"comments"=>[{"id"=>"1"}, {"delete"=>"true", "id"=>"2"}]}
3 # In Rails 8.0.1
4 {"comments"=>[{"id"=>"1", "delete"=>"true"}, {"id"=>"2"}]}
```

Listing 5.6: URL Parameter Parsing Difference

The issue occurs when a Hash parameter name (`comments[]`) is repeated with no key and without a trailing equals character (`=`). Rack stops parsing the current Hash, parses the parameter's empty value as nil, and adds subsequent keys to a new Hash. The nil value is later removed by the ***deep_munge*** operation. Rails seems to ignore the empty value and continues adding keys to the current Hash. It should be noted that both variants of interpreted values can be achieved through legitimate query strings in either version of Rails. The chance of such a query string being accidentally created is limited because it requires omitting the trailing equals character (`=`), which is allowed when parsing [13] the query string, but required when serializing it[14]. Clients strictly following the URL Standard should not be affected by this inconsistency.

X41 does not regard this as a security threat.

The fuzzer source code may be found in Appendix A.1.

[13] `https://url.spec.whatwg.org/#urlencoded-parsing`
[14] `https://url.spec.whatwg.org/#urlencoded-serializing`

5.2.2.2 Solution Advice

X41 recommends to remove inconsistencies, or document the changed behaviour. The Rails implementation should be inspected for further inconsistencies in URL query string interpretation.

5.2.3 ROR-CR-23-102: HTTP Body Accepted in GET Requests

Affected Component: ActionDispatch

5.2.3.1 Description

X41 found that Ruby on Rails accepts HTTP *GET* requests with content bodies attached, and parses the body content into *params* as seen in the following listing 5.7:

```
1 GET / HTTP/1.1
2 Host: 127.0.0.1:3000
3 Content-Type: application/json
4
5 {"foo":1}
```

Listing 5.7: HTTP GET Request with Body

Per RFC[15] 9110[16], "content received in a *GET* request has no generally defined semantics", and "cannot alter the meaning or target of the request".

Developers might be unaware of this functionality and not expect e.g. JSON[17] value types to be available as request content types.

Developers might be unaware of this functionality and not expect e.g., JSON value types to be available as request content types.

Attackers might also use this uncommon functionality to submit request parameters that do not show up in web server logs, or bypass the filters of a WAF[18].

5.2.3.2 Solution Advice

X41 recommends to ignore the request body for *GET*, *HEAD*, and *DELETE* requests, unless support of this functionality was specifically enabled.

[15] Request for Comments
[16] `https://www.rfc-editor.org/rfc/rfc9110.html#section-9.3.1-6`
[17] JavaScript Object Notation
[18] Web Application Firewall

5.2.4 ROR-CR-23-103: Request Parameter Type Tampering

Affected Component:	ActionDispatch::ParamBuilder, ActionController::Parameters

5.2.4.1 Description

Contrary to the URL Standard[19], Ruby on Rails supports URL query parameter values that are not strings. Specifically, the non-scalar value types `Array` and `Hash` are supported, and nested versions thereof.

The same issue exists when reading user-submitted values from an HTTP body and accessing a key without specifying the expected type.

Rails does provide a method to prevent this by using ***params.permit()***[20] when accessing parameters. However, this requires developers to explicitly use this function and does not affect the use of accessing ***params*** directly, where the default is to allow any supported types.

Developers may not always account for a parameter type being different from what they expect, and may not be aware of potential implications when passing a different type of value to a method.

5.2.4.2 Solution Advice

X41 recommends to limit the parameter values to scalar types by default, and requiring developers to explicitly permit other types when accessing user-submitted values.

[19] `https://url.spec.whatwg.org/#urlencoded-parsing`
[20] `https://api.rubyonrails.org/classes/ActionController/Parameters.html#method-i-permit`

5.2.5 ROR-CR-23-104: Subresource Integrity

Affected Component: ActionView::Helpers

5.2.5.1 Description

Helper methods such as ***javascript_include_tag***[21] and ***stylesheet_link_tag***[22] do support SRI when providing the `integrity` attribute. However, this is not documented, not used in examples, and not mentioned in the Helper methods such as ***javascript_include_tag***[23] and

5.2.5.2 Solution Advice

Documenting the availability of SRI, and encouraging its use could contribute to a more widespread use of this web security feature.

Additionally, Ruby on Rails could further contribute to the spread of SRI by requiring the `integrity` attribute when using a URL as a source. Setting the attribute to a value such as `false` could disable it when developers make the active decision of not using SRI.

[21] https://api.rubyonrails.org/classes/ActionView/Helpers/AssetTagHelper.html#method-i-javascript_include_tag

[22] https://api.rubyonrails.org/classes/ActionView/Helpers/AssetTagHelper.html#method-i-stylesheet_link_tag

[23] https://api.rubyonrails.org/classes/ActionView/Helpers/AssetTagHelper.html#method-i-javascript_include_tag

5.2.6 ROR-CR-23-105: ANSI Escape Sequences Logged Unfiltered

Affected Component: Logging

5.2.6.1 Description

X41 found that Rails' logging accepts and stores ANSI[24] escape sequences unfiltered in log files, which can lead to unexpected behavior when viewing logs.

ANSI escape sequences are special character combinations beginning with '\e[' that control terminal text formatting such as colors, cursor positioning, and text attributes. When logs containing these sequences are displayed in compatible terminals, the formatting is applied rather than the raw codes being shown.

Users may be unaware that these sequences are stored verbatim in log files, which can lead to log parsers breaking due to unexpected control characters, terminal manipulation when viewing logs and log data being hidden or visually altered

Attackers might leverage this behavior to manipulate log presentation, potentially hiding malicious activities or creating misleading log entries when viewed by administrators.

5.2.6.2 Solution Advice

X41 recommends that logging systems should sanitize or escape ANSI control sequences before writing to log files, unless terminal formatting is specifically required.

[24] American National Standards Institute

6 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

ANSI American National Standards Institute . . . 36
API Application Programming Interface . . . 11
AST Abstract Syntax Tree . . . 25
AWS Amazon Web Services . . . 11
CSP Content Security Policy . . . 8
CSRF Cross-Site Request Forgery . . . 15
CSS Cascading Style Sheets . . . 13
CWE Common Weakness Enumeration . . . 19
HSTS HTTP Strict Transport Security . . . 8
HTTP HyperText Transfer Protocol . . . 11
HTTPS HyperText Transfer Protocol Secure . . . 11
JSON JavaScript Object Notation . . . 33
PRNG Pseudo Random Number Generator . . . 30
REST Representational State Transfer . . . 14
RFC Request for Comments . . . 33
SMTP Simple Mail Transfer Protocol . . . 23
SQL Structured Query Language . . . 12
SRI Subresource Integrity . . . 8
SSRF Server-Side Request Forgery . . . 15
TLS Transport Layer Security . . . 23
URL Uniform Resource Locator . . . 7
WAF Web Application Firewall . . . 33
XSS Cross-site Scripting . . . 15

A Appendix

A.1 URL Parameter Fuzzer

X41 wrote a differential fuzzer that compares the results of the (Rack) parameter parsing used in Rails 7 and the recently introduced implementation used in Rails 8.

The fuzzer ignores inconsistencies in what the different implementations deem invalid input, but does not ignore abnormal crashes.

After the fuzzer found various instances of the same inconsistency reported in Informational Note 5.2.2, it has been modified to ignore this as well (*KNOWNBUG*).

The fuzzer was executed as follows, and no further inconsistencies or abnormal crashes were found.

```
env ASAN_OPTIONS="allocator_may_return_null=1:detect_leaks=0:use_sigaltstack=0" \
  LD_PRELOAD=$(ruby -e 'require "ruzzy"; print Ruzzy::ASAN_PATH') \
  ruby trace.rb
```

Listing A.1: Fuzzing Command Line

```
require 'ruzzy'
Ruzzy.trace('fuzz.rb')
```

Listing A.2: trace.rb

```
require 'rack'
require 'rails'
require 'ruzzy'

# https://github.com/trailofbits/ruzzy/issues/22
Signal.trap("SIGALRM") do
    puts "Alarm received!"
end

```

```
include Rack::Request::Helpers

# matches patterns where a hash param is repeated with no key and no value
# such as name[]key1&name[]&name[]key2
KNOWNBUG = /
        (\A|&)                  # start of param
        \s*                     #
        (?<name> [^&=] (\[[^&=]*\])* [^&=\[]* (\[[^&=]*\])* \[\] ) # ...
        [^&=]*                  # key name after the brackets
        (=[^&\s]* | &+)         # "=" param-value, or "&"
        \s*                     #
        ( [^&]* &)*             # any amount of other params
        \s*                     #
        \k<name>                # the param name from above again, with no key or value
        \s*                     #
        (&|\Z)                  # end of param
    /x

def try_rack(str)
    ActionDispatch::Request::Utils::NoNilParamEncoder.normalize_encode_params(parse_query(str))
rescue Rack::Utils::ParameterTypeError, Rack::Utils::InvalidParameterError, Rack::QueryParser::
↪  ParamsTooDeepError => e
    return {"error" => true}
end

def try_rails(str)
    ActionDispatch::ParamBuilder.from_query_string(str)
rescue ActionDispatch::ParamError => e
    return {"error" => true}
end

fuzz = lambda do |input|
    rack = try_rack(input)
    rails = try_rails(input)
    return 0 if rails === {"error" => true} or rack === {"error" => true}
    unless rack === rails
        if KNOWNBUG.match(input)
            return 0
        end

        err = "Got different results for: #{input.inspect}\n"
        err += "rack:  #{rack.inspect}\n"
        err += "rails: #{rails.inspect}"
        raise err
    end
    return 0
end

Ruzzy.fuzz(fuzz)
```

Listing A.3: fuzz.rb