

# Universidad Isabel I

## **TRABAJO FIN DE MÁSTER**

Curso 2024/25

### **SECURE FRAMEWORK:**

Desarrollo de un framework de seguridad Ruby on Rails que permita la construcción de aplicaciones web con componentes seguros

Alumno:

José Antonio Torres Coca

Tutor:

Antonio Pérez Carrasco

UNIVERSIDAD INTERNACIONAL ISABEL I DE CASTILLA

FACULTAD DE CIENCIAS Y TECNOLOGÍA

**MÁSTER EN CIBERSEGURIDAD**

Autor: José Antonio Torres Coca

Trabajo Fin de Máster

Curso 2024/25

*SECURE FRAMEWORK: Desarrollo de un framework de seguridad Ruby on Rails que permita la construcción de aplicaciones web con componentes seguros*

Tutor: Antonio Pérez Carrasco

Universidad Internacional Isabel I de Castilla

Facultad de Ciencias y Tecnología

**Máster en Ciberseguridad**

## Resumen

En el panorama actual del desarrollo web, persiste una brecha significativa entre las buenas prácticas de seguridad y su implementación efectiva. Esta situación expone a las aplicaciones a vulnerabilidades comunes como inyecciones SQL, XSS y CSRF, a menudo por la falta de herramientas que faciliten la integración de la seguridad en el ciclo de vida del desarrollo.

Este trabajo aborda dicho problema mediante el diseño y la implementación de **secure\_framework**, un framework de seguridad para Ruby on Rails. La solución, distribuida como una gema, adopta un enfoque de "seguridad por defecto" para permitir la construcción de aplicaciones web robustas. El framework orquesta la instalación y configuración de componentes esenciales para la autenticación, autorización, gestión de dependencias y la configuración de cabeceras de seguridad, abstrayendo la complejidad para el desarrollador.

Para validar su eficacia, se construyó una aplicación de demostración sobre la que se ejecutó un riguroso proceso de pruebas. Los análisis de seguridad estáticos con Brakeman y de estilo con Rubocop no detectaron ninguna advertencia. Asimismo, una completa suite de pruebas funcionales con Rspec, que simulaba diversos escenarios de ataque, concluyó sin fallos.

Los resultados demuestran que el framework constituye una solución efectiva para mitigar de raíz un amplio espectro de debilidades, validando la hipótesis de que un enfoque de seguridad por defecto y centrado en el desarrollador es viable y eficaz.

El código fuente del framework y de la aplicación de demostración se encuentra disponible en los siguientes repositorios de GitHub:

- **Framework:** [https://github.com/joseantonio2001/secure\\_framework](https://github.com/joseantonio2001/secure_framework)
- **Aplicación de demostración:** [https://github.com/joseantonio2001/demo\\_app](https://github.com/joseantonio2001/demo_app)

**Palabras clave:** Ruby on Rails, framework de seguridad, seguridad por defecto, desarrollo seguro.

## Abstract

In today's web development landscape, a significant gap persists between good security practices and their effective implementation. This situation exposes applications to common vulnerabilities such as SQL injections, XSS and CSRF, often due to a lack of tools that facilitate the integration of security into the development lifecycle.

This work addresses this problem by designing and implementing `secure_framework`, a security framework for Ruby on Rails. The solution, distributed as a gem, adopts a 'security by default' approach to enable the construction of robust web applications. The framework orchestrates the installation and configuration of essential components for authentication, authorisation, dependency management and security header configuration, abstracting complexity for the developer.

To validate its effectiveness, a demo application was built and a rigorous testing process was run on it. Static security analysis with Brakeman and style analysis with Rubocop did not detect any warnings. In addition, a full functional test suite with Rspec, simulating various attack scenarios, was concluded without failure.

The results demonstrate that the framework is an effective solution for root mitigating a wide range of weaknesses, validating the hypothesis that a default, developer-centric approach to security is viable and effective.

The source code of the framework and demo application is available in the following GitHub repositories:

- **Framework:** [https://github.com/joseantonio2001/secure\\_framework](https://github.com/joseantonio2001/secure_framework)
- **Demonstration application:** [https://github.com/joseantonio2001/demo\\_app](https://github.com/joseantonio2001/demo_app)

**Keywords:** Ruby on Rails, security framework, security by default, secure development.

## Agradecimientos

La culminación de este Trabajo Fin de Máster representa mucho más que la consecución de un objetivo académico; marca el final de una etapa de profundos desafíos personales, de resiliencia y, en última instancia, de autodescubrimiento. Este proyecto ha sido un faro en un año complejo, y su realización no habría sido posible sin el apoyo de varias personas fundamentales a las que deseo expresar mi más sincera gratitud.

En primer lugar, agradezco a mi tutor, Antonio Pérez Carrasco, por su guía y confianza a lo largo del desarrollo de este trabajo. Su orientación ha sido crucial para dar forma y rigor a este proyecto.

Desde una perspectiva más personal, este último año supuso un regreso a casa que se tornó en un viaje inesperado hacia la soledad y la introspección. En los momentos de mayor oscuridad, cuando el sol parecía solo una imagen tras la ventana, encontré pilares que me sostuvieron. Agradezco a mi familia, por su apoyo y su preocupación silenciosa pero constante. A mis amigos, que, aunque compartimos escasos momentos, representaron un respiro necesario, una bocanada de aire fresco que me recordaba que la oscuridad no era permanente.

Finalmente, y en un plano que trasciende lo terrenal, quiero dedicar este logro a mi abuelo Pepe, de quien soy heredero no en bienes, sino en convicciones. En un mundo de incertidumbres, su recuerdo es mi única certeza y la razón última de mi fe.

José Antonio Torres Coca

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Objetivos</b>	<b>3</b>
<b>3. Metodología</b>	<b>4</b>
Fase 1: Estudio del estado del arte y tecnologías existentes	4
Fase 2: Definición de requisitos y componentes del framework	4
Fase 3: Diseño e implementación del framework	5
Fase 4: Validación y pruebas de seguridad	5
<b>4. Estado del Arte y Justificación del Trabajo</b>	<b>7</b>
4.1. Introducción al desarrollo seguro de aplicaciones web	7
Security by Design: más allá de las buenas prácticas	8
El rol del desarrollador: entre la abstracción y la responsabilidad	8
4.2. Ruby on Rails: Evolución de la seguridad	9
Seguridad desde el diseño: filosofía de Rails	9
4.2.1. Evolución por Versiones	10
4.3. Herramientas, gemas y prácticas actuales en seguridad con RoR	15
4.3.1. Herramientas de Análisis Estático de Seguridad (SAST)	16
4.3.2. Gemas de Seguridad para Ruby on Rails	18
4.3.3. Prácticas Actuales de Seguridad en RoR	20
4.3.4. Conclusiones y Recomendaciones para la Seguridad en RoR	23
4.4. Limitaciones actuales y necesidad de marcos reutilizables	24
4.4.1. Problemas Recurrentes: El Factor Humano y la Complejidad del Ecosistema	24
4.4.2. Falta de Formación y Concienciación en Seguridad	26
4.4.3. Justificación del framework propuesto: Hacia Componentes Seguros y Reutilizables	27
<b>5. Definición de Requisitos y Componentes del Framework</b>	<b>29</b>
5.1. Requisitos No Funcionales	29
5.1.1. Seguridad	29
5.1.2. Rendimiento y Estabilidad	30
5.1.3. Usabilidad y Experiencia del Desarrollador	30
5.1.4. Mantenibilidad	31
5.1.5. Compatibilidad	31
5.2. Requisitos Funcionales	31
5.2.1. Autenticación Segura	31
5.2.2. Autorización y Control de Acceso	32
5.2.3. Validación y Sanitización de Entradas	32
5.2.4. Protección contra Vulnerabilidades Comunes	32
5.2.5. Gestión Segura de Dependencias	33
5.2.6. Gestión Segura de Secretos	33
5.3. Componentes del Framework	34

5.3.1. Gestión de Identidad y Acceso (IAM)	35
5.3.2. Protección de Datos y Entrada/Salida Segura	37
5.3.3. Fortalecimiento de la Seguridad Aplicativa	39
5.3.4. Operaciones de Desarrollo Seguro (SecDevOps)	41
5.3.5. Monitorización y Auditoría de Seguridad	43
<b>6. Diseño e Implementación: secure_framework</b>	<b>45</b>
6.1. Arquitectura del Framework	45
6.1.1. Justificación del Diseño Arquitectónico	45
6.1.2. Principios Arquitectónicos	46
6.1.3. Visión General de la Arquitectura	47
6.1.4. Componentes Arquitectónicos Principales y su Interacción	48
6.2. Implementación de Componentes Clave	49
6.2.1. Orquestación de la Instalación (InstallGenerator)	49
6.2.2. Gestión Segura de Secretos	50
6.2.3. Autenticación (Devise)	51
6.2.4. Autorización (Pundit)	52
6.2.5. Sanitización de Entradas (Sanitize)	53
6.2.6. Política de Seguridad de Contenido (CSP)	53
6.2.7. Cabeceras de Seguridad (Secure Headers)	54
6.2.8. Protección CSRF Reforzada	55
6.2.9. Auditoría de Dependencias (Bundler-Audit)	56
6.2.10. Logging de Seguridad (Lograge y Custom Logger)	56
6.3. Guía de Instalación y Uso Básico del Framework	58
6.3.1. Prerrequisitos	58
6.3.2. Proceso de Instalación	58
6.3.3. Configuración Inicial y Pasos Post-Instalación	59
<b>7. Validación y Pruebas</b>	<b>60</b>
7.1. Diseño de la Aplicación de Demostración	60
7.1.1. Introducción a demo_app	60
7.1.2. Arquitectura	61
7.1.3. Modelo de Datos y Base de Datos	62
7.1.4. Flujo de Autenticación y Gestión de Sesiones	63
7.1.5. Sistema de Autorización y Roles de Usuario	65
7.1.6. Rutas y Controladores Principales	67
7.2. Pruebas de Seguridad Aplicadas	68
7.2.1. Análisis Estático de Código (SAST)	68
7.2.2. Pruebas Funcionales de Seguridad con Rspec	70
7.3. Análisis y discusión de los Resultados	83
7.3.1. Discusión de los Resultados del Análisis Estático (SAST)	83
7.3.2. Discusión de los Resultados de las Pruebas Funcionales (RSpec)	85
7.3.3. Síntesis y Validación Holística del Framework	86
<b>8. Conclusiones y líneas futuras</b>	<b>87</b>
<b>Referencias bibliográficas</b>	<b>89</b>

## Glosario de Términos

- **Gema:** Es el formato estándar para empaquetar y distribuir librerías o aplicaciones en el ecosistema de Ruby. Cada gema contiene código y metadatos que facilitan su instalación y gestión de dependencias a través de la herramienta RubyGems.
- **Rails Engine:** Es una aplicación Rails en miniatura que puede ser montada dentro de otra aplicación (anfitriona) para proporcionar funcionalidades específicas. A diferencia de una gema, un Engine puede tener sus propios modelos, vistas y controladores de manera más estructurada, pero a menudo con un mayor acoplamiento.
- **ERB (Embedded Ruby):** Es el sistema de plantillas por defecto de Rails. Permite incrustar código Ruby directamente dentro de archivos HTML (con la extensión .html.erb). El servidor procesa este código para generar dinámicamente el contenido HTML final que se envía al navegador del cliente.
- **Cop:** En el contexto de la herramienta RuboCop, un "cop" es una regla de análisis individual que verifica una construcción de código específica. Estos cops se agrupan en departamentos (por ejemplo, "Security", "Style") para hacer cumplir las guías de estilo y las buenas prácticas.
- **Middleware (en el contexto de Rack/Rails):** Es un componente de software que se sitúa en una pila entre el servidor web y la aplicación Rails para procesar las peticiones y respuestas HTTP. Cada middleware tiene una responsabilidad específica, como gestionar sesiones, cookies o cabeceras de seguridad, actuando de forma secuencial.
- **Rake / Tarea Rake:** Rake es una utilidad de automatización de tareas escrita en Ruby, similar a make. Una "Tarea Rake" es un script definido que ejecuta una serie de pasos para realizar una acción específica, como migrar la base de datos o, en este caso, auditar dependencias.
- **Linter:** Un "linter" es una herramienta de análisis estático de código que se utiliza para detectar y reportar errores de programación, fallos de estilo, y construcciones de código sospechosas. Su objetivo es mejorar la calidad y la mantenibilidad del código fuente.



## 1. Introducción

En el panorama digital actual, el desarrollo de aplicaciones web se enfrenta a un entorno cada vez más complejo y dinámico en el que la seguridad constituye un aspecto crítico. La creciente sofisticación de los ataques, junto con el auge del desarrollo ágil y el despliegue continuo, ha generado una situación en la que las vulnerabilidades pueden introducirse en cualquier etapa del ciclo de vida del software. En este contexto, resulta fundamental integrar mecanismos de protección desde las fases más tempranas del desarrollo, adoptando un enfoque de "seguridad por diseño" que garantice la robustez de las aplicaciones frente a amenazas conocidas y emergentes. La seguridad ya no puede concebirse como una etapa posterior ni como una responsabilidad exclusiva de especialistas; debe formar parte del proceso de diseño, codificación y despliegue desde el inicio.

A pesar de la disponibilidad de documentación, herramientas y estándares orientados a mejorar la seguridad en el desarrollo web, en la práctica persiste una brecha significativa entre las buenas prácticas teóricas y su implementación efectiva en proyectos reales. Esta distancia se traduce con frecuencia en aplicaciones vulnerables a ataques comunes, tales como inyecciones SQL, cross-site scripting (XSS), falsificación de peticiones en sitios cruzados (CSRF) o gestión deficiente de sesiones. Una de las principales causas de esta situación radica en la complejidad inherente al desarrollo seguro, especialmente cuando los desarrolladores carecen de formación específica en ciberseguridad o trabajan bajo presiones de tiempo que priorizan la funcionalidad sobre la protección.

El problema se agrava cuando se observa que, en muchos entornos profesionales, las decisiones de diseño y arquitectura del software no consideran explícitamente la seguridad como un criterio prioritario. Esto conduce a la producción de código que, aunque funcional, incorpora riesgos que podrían haberse evitado mediante una planificación más rigurosa. En consecuencia, no solo se compromete la integridad y confidencialidad de los datos de los usuarios, sino que también se expone a las organizaciones a pérdidas económicas, sanciones legales y daños reputacionales. Esta realidad evidencia la necesidad urgente de facilitar la incorporación de medidas de seguridad efectivas en el flujo habitual de trabajo de los desarrolladores, sin imponer barreras técnicas excesivas.

Ante este panorama, el presente trabajo propone el desarrollo de un *framework* basado en Ruby on Rails que permita construir aplicaciones web seguras de forma modular, escalable y

accesible. La elección de Ruby on Rails responde a una combinación de factores técnicos y estratégicos. En primer lugar, se trata de un *framework* maduro, con una comunidad activa y un ecosistema rico en bibliotecas, herramientas y guías que promueven buenas prácticas desde su diseño. Rails fomenta una estructura clara del código, basada en el patrón Modelo-Vista-Controlador (MVC), que facilita la separación de responsabilidades y el mantenimiento del software. Además, incorpora por defecto una serie de mecanismos de protección frente a ataques comunes, como la sanitización automática de parámetros, la protección contra CSRF y la gestión segura de sesiones.

Desde una perspectiva tecnológica, Ruby on Rails permite una rápida iteración y desarrollo, lo que lo convierte en un entorno ideal para la creación de prototipos funcionales que puedan evolucionar hacia soluciones robustas en producción. Esta característica es especialmente relevante en proyectos donde se requiere validar enfoques de seguridad de forma ágil. Asimismo, su filosofía de “convención sobre configuración” reduce la probabilidad de errores humanos y facilita la adopción de patrones seguros sin necesidad de una configuración detallada por parte del desarrollador.

La motivación principal de este proyecto radica en la necesidad de proporcionar a los desarrolladores una herramienta que no solo les facilite la construcción de aplicaciones seguras, sino que además les eduque y guíe en la adopción de buenas prácticas. Muchas soluciones existentes requieren un conocimiento avanzado de conceptos de seguridad, lo que limita su adopción fuera de equipos especializados. En cambio, el enfoque aquí propuesto busca equilibrar la usabilidad con la seguridad, integrando componentes reutilizables y configurables que encapsulen funcionalidades críticas, como la autenticación, el control de acceso, la validación de entradas o el registro de actividades. De este modo, se pretende empoderar a desarrolladores generalistas para que puedan producir software más seguro sin necesidad de convertirse en expertos en ciberseguridad.

El objetivo general del proyecto es diseñar y desarrollar un *framework* extensible, basado en Ruby on Rails, que permita la creación de aplicaciones web seguras mediante la reutilización de componentes previamente diseñados con criterios de seguridad. La contribución esperada se sitúa en la intersección entre el desarrollo web y la ciberseguridad aplicada, proporcionando una solución que pueda integrarse en el flujo de trabajo habitual de los equipos de desarrollo. A nivel profesional, esta iniciativa se alinea con los principios de la ingeniería segura del software y responde a la creciente demanda de herramientas que integren seguridad de forma nativa, práctica y sostenible.

## 2. Objetivos

El objetivo general de este trabajo es **generar un *framework* seguro basado en Ruby on Rails** que facilite la construcción de aplicaciones web seguras, integrando por defecto mecanismos de protección frente a las vulnerabilidades más comunes, y permitiendo a los desarrolladores centrarse en la lógica de negocio sin necesidad de conocimientos avanzados en ciberseguridad.

A partir de este objetivo general, se establecen los siguientes objetivos específicos:

- **Realizar un estudio del arte** sobre la seguridad en Ruby on Rails, analizando sus mecanismos nativos de protección y evaluando herramientas y librerías complementarias existentes.
- **Definir la arquitectura** de un *framework* modular que incorpore componentes seguros reutilizables y configuraciones predefinidas que sigan las mejores prácticas recomendadas.
- **Construir el *framework*** utilizando Ruby on Rails, integrando de manera automática las medidas de seguridad identificadas.
- **Validar la seguridad y funcionalidad del *framework* desarrollado** mediante el diseño de una aplicación web demostrativa y la ejecución de pruebas de seguridad.

### 3. Metodología

Para el desarrollo del trabajo se ha adoptado una metodología estructurada en cuatro fases principales. A continuación, se describe la metodología empleada, estructurada en fases que reflejan la evolución del proyecto:

#### Fase 1: Estudio del estado del arte y tecnologías existentes

Esta fase tiene como finalidad establecer una base sólida de conocimiento respecto a las prácticas actuales de seguridad en el desarrollo web, centrándose especialmente en Ruby on Rails (RoR). Se realizará una revisión pormenorizada de las características de seguridad que ofrece este *framework*. Esta revisión permitirá identificar ventajas, limitaciones y oportunidades de mejora.

Las tareas específicas de esta fase son:

- **Revisión bibliográfica y documental** sobre las prácticas recomendadas de desarrollo seguro en Ruby on Rails, incluyendo fuentes académicas, guías de OWASP y documentación oficial del *framework*.
- **Análisis de herramientas y gemas existentes** en el ecosistema RoR con funcionalidades de seguridad (e.g., Devise, Pundit, Brakeman, etc.).
- **Identificación de brechas de seguridad comunes** en el desarrollo con RoR y estudio de patrones de mitigación aplicables.

Esta fase proporciona el input teórico y técnico necesario para abordar con criterio el diseño de un *framework* que priorice la seguridad desde su concepción.

#### Fase 2: Definición de requisitos y componentes del *framework*

Con los resultados del estudio preliminar, se procederá a definir los requisitos funcionales y no funcionales del *framework*. El enfoque estará centrado en garantizar seguridad desde el diseño (*security by design*), estableciendo una arquitectura modular y fácilmente integrable en proyectos Rails existentes.

Las actividades de esta fase comprenden:

- **Especificación de requisitos funcionales**, incluyendo la creación, gestión y validación de componentes seguros reutilizables (por ejemplo, autenticación, control de acceso, protección CSRF/XSS).
- **Definición de requisitos de seguridad**, alineados con las recomendaciones de OWASP y las vulnerabilidades identificadas en la fase anterior.
- **Propuesta inicial de componentes del *framework***, clasificándolos por su funcionalidad (autenticación, autorización, cifrado, *logging* seguro, etc.).
- **Elaboración de un modelo de arquitectura de alto nivel**, detallando la interacción entre módulos y la forma en que se integran en una aplicación típica RoR.

Esta fase constituye la base conceptual del desarrollo e implementación posterior.

### Fase 3: Diseño e implementación del *framework*

Esta fase se subdivide en dos tareas principales, que pueden realizarse de forma parcialmente paralela pero coordinada:

- **Diseño del *framework***: En esta tarea se formaliza la arquitectura definida anteriormente. Se establecerán los esquemas de integración con aplicaciones Rails, las interfaces públicas del *framework* y las dependencias necesarias. Se utilizará un enfoque de diseño modular, orientado a componentes y siguiendo principios SOLID.
- **Desarrollo e implementación**: Una vez definidos los aspectos arquitectónicos, se procederá a la implementación iterativa de los componentes definidos. La prioridad se asignará a aquellos con mayor impacto en la seguridad (como el sistema de autenticación y control de sesiones). Se emplearán buenas prácticas de codificación segura, revisiones continuas y pruebas unitarias desde las primeras etapas del desarrollo.

Ambas tareas se retroalimentan: el diseño puede ajustarse en función de los descubrimientos durante la implementación, y viceversa.

### Fase 4: Validación y pruebas de seguridad

Para asegurar la robustez y fiabilidad del *framework*, se diseñará un conjunto de pruebas exhaustivas orientadas a la validación funcional y, especialmente, a la verificación de las propiedades de seguridad. Esta fase incluye:

- **Pruebas** sobre los módulos desarrollados, incluyendo test unitarios, de integración y funcionales.
- **Evaluación mediante herramientas de análisis estático y dinámico**, como Brakeman para la detección temprana de vulnerabilidades en el código fuente.
- **Pruebas de caja negra** mediante escenarios simulados de ataque para validar la resistencia frente a amenazas como inyecciones SQL, XSS, CSRF o elevación de privilegios.
- **Prueba de integración con una aplicación Rails de ejemplo**, que permitirá verificar la interoperabilidad y facilidad de adopción del *framework* por parte de desarrolladores externos.

Los resultados de esta fase permitirán corregir defectos y realizar mejoras finales antes de la entrega del producto.

## 4. Estado del Arte y Justificación del Trabajo

En esta sección se presenta un análisis detallado del estado actual en materia de desarrollo seguro de aplicaciones web, con especial atención al enfoque de seguridad en Ruby on Rails. Se examinan los principios fundamentales del enfoque *Security by Design*, la evolución del *framework* atendiendo a aspectos de seguridad y protección, así como herramientas, gemas y buenas prácticas empleadas actualmente. Finalmente, se justifica la necesidad del trabajo propuesto en base a las limitaciones observadas en el ecosistema actual.

### 4.1. Introducción al desarrollo seguro de aplicaciones web

En la última década, el desarrollo seguro de aplicaciones web ha dejado de ser una preocupación secundaria para convertirse en un componente crítico del ciclo de vida del software. La creciente sofisticación de las amenazas, la expansión del perímetro de ataque debido a arquitecturas distribuidas y la presión regulatoria (como el RGPD [1] o la ISO/IEC 27001 [2]) obligan a incorporar medidas de seguridad desde las primeras fases del desarrollo. Este enfoque proactivo se conoce como *security by design* y se basa en integrar controles de seguridad durante todo el ciclo de vida del software (SDLC, por sus siglas en inglés), desde la especificación de requisitos hasta la fase de mantenimiento.

Uno de los pilares fundamentales para asegurar las aplicaciones web modernas es la identificación y mitigación sistemática de vulnerabilidades. En este contexto, la **OWASP Foundation** ha establecido una referencia global con su informe *OWASP Top 10* [3], el cual agrupa las diez principales categorías de riesgos de seguridad en aplicaciones web. La versión más reciente, **OWASP Top 10 – 2021**, agrupa las amenazas de la siguiente manera:

**Tabla 1.** OWASP Top 10 – 2021.

Categoría OWASP 2021	Descripción
A01:2021 - Pérdida de Control de Acceso	Pérdida de control sobre permisos y roles.
A02:2021 - Fallas Criptográficas	Gestión incorrecta del cifrado y datos sensibles.
A03:2021 - Inyección	Inyecciones SQL, NoSQL, OS y similares.
A04:2021 - Diseño Inseguro	Ausencia de patrones de diseño seguros.
A05:2021 - Configuración de Seguridad Incorrecta	Configuraciones erróneas o inseguras.
A06:2021 - Componentes Vulnerables y Desactualizados	Uso de componentes obsoletos con CVEs activos.
A07:2021 - Fallas de Identificación y Autenticación	Gestión deficiente de autenticación o sesión.
A08:2021 - Fallas en el Software y en la Integridad de los Datos	Fallos al validar integridad en dependencias.

A09:2021 - Fallas en el Registro y Monitoreo	Falta de monitoreo efectivo de seguridad.
A10:2021 - Falsificación de Solicitudes del Lado del Servidor	Manipulación de peticiones del servidor.

Estas categorías no sólo sirven como guía de evaluación, sino también como fundamento técnico para definir requisitos de seguridad concretos en el desarrollo.

## Security by Design: más allá de las buenas prácticas

El concepto de *Security by Design* implica adoptar una mentalidad defensiva en cada fase del SDLC. Más que aplicar controles de forma reactiva, este enfoque propone anticipar las amenazas e implementar mecanismos preventivos y correctivos desde la concepción de la arquitectura. Esto incluye, entre otras prácticas:

- Minimización de superficie de ataque (*attack surface minimization*).
- Principio del menor privilegio.
- Uso de controles de acceso explícitos.
- Validación estricta de entradas.
- Diseño modular con aislamiento de responsabilidades (por ejemplo, separación entre lógica de negocio, autenticación y presentación).
- Uso de bibliotecas con seguridad comprobada y políticas de actualización continua.

Investigaciones como **Towards the integration of security practices in agile software development: A systematic mapping review** [4] demuestran que la integración estructurada de prácticas de seguridad en metodologías ágiles, si bien desafiante, es esencial para reducir la exposición a vulnerabilidades. El estudio evidencia que muchas organizaciones adoptan medidas reactivas ante incidentes, lo que incrementa el coste y el impacto de los fallos. En contraste, las metodologías que promueven la incorporación temprana de requisitos de seguridad (como la definición de *user stories* seguras, revisiones continuas de código con enfoque en seguridad o la automatización de pruebas de vulnerabilidad) muestran una mejora significativa en la calidad del software y su resiliencia ante amenazas.

## El rol del desarrollador: entre la abstracción y la responsabilidad

Un desafío relevante en el desarrollo seguro es el equilibrio entre productividad y seguridad. Lenguajes y *frameworks* modernos (como Ruby on Rails, Django, Laravel, etc.) ofrecen abstracciones potentes que facilitan el desarrollo rápido (*rapid prototyping*), pero esta



velocidad puede enmascarar riesgos si no se entienden los detalles de seguridad que subyacen bajo dichas abstracciones. Por ejemplo, prácticas como el uso incorrecto de funciones *helper* para sanitización, la sobreconfianza en autenticación por defecto o una configuración inadecuada de *middleware* pueden introducir vulnerabilidades graves incluso en aplicaciones bien estructuradas.

La responsabilidad del desarrollador no debe limitarse a implementar funcionalidades, sino a entender las implicaciones de seguridad de cada decisión técnica. Por tanto, existe una necesidad creciente de herramientas y *frameworks* que no sólo permitan desarrollar aplicaciones seguras, sino que induzcan por defecto patrones seguros de diseño y configuración. En este sentido, la propuesta de este trabajo (un *framework* basado en componentes seguros reutilizables para Ruby on Rails) se alinea con esta visión de seguridad integrada, guiada y automatizada.

## 4.2. Ruby on Rails: Evolución de la seguridad

Desde su lanzamiento en 2004 por David Heinemeier Hansson, Ruby on Rails (RoR) [5] ha evolucionado significativamente no solo como *framework* de desarrollo rápido de aplicaciones web, sino también como plataforma consciente de la seguridad. A medida que la popularidad de Rails crecía, también lo hacían los ataques dirigidos a aplicaciones construidas sobre este *framework*, lo que obligó a su comunidad a adoptar un enfoque progresivamente más robusto frente a las vulnerabilidades. Esta evolución ha estado marcada por una serie de hitos clave, tanto en términos de protección incorporada como en respuesta a incidentes de seguridad relevantes.

### Seguridad desde el diseño: filosofía de Rails

Una de las características distintivas de Ruby on Rails es su adhesión a la **convención sobre configuración** (*convention over configuration*) [6], que permite estandarizar comportamientos por defecto. Esta filosofía también se ha trasladado al ámbito de la seguridad, promoviendo un modelo de seguridad “por defecto seguro” (*secure by default*). Por ejemplo, desde versiones tempranas, Rails escapaba automáticamente el contenido HTML para prevenir ataques XSS (Cross-Site Scripting), una práctica que *frameworks* contemporáneos no adoptaban inicialmente.

Sin embargo, este enfoque por sí solo no fue suficiente para mitigar vulnerabilidades estructurales, especialmente aquellas derivadas del modelo de *binding* automático de parámetros, lo que derivó en problemas críticos como la **vulnerabilidad de asignación masiva (Mass Assignment)** [7], uno de los fallos más explotados en aplicaciones Rails previas a 2013.

#### 4.2.1. Evolución por Versiones

##### Los Primeros Pasos: Ruby on Rails 1.x y 2.x (2004-2008)

En los primeros años de Ruby on Rails, el panorama de la seguridad web se centraba en vulnerabilidades fundamentales como la inyección SQL y el Cross-Site Scripting (XSS).

Un reconocimiento temprano de los riesgos asociados con el manejo dinámico de parámetros en las aplicaciones web se manifestó en la introducción de **attr\_protected** y **attr\_accessible** en Ruby on Rails 2. La asignación masiva, que permite la actualización simultánea de múltiples atributos de un objeto utilizando un hash de parámetros proveniente de la entrada del usuario, presentaba un riesgo significativo. Usuarios malintencionados podían incluir parámetros inesperados en sus peticiones para modificar atributos sensibles a los que no deberían tener acceso, como establecer una bandera “is\_admin” en “true”. Para mitigar este riesgo, Rails 2 introdujo **attr\_protected**, que permitía a los desarrolladores especificar una lista de atributos que no podían ser asignados masivamente, y **attr\_accessible**, que permitía definir una lista de atributos que sí podían serlo. La evolución de un enfoque de lista negra (**attr\_protected**) a una lista blanca (**attr\_accessible**) reflejó una comprensión creciente de los principios de seguridad, ya que la lista blanca generalmente se considera más segura por defecto.

Además, Rails 2 incorporó medidas iniciales para la protección contra ataques de Cross-Site Request Forgery (CSRF). Los ataques CSRF explotan la confianza entre una aplicación web y sus usuarios, engañándolos para que realicen acciones no deseadas en un sitio de confianza cuando están autenticados. Rails 2 introdujo **protect\_from\_forgery**, que, al incluirse en el controlador de la aplicación, genera un token único que se incrusta en cada formulario y petición AJAX. El servidor verifica este token para asegurarse de que la petición se originó en la propia aplicación y no en un sitio de terceros malicioso. Esta temprana adopción de la protección CSRF demostró una mentalidad de seguridad dentro del equipo de desarrollo de Rails desde las etapas formativas del *framework*.

A pesar de estas medidas iniciales, las versiones 1.x y 2.x de Ruby on Rails no estuvieron exentas de vulnerabilidades. Durante esta época se identificaron varias vulnerabilidades (CVE), entre las que se incluyen problemas de fijación de sesión (CVE-2007-6077, CVE-2007-5380), acceso arbitrario a archivos (CVE-2007-5379), XSS (CVE-2007-3227) y potencial ejecución remota de código (CVE-2006-4112, CVE-2006-4111), así como la omisión de CSRF (CVE-2008-7248), inyección de encabezado (CVE-2008-5189) e inyección SQL (CVE-2008-4094). Estos CVE sugieren que, si bien Rails 2 contaba con características de seguridad iniciales, también estuvo sujeto a diversas vulnerabilidades que se descubrieron y abordaron con el tiempo.

### **Consolidación y Conciencia: Ruby on Rails 3.x (2010-2013)**

El período de Ruby on Rails 3.x estuvo marcado por una creciente conciencia de los problemas de seguridad web en general, lo que influyó en el desarrollo del *framework*. Rails 3 continuó construyendo sobre las bases de seguridad establecidas en versiones anteriores, al tiempo que introdujo mejoras significativas y abordó nuevas amenazas emergentes.

Las mejoras en la protección CSRF fueron una característica clave de Rails 3. Si bien el mecanismo básico de **protect\_from\_forgery** se introdujo en Rails 2, la serie 3.x probablemente refinó su implementación y facilitó su habilitación por defecto. Un avance aún más significativo fue la adopción de HTML escaping por defecto en las plantillas ERB a partir de Rails 3.0. Esta medida representó una mejora sustancial en la seguridad, ya que redujo en gran medida el riesgo de que los desarrolladores introdujeran inadvertidamente vulnerabilidades XSS al olvidar escapar manualmente los datos proporcionados por el usuario en sus vistas. Este cambio reflejó una creciente comprensión dentro de la comunidad de Rails de la importancia de los valores predeterminados seguros en los *frameworks* de desarrollo web. Además, la corrección de la vulnerabilidad de inyección de encabezado en el método **redirect\_to** en la versión 2.1.2 se mantuvo en la serie Rails 3.x, asegurando la protección contra este tipo de ataque.

A pesar de estas mejoras, la serie 3.x de Ruby on Rails estuvo sujeta a un número considerable de vulnerabilidades reportadas, como lo demuestran los diversos CVE asociados con este período. Estos incluyeron vulnerabilidades relacionadas con fallas de formato de cadena en Action Mailer (CVE-2013-4389), problemas de divulgación de existencia de archivos en Action Pack (CVE-2014-7818, CVE-2014-7829) y vulnerabilidades XSS en los

helpers **sanitize** (CVE-2013-1857) y **number\_to\_currency** (OSVDB-100524, CVE-2014-0081). También se reportaron vulnerabilidades de fuga de información en Action View (CVE-2016-2097, CVE-2016-0752), una vulnerabilidad XSS en el helper **simple\_format** (OSVDB-100527) y una corrección incompleta para los riesgos de generación de consultas inseguras (OSVDB-100528). Además, se identificaron posibles vulnerabilidades de ejecución remota de código (CVE-2016-2098) y vulnerabilidades de denegación de servicio (DoS) en Action View (OSVDB-100525) y al usar **render :text** (CVE-2014-0082). También se descubrieron vulnerabilidades de recorrido de directorios (CVE-2014-0130), vulnerabilidades de inyección SQL en Active Record (CVE-2014-3482) y potencial ejecución remota de código a través de la serialización YAML (CVE-2013-0277), así como una omisión para **attr\_protected** (CVE-2013-0276). Active Support también se vio afectado por vulnerabilidades de análisis XML (CVE-2013-1856, CVE-2015-3227). Es importante destacar que también se encontraron vulnerabilidades en gemas utilizadas con Rails 3.x, como **i18n** (CVE-2014-10077), **json** (CVE-2020-10663), **mail** (CVE-2012-2140, CVE-2012-2139, CVE-2015-9097) y **rack** (CVE-2013-0183, CVE-2013-0262, CVE-2019-16782, CVE-2018-16471, CVE-2015-3225).

La gran cantidad y variedad de CVE reportados para Rails 3.x indican un *framework* en maduración bajo un escrutinio cada vez mayor. Las vulnerabilidades abarcan diferentes componentes, lo que destaca la complejidad de asegurar un *framework* web de pila completa. La aparición de vulnerabilidades relacionadas con gemas también subraya la importancia de la gestión de dependencias en la seguridad. Este período probablemente representa una curva de aprendizaje significativa para la comunidad de Rails, lo que lleva a una comprensión más profunda de las posibles trampas de seguridad y al desarrollo de prácticas de seguridad más sólidas en versiones posteriores.

### El Enfoque en la Protección de Datos: Ruby on Rails 4.x (2013-2016)

Ruby on Rails 4.x marcó un punto de inflexión significativo en la evolución de la seguridad del *framework* con la introducción de Strong Parameters. Esta característica fue una respuesta directa a las vulnerabilidades de asignación masiva que habían afectado a versiones anteriores. La filosofía detrás de Strong Parameters era "asumir inseguro hasta que se demuestre lo contrario". Esto significaba que Rails marcaba todos los parámetros entrantes como no permitidos para la asignación masiva hasta que el desarrollador los permitiera explícitamente utilizando los métodos **require** y **permit**. El incidente de GitHub en 2012 [8]

sirvió como un ejemplo destacado de una vulnerabilidad de asignación masiva que probablemente influyó en este cambio fundamental en la forma en que Rails manejaba la entrada del usuario.

Además de Strong Parameters, Rails 4.x introdujo otras mejoras de seguridad. Rails 4.0 añadió por defecto nuevos encabezados de seguridad: **X-Frame-Options** para prevenir ataques de clickjacking, **X-XSS-Protection** para habilitar los filtros XSS del navegador y **X-Content-Type-Options** para prevenir el MIME sniffing. En Rails 4.1, la protección CSRF se extendió a las peticiones GET con respuestas JavaScript, abordando una posible vulnerabilidad donde sitios maliciosos podían intentar ejecutar URL JavaScript para extraer datos sensibles. Rails 4.1 también introdujo **config/secrets.yml** como un lugar dedicado para almacenar información sensible como **secret\_key\_base** y claves de API, promoviendo una mejor organización y un manejo potencialmente más seguro de las credenciales. Además, se introdujeron los Message Verifiers en Rails 4.1, que permitían a los desarrolladores generar y verificar mensajes firmados, útiles para transportar de forma segura datos sensibles como los tokens "recordarme". La adición de encabezados de seguridad por defecto demostró una creciente conciencia de las estrategias de defensa en profundidad, proporcionando una capa adicional de protección contra ataques comunes basados en el navegador sin requerir una implementación explícita por parte del desarrollador en cada caso.

A pesar de estas significativas mejoras, Rails 4.x no fue inmune a las vulnerabilidades reportadas. Se informaron vulnerabilidades de ataque de tiempo en la autenticación básica (CVE-2015-7576), vulnerabilidades de fuga de información en Action View (CVE-2016-2097, CVE-2016-0752), vulnerabilidades XSS en Action View (CVE-2016-6316), vulnerabilidades de fuga de objetos y DoS en Action Pack (CVE-2016-0751, CVE-2015-7581) y posibles vulnerabilidades de ejecución remota de código en Action Pack (CVE-2016-2098). También se destacaron vulnerabilidades de omisión de atributos anidados en Active Record (CVE-2015-7577), vulnerabilidades DoS en Active Support (CVE-2015-3227) y vulnerabilidades XSS en Rack (CVE-2018-16471).

### **Seguridad Moderna: Ruby on Rails 5.x y 6.x (2016-2021)**

La evolución de las prácticas y características de seguridad continuó en Ruby on Rails 5.x y 6.x. En las nuevas aplicaciones de Rails 5 se habilitó por defecto HSTS (HTTP Strict Transport Security) con la directiva **IncludeSubdomains**, lo que supuso una postura más

filme en la aplicación de conexiones seguras. Rails 5 también facilitó la activación de **config.force\_ssl** y **config.ssl\_options** y añadió tokens CSRF por formulario. Rails 6 introdujo **ActionDispatch::HostAuthorization** para proteger contra ataques de DNS rebinding y añadió metadatos de propósito y caducidad en las cookies firmadas/encryptadas, además de asegurar que solo se emitiera un valor de encabezado nonce Content-Security-Policy por petición. El enfoque en características como HSTS por defecto y **HostAuthorization** indicó un enfoque proactivo para mitigar vectores de ataque más sofisticados y promover configuraciones seguras desde el principio.

Sin embargo, las series 5.x y 6.x también estuvieron sujetas a una amplia gama de vulnerabilidades reportadas. Estas incluyeron vulnerabilidades XSS en los helpers de escape literal de JavaScript de ActionView (CVE-2020-5267), vulnerabilidades relacionadas con **redirect\_to** y **polymorphic\_url** con entrada no confiable (CVE-2021-22885), vulnerabilidades CSRF en rails-ujs (CVE-2020-8167), vulnerabilidades de divulgación de contenido de archivos en Action View (CVE-2019-5418), vulnerabilidades de omisión de Strong Parameters (CVE-2020-8164), vulnerabilidades DoS en la autenticación de tokens de Action Controller (CVE-2021-22904) y el adaptador PostgreSQL de Active Record (CVE-2021-22880), vulnerabilidades en Active Storage (CVE-2018-16477, CVE-2022-21831, CVE-2020-8162) y Active Support (CVE-2020-8165), vulnerabilidades ReDoS en Action Dispatch y Active Support (CVE-2023-22792, CVE-2023-22795, CVE-2023-22796), XSS basado en DOM en rails-ujs (CVE-2023-23913), vulnerabilidades de divulgación de archivos en Active Support (CVE-2023-38037) y XSS en **SafeBuffer#bytesplice** (CVE-2023-28120). También se destacaron vulnerabilidades en gemas como Nokogiri, Puma (CVE-2021-29509, CVE-2021-41136, CVE-2022-24790, CVE-2022-23634, CVE-2023-40175, CVE-2024-21647), Rack (CVE-2022-30123, CVE-2022-30122, CVE-2020-8184), rails-html-sanitizer y Railties (CVE-2019-5420).

La aparición de vulnerabilidades como ReDoS destaca la importancia de las prácticas de codificación segura incluso en áreas aparentemente benignas como el uso de expresiones regulares.

### **El Presente y el Futuro: Ruby on Rails 7.x y 8.x (2021-Presente)**

Las versiones más recientes de Ruby on Rails, 7.x y 8.x, demuestran un compromiso continuo con la seguridad. Rails 8.0 introdujo un generador de autenticación predeterminado

y estableció **Regexp.timeout** en 1 segundo por defecto para prevenir ataques ReDoS. Rails 7.2 incluyó Brakeman por defecto en las nuevas aplicaciones. Rails también cuenta con una configuración predeterminada de encabezados de seguridad, el método **deep\_munge** para prevenir la generación de consultas inseguras y recomienda el middleware **ActionDispatch::HostAuthorization**. Las versiones recientes mantienen el soporte para correcciones de seguridad. Este enfoque en incluir herramientas de seguridad como Brakeman por defecto y establecer valores predeterminados sensatos como **Regexp.timeout** indica un mayor énfasis en la "seguridad por defecto" dentro del *framework* Rails, con el objetivo de facilitar a los desarrolladores la creación de aplicaciones seguras desde el principio.

Sin embargo, incluso las últimas versiones de Rails son susceptibles a vulnerabilidades de seguridad, lo que enfatiza que la seguridad es una batalla continua. Se han informado vulnerabilidades ReDoS recientes en Action Mailer y Action Text (CVE-2024-47889, CVE-2024-47888, CVE-2024-47887, CVE-2024-41128), una posible omisión de Content Security Policy en Action Dispatch (CVE-2024-54133), una fuga de información de sesión sensible en Active Storage (CVE-2024-26144), una vulnerabilidad XSS en la traducción de Action Controller (CVE-2024-26143) y una vulnerabilidad ReDoS en el análisis del encabezado Accept en Action Dispatch (CVE-2024-26142). También se han encontrado vulnerabilidades en rails-html-sanitizer (CVE-2024-53989, CVE-2024-53988, CVE-2024-53987, CVE-2024-53986, CVE-2024-53985), un problema de HTML no saneado en ActionText ContentAttachment (CVE-2024-32464) y encabezados de seguridad faltantes en Action Pack en respuestas no HTML (CVE-2024-28103).

### 4.3. Herramientas, gemas y prácticas actuales en seguridad con RoR

Esta sección profundiza en las herramientas, gemas y prácticas contemporáneas que los desarrolladores pueden emplear para fortificar sus aplicaciones RoR contra una variedad de amenazas cibernéticas. El objetivo es proporcionar una visión concisa, profesional, profunda y técnica del estado actual de la seguridad en el ecosistema de RoR.



### 4.3.1. Herramientas de Análisis Estático de Seguridad (SAST)

Las herramientas de Análisis Estático de Seguridad de Aplicaciones (SAST) examinan el código fuente, bytecode o código binario de una aplicación en busca de patrones que puedan indicar vulnerabilidades de seguridad, sin necesidad de ejecutar la aplicación. Para Ruby on Rails, existen herramientas SAST específicas que comprenden las convenciones y la estructura del *framework*, permitiendo una detección más precisa de problemas de seguridad comunes.

#### Brakeman: Escáner de Vulnerabilidades Específico para Rails

Brakeman [9][10] analiza el código fuente directamente y puede detectar una amplia gama de vulnerabilidades comunes en aplicaciones Rails. Algunos de los tipos de advertencias que Brakeman puede generar incluyen:

- **Inyección de SQL (SQL Injection):** Detecta el uso inseguro de parámetros en consultas Active Record, como la interpolación directa de cadenas en métodos **where** o **find\_by\_sql**.
- **Cross-Site Scripting (XSS):** Identifica lugares donde la salida no sanitizada o el uso de métodos como **raw** o **html\_safe** podrían permitir la inyección de scripts.
- **Inyección de Comandos (Command Injection):** Señala el uso de métodos como **system**, **eval**, o backticks con entrada controlada por el usuario.
- **Redirecciones Inseguras (Unsafe Redirects):** Advierte sobre redirecciones que utilizan parámetros del usuario sin una validación adecuada, lo que podría llevar a phishing o XSS.
- **Renderizado Dinámico de Rutas (Dynamic Render Paths):** Detecta si la entrada del usuario se utiliza para determinar la plantilla a renderizar, lo que podría exponer vistas no deseadas.
- **Asignación Masiva (Mass Assignment):** Como nombramos anteriormente, en versiones antiguas de Rails, esto era un gran problema. Brakeman puede identificar el uso de **attr\_accessible** o la falta de Strong Parameters en versiones más nuevas donde aún podría ser un riesgo si se configura incorrectamente.
- **Falsificación de Solicitudes entre Sitios (CSRF):** Verifica la presencia de protecciones CSRF.



- **Fugas de Información (Information Disclosure):** Puede detectar la exposición de archivos sensibles o información de configuración.
- **Deserialización Insegura (Unsafe Deserialization):** Advierte sobre el uso de métodos como **YAML.load** con datos no confiables.
- **Configuraciones de Seguridad Débiles:** Problemas con la configuración de cookies de sesión, falta de encabezados de seguridad, etc..
- **Dependencias No Mantenidas:** Puede señalar el uso de dependencias que ya no reciben actualizaciones.

Brakeman asigna un **nivel de confianza** (High, Medium, Weak) a cada advertencia, indicando la probabilidad de que la advertencia represente un problema real y su severidad potencial. Esta clasificación ayuda a priorizar los esfuerzos de remediación.

Además, esta herramienta permite generar informes en varios formatos (texto, HTML, JSON, CSV, etc.). El formato HTML es particularmente útil para la revisión manual, ya que proporciona detalles sobre la advertencia, el archivo y la línea de código afectada, y enlaces a documentación relevante.

### **RuboCop y Extensiones de Seguridad (rubocop-rails, rubocop-security)**

RuboCop [11] es una herramienta de análisis estático de código Ruby (linter y formateador) muy popular, enfocada principalmente en mantener la consistencia del estilo de código y adherirse a las guías de estilo de la comunidad Ruby. Si bien su propósito principal no es la detección de vulnerabilidades de seguridad de la misma manera que Brakeman, RuboCop puede contribuir a la seguridad a través de:

- **Cops de Seguridad Incorporados:** El propio RuboCop incluye un departamento de "Security" con cops que verifican construcciones de código conocidas por estar asociadas con posibles problemas de seguridad.
- **Extensiones Específicas:** Existen extensiones para RuboCop que añaden cops adicionales, algunas de las cuales pueden tener un enfoque en Rails o en aspectos de seguridad.

El conjunto de cops de seguridad estándar de RuboCop (**Security/**) incluye verificaciones para:

- **Security/Eval**: Desaconseja el uso de **eval**, que puede ser peligroso si se usa con entrada no confiable.
- **Security/Open**: Advierte contra el uso de **Kernel#open** con entrada de usuario si esta comienza con una barra vertical (`|`), ya que puede llevar a la ejecución de comandos.
- **Security/JSONLoad**: Prefiere **JSON.parse** sobre **JSON.load** para evitar la posible creación de objetos arbitrarios si la entrada JSON está manipulada (riesgo de deserialización insegura en ciertos contextos).
- **Security/MarshalLoad**: Advierte contra el uso de **Marshal.load** o **Marshal.restore** con datos no confiables, ya que puede llevar a la ejecución remota de código o DoS.
- **Security/YAMLLoad**: Prefiere **YAML.safe\_load** sobre **YAML.load** para prevenir la deserialización insegura de objetos YAML.
- **Security/IOMethods**: Verifica el uso de métodos de IO que podrían ser riesgosos si se usan con rutas de archivo controladas por el usuario sin la debida sanitización (aunque este cop puede ser más genérico).

**rubocop-rails**: Esta es una extensión oficial de RuboCop que añade cops específicos para las mejores prácticas y convenciones de Rails. Aunque su enfoque principal es el estilo y las prácticas de Rails, algunas de estas pueden tener implicaciones indirectas en la seguridad al promover un código más claro y menos propenso a errores. No parece tener un conjunto extenso de cops dedicados a la seguridad como lo haría una gema **rubocop-security**. La documentación de **rubocop-rails** no lista explícitamente "Security Cops" como una categoría principal de sus cops, a diferencia de RuboCop base que sí tiene un departamento **Security**.

Aunque RuboCop no reemplaza a una herramienta SAST dedicada a seguridad como Brakeman para aplicaciones Rails, complementa el análisis al enfocarse en patrones de código a un nivel más granular que pueden ser precursores de vulnerabilidades o malas prácticas de seguridad. Su capacidad para auto-corrregir muchas ofensas de estilo también ayuda a mantener un código más limpio y legible, lo que indirectamente beneficia la seguridad al facilitar las revisiones y la detección de errores.

### 4.3.2. Gemas de Seguridad para Ruby on Rails

El ecosistema de Ruby on Rails ofrece una variedad de gemas robustas que simplifican la implementación de características de seguridad esenciales.

#### Devise

Devise [12] es una solución de autenticación completa y flexible para aplicaciones Rails. Proporciona una suite completa de herramientas para manejar la autenticación de usuarios, el registro, la gestión de contraseñas y más. Devise utiliza bcrypt para el cifrado de contraseñas, tokens cifrados después del inicio de sesión y ofrece módulos para características como bloqueo de cuentas, confirmación de correo electrónico, gestión de sesiones (incluyendo timeouts) y funcionalidad de "remember me". Si bien Devise es una solución robusta, algunos proyectos con requisitos de autenticación muy específicos podrían considerar alternativas como Rodauth o incluso implementar una solución personalizada, aunque esto último conlleva un mayor riesgo de introducir vulnerabilidades. Existen extensiones de Devise, como **devise-security**, que añaden características de seguridad adicionales como caducidad de contraseñas, validación de contraseñas más robusta, limitación de sesiones y preguntas de seguridad. Para una autenticación de API en Rails, se recomienda considerar soluciones basadas en tokens como JWT en lugar de depender directamente de las funcionalidades orientadas a la web de Devise.

#### CanCanCan

CanCanCan [13] es una gema de autorización para Ruby y Ruby on Rails que restringe los recursos a los que un usuario determinado puede acceder. Permite definir permisos basados en los roles de los usuarios y las acciones que pueden realizar, centralizando la lógica de permisos en un único lugar para facilitar el mantenimiento y las pruebas. Los permisos se definen en una clase **Ability**, donde se especifican las acciones permitidas en función del rol del usuario. CanCanCan proporciona helpers para verificar permisos en vistas y controladores (**can?**, **cannot?**, **authorize!**) y facilita la obtención de todos los objetos a los que el usuario tiene acceso (**accessible\_by**). Se utiliza comúnmente para implementar control de acceso basado en roles (RBAC) y puede adaptarse a escenarios de multi-tenancy y listas de control de acceso (ACLs).

## Pundit

Pundit [14] es otra gema de autorización para Ruby on Rails que promueve el uso de clases Ruby simples y patrones de diseño orientados a objetos para construir un sistema de autorización robusto y escalable. A diferencia de CanCanCan, que se basa en una definición centralizada de habilidades, Pundit utiliza "policies" que son clases asociadas a modelos específicos y definen las reglas de acceso para las acciones correspondientes. Pundit proporciona un helper **authorize** en los controladores para verificar los permisos y un helper **policy\_scope** para filtrar las consultas de datos en función de los permisos del usuario. También incluye mecanismos para asegurar que las policies y los scopes se utilicen correctamente (**verify\_authorized**, **verify\_policy\_scoped**). Pundit ofrece una mayor granularidad en la definición de permisos a nivel de método dentro de las policies, lo que lo convierte en una opción adecuada para aplicaciones con requisitos de autorización complejos.

### 4.3.3. Prácticas Actuales de Seguridad en RoR

Más allá de las herramientas y gems, la adopción de prácticas de desarrollo seguras es fundamental para proteger las aplicaciones Ruby on Rails.

#### Prevención de Inyección SQL

Para prevenir este tipo de vulnerabilidad en RoR, es crucial utilizar las funcionalidades de Active Record para construir consultas, como **where** con hashes o arrays de condiciones, en lugar de interpolar directamente la entrada del usuario en cadenas SQL. Métodos como **find\_by** y los dynamic finders de Active Record utilizan consultas parametrizadas de forma predeterminada, lo que ayuda a prevenir la inyección SQL. Se debe evitar el uso de cadenas SQL crudas a menos que sea absolutamente necesario, y en esos casos, la entrada del usuario debe ser cuidadosamente sanitizada.

#### Prevención de Cross-Site Scripting (XSS)

Ruby on Rails proporciona protección automática contra XSS al escapar todo el contenido que se inserta en las vistas HTML de forma predeterminada. Sin embargo, se debe tener precaución con métodos como **html\_safe** y **raw**, que desactivan este escaping y pueden introducir vulnerabilidades si se utilizan con datos no confiables. Se recomienda utilizar Content Security Policy (CSP) para controlar los recursos que el navegador puede cargar para

la página, lo que puede ayudar a mitigar los ataques XSS. Si se necesita permitir HTML proporcionado por el usuario, se debe utilizar el método **sanitize** con precaución, especificando las etiquetas permitidas, en lugar de **html\_safe**.

### Prevención de Cross-Site Request Forgery (CSRF)

Rails incluye protección CSRF de forma predeterminada. Es fundamental asegurarse de que **protect\_from\_forgery** esté habilitado en el **ApplicationController** y de incluir la etiqueta `<%= csrf_meta_tags %>` en la sección **<head>** de las plantillas de la aplicación. Rails genera y verifica automáticamente tokens de autenticidad para todas las peticiones no-GET, asegurando que la petición provenga de la propia aplicación.

### Prevención de Deserialización Insegura

La deserialización insegura ocurre cuando una aplicación acepta datos serializados de fuentes no confiables y los deserializa sin una validación adecuada, lo que puede llevar a la ejecución remota de código. En Ruby, la librería **Marshal** se utiliza comúnmente para la serialización. Se debe evitar la deserialización de datos no confiables utilizando **Marshal.load** o **Marshal.restore**. Si es necesario manejar datos serializados, se deben considerar alternativas más seguras y aplicar estrictas validaciones sobre los datos antes de deserializarlos.

### Autenticación y Gestión de Sesiones Seguras

Utilizar una gema de autenticación robusta como Devise es una práctica recomendada. Es esencial forzar el uso de HTTPS en toda la aplicación para proteger las sesiones de la interceptación. Las cookies de sesión deben configurarse con los flags **secure** (para que solo se transmitan a través de HTTPS), **httponly** (para evitar el acceso por scripts del lado del cliente, mitigando ataques XSS) y **samesite** (para prevenir ataques CSRF). Para información sensible, se puede considerar el almacenamiento de sesiones en el servidor en lugar de depender únicamente de las cookies. También es importante invalidar las sesiones antiguas para evitar su reutilización maliciosa.

### Control de Acceso Basado en Roles y Permisos (RBAC)

Implementar la autorización utilizando gemas como CanCanCan o Pundit es crucial para restringir el acceso a recursos y funcionalidades basándose en los roles de los usuarios. Esto

asegura que incluso los usuarios autenticados solo puedan acceder a las partes de la aplicación para las que tienen los permisos necesarios.

### **Implementación de Encabezados de Seguridad HTTP (Secure Headers)**

La gema **secure\_headers** facilita la configuración de encabezados de seguridad HTTP que pueden mejorar significativamente la seguridad de una aplicación web. Estos encabezados incluyen Content Security Policy (CSP) para controlar los orígenes de los recursos permitidos, HTTP Strict Transport Security (HSTS) para forzar el uso de HTTPS, X-Frame-Options para prevenir clickjacking, y X-XSS-Protection para habilitar filtros XSS en navegadores antiguos.

### **Gestión Segura de Secretos y Credenciales en RoR**

Es fundamental evitar la codificación dura de secretos y credenciales (como claves de API, contraseñas de bases de datos, etc.) en el código o en archivos de configuración. Rails proporciona la funcionalidad **credentials.yml.enc** para almacenar secretos cifrados, utilizando una clave maestra que debe mantenerse segura y nunca subirse al repositorio. También se pueden utilizar variables de entorno para configuraciones sensibles.

### **Validación y Sanitización de Entradas de Usuario**

Validar todas las entradas del usuario en el lado del servidor es esencial para asegurar que los datos cumplen con las expectativas de la aplicación (tipo, formato, longitud, etc.). Además de la validación, las entradas deben ser sanitizadas para eliminar o escapar cualquier código malicioso potencial antes de ser procesadas o mostradas.

### **Gestión y Actualización Segura de Dependencias (uso de bundler-audit)**

Mantener las gemas de Ruby actualizadas a las últimas versiones es crucial para corregir vulnerabilidades de seguridad conocidas. La gema **bundler-audit** compara el **Gemfile.lock** de la aplicación con una base de datos de vulnerabilidades conocidas y alerta sobre posibles problemas. Se recomienda integrar **bundler-audit** en el proceso de desarrollo y en los pipelines de Integración Continua/Despliegue Continuo (CI/CD). Herramientas como Dependabot o el dependency scanner de GitHub pueden automatizar el proceso de identificación y actualización de dependencias vulnerables.

#### 4.3.4. Conclusiones y Recomendaciones para la Seguridad en RoR

Asegurar una aplicación Ruby on Rails es un proceso continuo que requiere una combinación de herramientas, gemas y prácticas de desarrollo seguras. Las herramientas de análisis estático como Brakeman y Rubocop proporcionan mecanismos valiosos para identificar vulnerabilidades en las primeras etapas del ciclo de desarrollo. Gemas como Devise, CanCanCan y Pundit simplifican la implementación de características de seguridad esenciales como la autenticación y la autorización.

Para desarrollar aplicaciones RoR seguras, se recomienda:

- **Integrar herramientas SAST** como Brakeman en el flujo de trabajo de desarrollo y en los pipelines de CI/CD para realizar análisis de seguridad automáticos de forma regular.
- **Utilizar gemas de autenticación y autorización** como Devise y Pundit (o CanCanCan), configurándolas de forma segura y siguiendo las mejores prácticas para gestionar usuarios y permisos.
- **Gestionar las dependencias de forma segura** utilizando **bundler-audit** para detectar vulnerabilidades conocidas y mantener las gemas actualizadas con las últimas versiones y parches de seguridad.
- **Implementar encabezados de seguridad HTTP** utilizando la gema **secure\_headers** para proteger contra ataques basados en el navegador como XSS y clickjacking.
- **Seguir prácticas de codificación segura** para prevenir vulnerabilidades comunes como inyección SQL y XSS, utilizando las funcionalidades de Rails para el escaping y evitando la manipulación insegura de entradas de usuario en consultas a la base de datos.
- **Gestionar los secretos y credenciales de forma segura** utilizando las funcionalidades de Rails como **credentials.yml.enc** y evitando la codificación dura de información sensible en el código.
- **Implementar un logging y monitoring adecuados** para registrar eventos de seguridad relevantes y detectar posibles incidentes.

- **Realizar auditorías de seguridad regulares** y pruebas de penetración para identificar y abordar posibles debilidades en la aplicación.
- **Mantenerse al día** con las últimas vulnerabilidades y mejores prácticas en seguridad de Ruby on Rails, participando en la comunidad y revisando los *advisories* de seguridad.

#### **4.4. Limitaciones actuales y necesidad de marcos reutilizables**

A pesar de la madurez del *framework* Ruby on Rails y de las características de seguridad que inherentemente ofrece, las aplicaciones web construidas con esta tecnología siguen siendo susceptibles a diversas brechas de seguridad. Esta persistencia de vulnerabilidades sugiere que confiar únicamente en las características de seguridad predeterminadas del marco es insuficiente para garantizar una protección integral.

##### **4.4.1. Problemas Recurrentes: El Factor Humano y la Complejidad del Ecosistema**

###### **Errores de Desarrollo Comunes**

A pesar de la disponibilidad de amplios recursos y herramientas de desarrollo seguro, los desarrolladores frecuentemente cometen errores que comprometen la seguridad de las aplicaciones web. Estos errores a menudo surgen de una falta de comprensión profunda de los principios de seguridad o de la incapacidad de aplicar dicho conocimiento de manera efectiva durante el proceso de desarrollo. Un error común es la validación y sanitización inadecuada de la entrada del usuario. Si los datos proporcionados por el usuario no se verifican y limpian correctamente antes de su procesamiento, pueden explotarse para inyectar código malicioso o manipular la lógica de la aplicación.

Otro error frecuente implica el manejo inseguro de cookies, sesiones y tokens JWT. Los desarrolladores pueden almacenar información sensible en cookies sin cifrarla adecuadamente o pueden implementar mecanismos de gestión de sesiones débiles que los atacantes pueden secuestrar. Además, el manejo inadecuado de errores puede revelar información sensible sobre el sistema a posibles atacantes. Los mensajes de error detallados que exponen la estructura interna de la aplicación o los detalles de la configuración pueden proporcionar información valiosa a los adversarios, facilitando sus intentos de explotar vulnerabilidades.



Los errores de seguridad pueden clasificarse ampliamente en "errores simples" que resultan de pequeños errores de programación al intentar implementar una práctica de seguridad válida, y vulnerabilidades que surgen de una comprensión errónea de los conceptos de seguridad. Si bien los errores simples pueden ser menos comunes, las vulnerabilidades derivadas de malentendidos conceptuales son significativamente más frecuentes, lo que subraya la necesidad de una educación y formación en seguridad más sólidas para los desarrolladores. A menudo, la seguridad no es el objetivo principal de los desarrolladores, lo que lleva a que las consideraciones de seguridad se pasen por alto o se implementen incorrectamente.

### **Misconfiguraciones y Despliegue Inseguro**

Incluso cuando el código de una aplicación web está desarrollado con las mejores intenciones de seguridad, las misconfiguraciones en el entorno de despliegue pueden introducir vulnerabilidades significativas. Una misconfiguración de seguridad se define como un fallo de seguridad causado directamente por la forma en que se configura una aplicación o su entorno, y no por una vulnerabilidad en la propia aplicación. Un informe reveló que una proporción sustancial de empresas experimentan al menos una misconfiguración de seguridad crítica.

Las misconfiguraciones comunes incluyen el uso de credenciales predeterminadas y débiles. Muchos sistemas, aplicaciones y servicios en la nube vienen con nombres de usuario y contraseñas predeterminados que, si no se cambian, proporcionan un punto de entrada fácil para los atacantes. Las configuraciones predeterminadas inseguras también representan un riesgo. Las instalaciones de software pueden tener características de seguridad críticas desactivadas de forma predeterminada o permisos demasiado permisivos, lo que aumenta la superficie de ataque.

La falta de actualizaciones de seguridad es otra misconfiguración frecuente. No aplicar parches y actualizaciones de software de manera oportuna deja los sistemas vulnerables a las vulnerabilidades conocidas que los atacantes pueden explotar. Los controles de acceso inadecuados son igualmente problemáticos. Cuando los sistemas no restringen adecuadamente el acceso de los usuarios en función de los roles o las políticas de seguridad definidas, puede producirse un acceso no autorizado a datos o funcionalidades sensibles.

La insuficiencia en el registro y la monitorización es otra área de preocupación. Sin un registro adecuado de los eventos relacionados con la seguridad y el seguimiento de la actividad del sistema en busca de anomalías, resulta difícil detectar y responder a los incidentes de seguridad de manera eficaz. Además, las reglas de firewall y los grupos de seguridad configurados incorrectamente pueden exponer servicios y puertos innecesariamente, mientras que la configuración incorrecta del cifrado puede dejar los datos vulnerables a la interceptación. Habilitar o instalar características innecesarias, no cambiar las cuentas y contraseñas predeterminadas y no deshabilitar el listado de directorios son ejemplos adicionales de misconfiguraciones que pueden explotarse. La exposición de información sensible a través de mensajes de error detallados o la configuración incorrecta de los permisos de los servicios en la nube también pueden llevar a graves compromisos de seguridad.

### **Uso Incorrecto y Vulnerabilidades en Gemas**

Muchas gemas pueden agregar características valiosas a una aplicación, pero también pueden contener comportamientos o vulnerabilidades que podrían ser explotadas por atacantes. Por lo tanto, los desarrolladores deben ser conscientes de las implicaciones de seguridad de las gemas que incluyen en sus proyectos.

Identificar y gestionar las vulnerabilidades en las dependencias de las gemas puede ser un desafío. Las vulnerabilidades se descubren continuamente en las bibliotecas de software y es esencial que los desarrolladores mantengan sus dependencias actualizadas para mitigar los riesgos. Herramientas como **bundler-audit** pueden ayudar a detectar versiones vulnerables de gemas enumerando las vulnerabilidades conocidas en las gemas utilizadas por un proyecto.

El concepto de "dependency hell", donde los conflictos entre las dependencias pueden dificultar la actualización, y el riesgo de gemas de código abierto comprometidas son preocupaciones adicionales. Los atacantes pueden inyectar código malicioso en las gems, que luego pueden ser descargadas y utilizadas por los desarrolladores sin saberlo, lo que lleva a posibles compromisos de seguridad.

#### **4.4.2. Falta de Formación y Concienciación en Seguridad**

La persistencia de vulnerabilidades en las aplicaciones web de Ruby on Rails se ve exacerbada por una formación y concienciación en seguridad insuficientes entre los

desarrolladores. Un porcentaje significativo de desarrolladores no considera la seguridad de las aplicaciones como una prioridad principal y carece de un conocimiento profundo de las prácticas de desarrollo seguro.

Existe una brecha notable entre el reconocimiento de los riesgos de seguridad y la implementación real de prácticas seguras. Si bien los desarrolladores pueden ser conscientes de la importancia de la seguridad, pueden carecer de las habilidades o la motivación necesarias para abordar las vulnerabilidades de manera efectiva. Esta desconexión subraya la necesidad de una formación que no solo aumente la concienciación, sino que también proporcione orientación práctica y conocimientos aplicables.

#### **4.4.3. Justificación del *framework* propuesto: Hacia Componentes Seguros y Reutilizables**

##### **Necesidad de Componentes Seguros Reutilizables**

Los componentes seguros y reutilizables ofrecen una solución prometedora para abordar los desafíos persistentes en la seguridad de las aplicaciones web de Ruby on Rails. Al encapsular patrones de seguridad comunes en componentes reutilizables, los desarrolladores pueden aplicar consistentemente las mejores prácticas de seguridad en toda una aplicación. Este enfoque reduce el tiempo y el esfuerzo de desarrollo al aprovechar componentes probados y prediseñados para tareas comunes relacionadas con la seguridad.

La reutilización de componentes seguros también promueve la encapsulación y la modularidad. La encapsulación garantiza que el funcionamiento interno de un componente esté oculto del mundo exterior, evitando conflictos y garantizando un comportamiento consistente en diferentes partes de una aplicación. La modularidad permite a los desarrolladores combinar diferentes componentes para construir interfaces más complejas sin introducir una complejidad innecesaria.

##### **Seguridad por Defecto y Facilidad de Uso**

Priorizar la "seguridad por defecto" es crucial para minimizar la carga sobre los desarrolladores y reducir la probabilidad de misconfiguraciones. Un marco con componentes seguros incorporados puede guiar a los desarrolladores hacia prácticas seguras sin requerir un conocimiento extenso de seguridad. Esto contrasta con la situación actual, donde los

desarrolladores a menudo necesitan reunir medidas de seguridad utilizando varias gemas y configuraciones, lo que aumenta el riesgo de errores.

Un marco que priorice la seguridad por defecto y ofrezca componentes seguros fáciles de usar puede reducir significativamente la barrera para construir aplicaciones de Ruby on Rails seguras. Al hacer de la seguridad el estándar y proporcionar herramientas intuitivas, es más probable que los desarrolladores construyan aplicaciones seguras, incluso si carecen de formación especializada en seguridad.

La originalidad del trabajo se manifiesta en su enfoque centrado en el desarrollador no experto en seguridad. Mientras que las soluciones existentes asumen cierto nivel de conocimiento especializado, este *framework* abstraerá completamente los detalles técnicos complejos, permitiendo a los desarrolladores concentrarse en la lógica de negocio mientras el sistema maneja automáticamente los aspectos de seguridad. Este paradigma de "seguridad invisible" representa un avance significativo respecto al estado actual de la técnica.

## 5. Definición de Requisitos y Componentes del Framework

En este capítulo se procede a la definición detallada de los requisitos que guiarán el diseño y la implementación del *framework*. Para una mayor claridad y organización, estos requisitos se han clasificado en dos grandes bloques: los **Requisitos No Funcionales (RNF)**, que establecen los criterios de calidad y las propiedades del sistema, y los **Requisitos Funcionales (RF)**, que describen las acciones y funcionalidades específicas que el framework deberá realizar.

### 5.1. Requisitos No Funcionales

Los requisitos no funcionales (RNF) definen las características de calidad del *framework*, estableciendo criterios que deben cumplirse más allá de las funcionalidades explícitas. En el contexto de un entorno seguro de desarrollo basado en Ruby on Rails, estos requisitos son esenciales para garantizar la robustez, la seguridad, la eficiencia y la usabilidad del framework. Estos requisitos se agrupan en las siguientes categorías:

#### 5.1.1. Seguridad

Dado que el objetivo central del *framework* es la seguridad, este apartado es de máxima prioridad.

- **NFR-1: Robustez frente a Vulnerabilidades Comunes:** El *framework* debe integrar mecanismos intrínsecos que mitiguen las vulnerabilidades de seguridad web más prevalentes según el Top 10 de OWASP [3], incluyendo, pero no limitándose a:
  - **Inyección (A01:2021-Broken Access Control):** Prevención automática de inyecciones SQL, NoSQL, OS Command Injection, etc., mediante el uso de ORMs y consultas parametrizadas, y mecanismos de saneamiento de entrada robustos.
  - **Autenticación y Gestión de Sesiones Inseguras (A07:2021-Identification and Authentication Failures):** Implementación de sistemas de autenticación seguros por defecto (ej., basados en **Devise** o similar), con gestión de contraseñas (hashing, salting) y sesiones (CSRF tokens, manejo de sesión seguro).

- **Cross-Site Scripting (XSS) (A03:2021-Injection):** Mecanismos de escape y saneamiento de salida automáticos en las vistas, asegurando que el contenido generado por el usuario sea seguro antes de ser renderizado.
- **Componentes Vulnerables y Obsoletos (A06:2021-Vulnerable and Outdated Components):** Facilitar la gestión de dependencias seguras y actualizadas.
- **Control de Acceso Deficiente (A01:2021-Broken Access Control):** Integración de mecanismos de autorización robustos (ej., **CanCanCan**, **Pundit**) que faciliten la implementación de reglas de acceso granuladas y prevengan la escalada de privilegios.
- **NFR-2: Gestión de Secretos:** El *framework* debe ofrecer un mecanismo seguro para la gestión de secretos (claves API, credenciales de bases de datos, etc.), preferiblemente integrado con la funcionalidad de **Rails Encrypted Credentials**.
- **NFR-3: Auditoría y Logging de Seguridad:** Capacidad de registrar eventos de seguridad críticos para su posterior análisis y monitoreo.

### 5.1.2. Rendimiento y Estabilidad

El *framework* debe ser eficiente en el uso de recursos y ofrecer tiempos de respuesta adecuados.

- **NFR-4: Mínimo Impacto en el Rendimiento:** Los mecanismos de seguridad implementados por el *framework* no deben introducir una sobrecarga de rendimiento significativa en las aplicaciones web. Se buscará un equilibrio entre seguridad robusta y eficiencia operativa.

### 5.1.3. Usabilidad y Experiencia del Desarrollador

Un *framework* seguro solo será efectivo si es fácil de usar y no impone una carga excesiva a los desarrolladores.

- **NFR-5: Facilidad de Integración:** El *framework* deberá ser fácilmente integrable en proyectos Ruby on Rails nuevos o existentes, con una curva de aprendizaje suave para desarrolladores con experiencia en Rails.
- **NFR-6: Abstracción de Complejidad:** Debe abstraer la complejidad de la implementación de controles de seguridad, permitiendo a los desarrolladores asegurar sus aplicaciones sin necesidad de ser expertos en ciberseguridad.

- **NFR-7: Documentación Clara y Concisa:** Se proporcionará documentación detallada que explique la configuración, uso y las mejores prácticas de los componentes de seguridad del framework.

#### 5.1.4. Mantenibilidad

La capacidad de mantener y evolucionar el *framework* a lo largo del tiempo es fundamental para su viabilidad a largo plazo.

- **NFR-8: Modularidad:** El *framework* estará diseñado de forma modular, permitiendo la adición o eliminación de componentes de seguridad específicos sin afectar la totalidad del sistema.
- **NFR-9: Actualizabilidad:** Debe ser compatible con las últimas versiones de Ruby y Ruby on Rails, y facilitar la integración de actualizaciones de seguridad en las gemas y bibliotecas subyacentes.

#### 5.1.5. Compatibilidad

Para asegurar su adopción y mantenibilidad a largo plazo, el framework debe ser compatible con las versiones relevantes del ecosistema de Ruby on Rails. Por tanto, se establece el siguiente requisito:

- **NFR-10: Compatibilidad con Versiones de Rails:** El *framework* será compatible con versiones recientes y estables de Ruby on Rails

### 5.2. Requisitos Funcionales

Los requisitos funcionales definen las acciones y funcionalidades específicas que el *framework* deberá realizar para cumplir con su objetivo principal: facilitar la construcción de aplicaciones web seguras sobre Ruby on Rails, integrando protecciones por defecto y componentes seguros reutilizables. Estos requisitos se centran en lo que el sistema hará.

A continuación, se detallan los requisitos funcionales del framework.

#### 5.2.1. Autenticación Segura

Los requisitos funcionales para la gestión de la identidad y la autenticación segura de los usuarios son los siguientes:

- **FR-1: Gestión de Identidad y Sesiones:** El *framework* ofrecerá componentes para una gestión de identidad segura (registro, inicio/cierre de sesión) y gestión de sesiones robusta (protección contra fijación, timeouts configurables, almacenamiento seguro de tokens de sesión).
- **FR-2: Almacenamiento Seguro de Credenciales:** Se integrarán mecanismos para el almacenamiento de contraseñas utilizando algoritmos de hashing fuertes (ej. bcrypt) con gestión de sales.

### 5.2.2. Autorización y Control de Acceso

Para garantizar que los usuarios solo puedan acceder a los recursos permitidos, se define el siguiente requisito de control de acceso:

- **FR-3: Control de Acceso Basado en Roles (Núcleo):** Proveerá una forma simplificada pero efectiva de definir y aplicar políticas de control de acceso basadas en roles para proteger el acceso a diferentes partes de la aplicación.

### 5.2.3. Validación y Sanitización de Entradas

Con el fin de prevenir ataques de inyección y otros riesgos asociados a la manipulación de datos, el framework debe implementar los siguientes mecanismos de validación y sanitización:

- **FR-4: Prevención de Inyección SQL:** Integrará por defecto el uso seguro del ORM de Rails (Active Record) para construir consultas, previniendo la inyección SQL mediante el uso de consultas parametrizadas.
- **FR-5: Prevención de Cross-Site Scripting (XSS):** Asegurará el escapado automático de datos en las vistas. Proveerá directrices claras sobre el manejo seguro de HTML cuando sea indispensable, priorizando **sanitize** sobre **html\_safe** o **raw** con entradas no confiables.

### 5.2.4. Protección contra Vulnerabilidades Comunes

El framework debe integrar de manera proactiva defensas contra vulnerabilidades web prevalentes. Específicamente, se establecen los siguientes requisitos:

- **FR-6: Protección CSRF:** Integrará y habilitará por defecto la protección contra Cross-Site Request Forgery (CSRF) en todas las peticiones no-GET, utilizando tokens de autenticidad.
- **FR-7: Configuración de Cabeceras de Seguridad:** Aplicará automáticamente cabeceras HTTP de seguridad relevantes (ej. Content Security Policy (CSP), HTTP



Strict Transport Security (HSTS), X-Frame-Options, X-Content-Type-Options, Referrer-Policy) con configuraciones predeterminadas seguras.

### 5.2.5. Gestión Segura de Dependencias

Para mitigar los riesgos derivados de la cadena de suministro de software, se establece el siguiente requisito para la gestión de dependencias:

- **FR-8: Integración con bundler-audit:** Facilitará o recomendará activamente la integración y uso de herramientas como bundler-audit para la revisión de vulnerabilidades conocidas en las gemas del proyecto.

### 5.2.6. Gestión Segura de Secretos

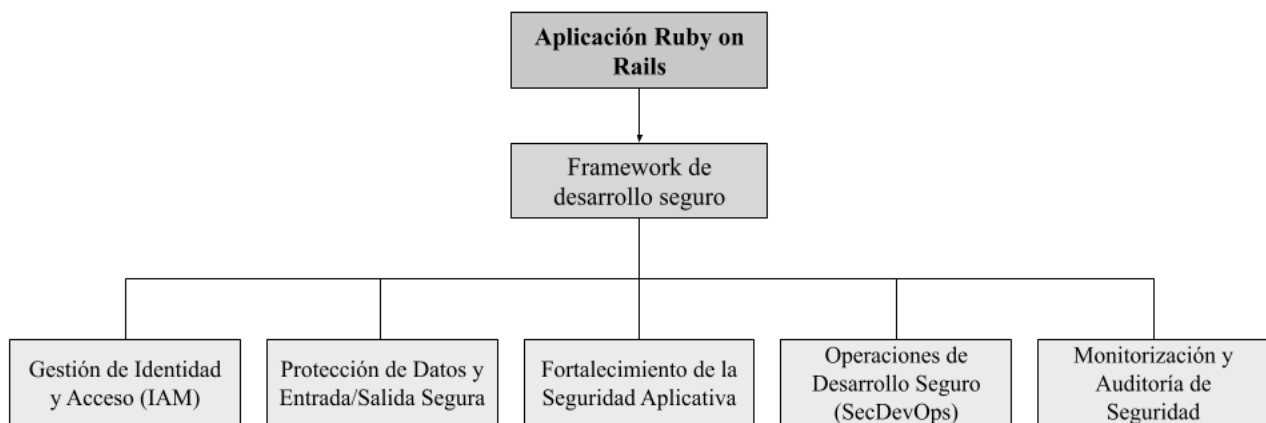
La correcta gestión de credenciales y secretos de aplicación es fundamental para la seguridad del sistema. Por ello, se define el siguiente requisito funcional:

- **FR-9: Uso de credentials.yml.enc:** El *framework* promoverá y se alineará con el uso del sistema de credenciales cifradas de Rails (credentials.yml.enc) para la gestión de secretos de aplicación, evitando la codificación de información sensible.

### 5.3. Componentes del Framework

Este apartado detalla los componentes principales que constituyen el *framework* de desarrollo seguro sobre Ruby on Rails (a partir de ahora se denominará como **secure\_framework**). El diseño de estos componentes se orienta a la modularidad, la facilidad de integración y la promoción de la "seguridad por defecto", tal como se ha establecido anteriormente en los requisitos funcionales y no funcionales.

A continuación, se presenta un diagrama que ilustra la arquitectura general del *framework* y cómo se agrupan los componentes:

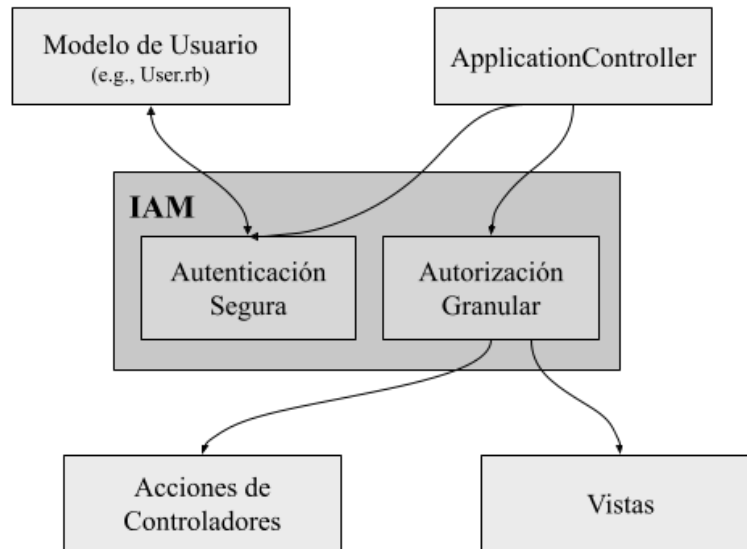


**Figura 5.1:** Arquitectura general del framework y grupos de componentes.

A continuación, se detallan los grupos de componentes y los componentes individuales dentro de ellos.

### 5.3.1. Gestión de Identidad y Acceso (IAM)

Este grupo de componentes se encarga de asegurar que solo usuarios legítimos accedan a la aplicación y únicamente a los recursos para los cuales tienen permiso.



**Figura 5.2:** Componentes e interacciones del grupo Gestión de Identidad y Acceso (IAM)

#### Componente de Autenticación Segura

- **Objetivo/Funcionalidad:** Proveer un sistema robusto y configurable para la verificación de la identidad de los usuarios y la gestión de sesiones seguras, cumpliendo con FR-1 y FR-2.
- **Características Clave:**
  - Integración con soluciones de autenticación probadas como Devise.
  - Almacenamiento seguro de credenciales mediante hashing fuerte (e.g., bcrypt) y gestión de sales.
  - Protección contra ataques de enumeración de usuarios y temporización.
  - Mecanismos de bloqueo de cuentas tras intentos fallidos.
  - Gestión segura de tokens de sesión (protección contra fijación, timeouts configurables).
- **Capacidades:**
  - Funcionalidades estándar: registro, inicio/cierre de sesión, recuperación de contraseña, opción "recuérdame".
  - Extensible para implementar autenticación de múltiples factores (MFA).
  - Generación de tokens seguros para APIs.
- **Restricciones/Consideraciones:**
  - Requiere la definición de un modelo de usuario en la aplicación.

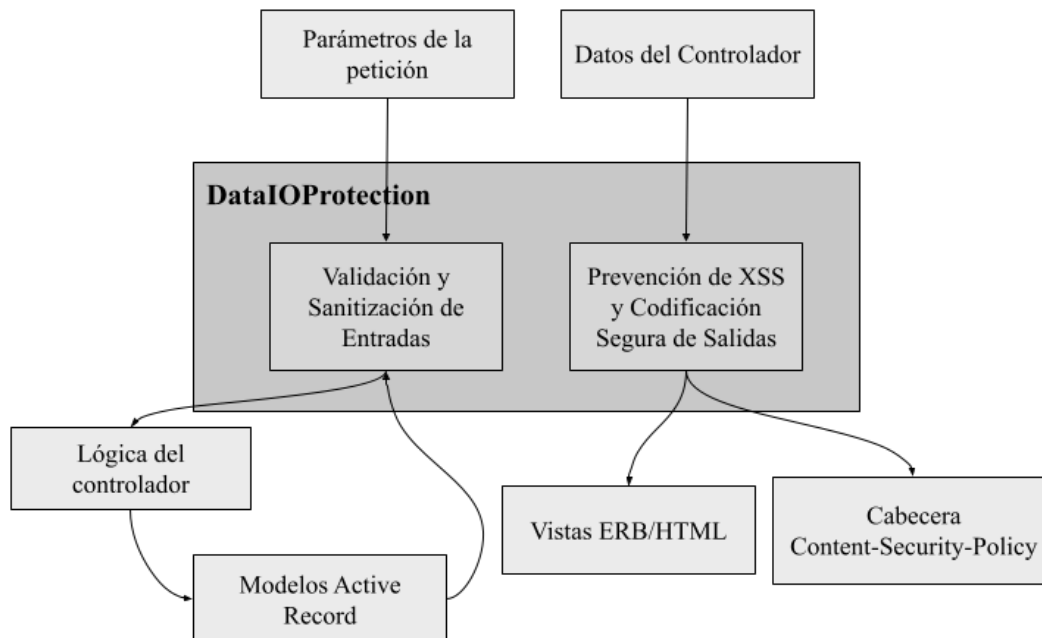
- La personalización avanzada de flujos de autenticación puede requerir un conocimiento más profundo de la gema subyacente (e.g., Devise).
- **Ejemplo de Uso/Caso Aplicable:** Un desarrollador necesita implementar un sistema de inicio de sesión. Ejecuta el generador del *framework*, que configura automáticamente Devise con las políticas de seguridad predefinidas, permitiendo al desarrollador centrarse en la lógica de negocio específica de su modelo de usuario.

### Componente de Autorización Granular

- **Objetivo/Funcionalidad:** Ofrecer un mecanismo flexible y claro para definir y aplicar políticas de control de acceso basadas en roles o capacidades, cumpliendo con FR-3 y contribuyendo a NFR-1 (Control de Acceso Deficiente).
- **Características Clave:**
  - Integración con gemas de autorización como Pundit o CanCanCan. Pundit es preferido por su enfoque en clases Ruby simples (policies).
  - Definición de políticas de acceso desacopladas de los controladores y modelos.
  - Facilidad para probar las políticas de autorización de forma aislada.
- **Capacidades:**
  - Implementación de Control de Acceso Basado en Roles (RBAC).
  - Definición de permisos a nivel de instancia de modelo y de acción.
  - Filtrado de colecciones de datos (scopes) basado en los permisos del usuario actual.
- **Restricciones/Consideraciones:**
  - Requiere que el desarrollador defina explícitamente las políticas para cada recurso que necesite protección.
  - La complejidad de las políticas puede crecer en aplicaciones con muchos roles y permisos interdependientes.
- **Ejemplo de Uso/Caso Aplicable:** En una aplicación de blog, solo los autores pueden editar sus propios posts y los administradores pueden editar cualquier post. El desarrollador define estas reglas en PostPolicy. El framework ayuda a asegurar que estas políticas se apliquen consistentemente en los controladores.

### 5.3.2. Protección de Datos y Entrada/Salida Segura

Estos componentes se enfocan en proteger la aplicación contra ataques que explotan la manipulación de entradas y la presentación insegura de datos.



**Figura 5.3:** Componentes e interacciones del grupo Protección de Datos y Entrada/Salida Segura

#### Componente de Validación y Sanitización de Entradas

- **Objetivo/Funcionalidad:** Asegurar que todas las entradas de usuario sean validadas y sanitizadas rigurosamente antes de ser procesadas, previniendo vulnerabilidades como Inyección SQL y otras basadas en entradas maliciosas, cumpliendo con FR-4 y FR-5 (parcialmente) y contribuyendo a NFR-1.
- **Características Clave:**
  - Uso seguro del ORM de Rails (Active Record) para prevenir Inyección SQL mediante consultas parametrizadas.
  - Validaciones a nivel de modelo para tipo de dato, formato, longitud, presencia, etc.
  - Mecanismos para la sanitización de entradas que no se gestionan directamente por Active Record (e.g., parámetros de búsqueda, URL).
- **Capacidades:**
  - Promover el uso de Strong Parameters de forma exhaustiva.
  - Ofrecer validadores personalizados para tipos de datos comunes (e.g., URLs seguras, formatos específicos).
  - Integrar mecanismos para la detección y rechazo de patrones maliciosos en las entradas.
- **Restricciones/Consideraciones:**

- La validación es específica del contexto; el *framework* puede proveer herramientas, pero el desarrollador debe definir las reglas.
- Una sanitización excesiva podría rechazar entradas válidas si no se configura con cuidado.
- **Ejemplo de Uso/Caso Aplicable:** Un formulario permite a los usuarios introducir un enlace a su sitio web. El framework proporciona un **SecureUrlValidator** que verifica que la URL utilice HTTPS y no contenga caracteres sospechosos, previniendo así posibles ataques XSS o de redirección abierta a través de este campo.

### Componente de Prevención de XSS y Codificación Segura de Salidas

- **Objetivo/Funcionalidad:** Prevenir ataques de Cross-Site Scripting (XSS) asegurando que todas las salidas se codifiquen adecuadamente y configurando políticas de seguridad de contenido (CSP), cumpliendo con FR-5 y contribuyendo a NFR-1.
- **Características Clave:**
  - Asegurar que el escapado automático de Rails en las vistas (`<%= ... %>`) esté siempre activo y no se eluda innecesariamente.
  - Proveer directrices y helpers para el manejo seguro de HTML cuando sea indispensable, priorizando **sanitize** sobre **html\_safe** o **raw** con entradas no confiables.
  - Configuración de una Content Security Policy (CSP) robusta por defecto.
- **Capacidades:**
  - Generación y gestión de una política CSP que restrinja las fuentes de scripts, estilos, imágenes, etc.
  - Facilitar la inclusión de nonces o hashes para scripts inline si son estrictamente necesarios.
  - Integración con las funcionalidades de Rails para CSP.
- **Restricciones/Consideraciones:**
  - Una CSP muy restrictiva puede romper funcionalidades si no se configura correctamente para incluir todas las fuentes legítimas.
  - El uso de **html\_safe** o **raw** sigue siendo una decisión del desarrollador; el framework puede advertir pero no siempre prevenir su mal uso.
- **Ejemplo de Uso/Caso Aplicable:** Al renderizar comentarios de usuarios, el framework asegura que cualquier HTML o script inyectado sea neutralizado por el escapado automático. Adicionalmente, la CSP previene que el navegador ejecute scripts de fuentes no autorizadas, añadiendo una capa de defensa en profundidad.

### 5.3.3. Fortalecimiento de la Seguridad Aplicativa

Componentes diseñados para aplicar configuraciones y mecanismos de seguridad que endurecen la aplicación contra diversas amenazas web.

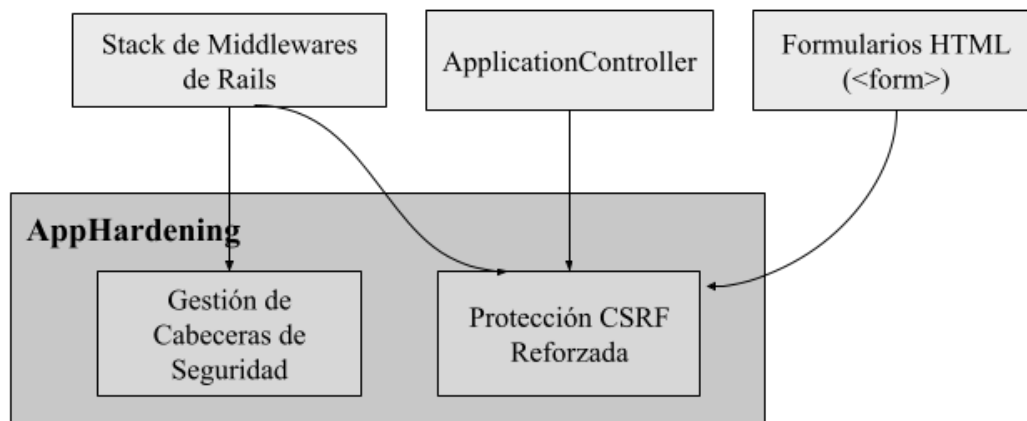


Figura 5.4: Componentes e interacciones del grupo Fortalecimiento de la Seguridad Aplicativa

#### Componente de Gestión de Cabeceras de Seguridad

- **Objetivo/Funcionalidad:** Aplicar automáticamente un conjunto de cabeceras HTTP de seguridad para proteger la aplicación contra ataques comunes como clickjacking, sniffing de MIME, y forzar el uso de HTTPS, cumpliendo con FR-7.
- **Características Clave:**
  - Configuración de cabeceras como **X-Frame-Options**, **X-Content-Type-Options**, **Strict-Transport-Security (HSTS)**, **Referrer-Policy**.
  - Integración con la gema **secure\_headers** o configuración directa a través de middlewares de Rails.
- **Capacidades:**
  - Habilitar HSTS para asegurar que el navegador solo acceda a la aplicación mediante HTTPS.
  - Prevenir que la aplicación sea embebida en *iframes* maliciosos (**X-Frame-Options: DENY o SAMEORIGIN**).
  - Evitar que el navegador interprete archivos con un tipo MIME incorrecto (**X-Content-Type-Options: nosniff**).
  - Controlar la cantidad de información de referente enviada (**Referrer-Policy**).
- **Restricciones/Consideraciones:**

- HSTS, una vez configurado, puede hacer que el sitio sea inaccesible si se revierte a HTTP antes de que expire el tiempo de HSTS en los navegadores de los usuarios. Debe implementarse con cuidado.
- **Ejemplo de Uso/Caso Aplicable:** El framework aplica automáticamente la cabecera **X-Frame-Options: SAMEORIGIN**. Esto impide que un sitio web malicioso cargue la aplicación del usuario en un iframe para engañar a los usuarios y que realicen acciones no deseadas (clickjacking).

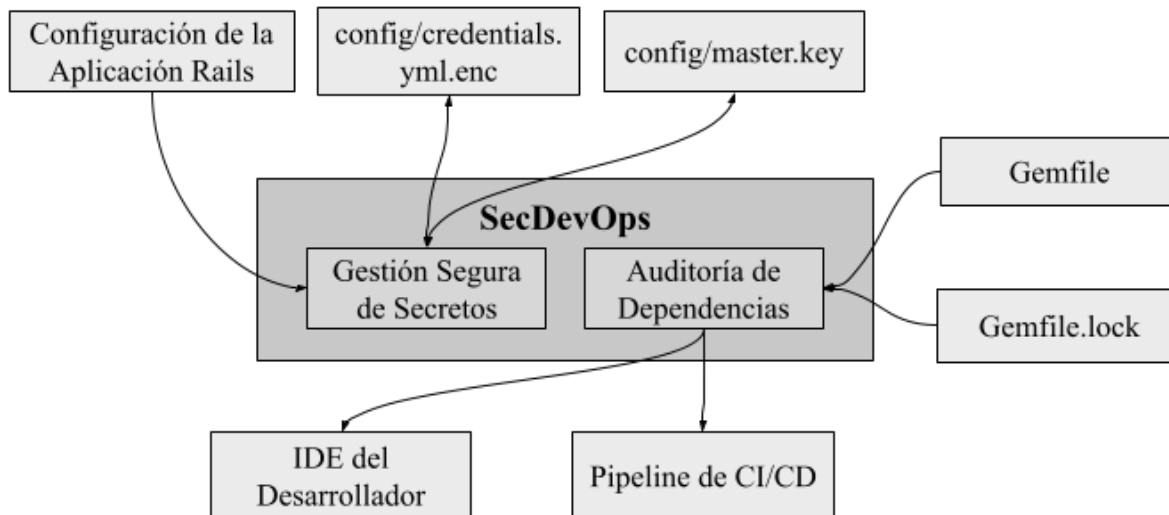
### Componente de Protección CSRF Reforzada

- **Objetivo/Funcionalidad:** Asegurar que la protección contra Cross-Site Request Forgery (CSRF) nativa de Rails esté correctamente implementada y habilitada en toda la aplicación, cumpliendo con FR-6.
- **Características Clave:**
  - Uso de tokens de autenticidad en todos los formularios y peticiones AJAX que modifican estado.
  - Verificación de estos tokens en el lado del servidor.
- **Capacidades:**
  - Protección por defecto para todos los métodos HTTP no-GET.
  - Configuración de la estrategia de protección (e.g., **protect\_from\_forgery with: :exception**).
- **Restricciones/Consideraciones:**
  - Las APIs que utilizan tokens de autenticación (e.g., JWT) para la gestión de sesiones pueden necesitar configuraciones CSRF específicas o deshabilitarla para esos endpoints, siempre que se implementen otras medidas anti-CSRF adecuadas para APIs stateless (como la comprobación del header **Origin/Referer** o double submit cookie para APIs stateful).
- **Ejemplo de Uso/Caso Aplicable:** Un usuario está logueado en la aplicación. Un atacante intenta engañarlo para que haga clic en un enlace en un sitio malicioso que envía una petición **POST** a la aplicación para eliminar la cuenta del usuario. Gracias a la protección CSRF, la petición del atacante carecerá del token CSRF válido, y la aplicación la rechazará, protegiendo la cuenta del usuario.



### 5.3.4. Operaciones de Desarrollo Seguro (SecDevOps)

Componentes que ayudan a integrar la seguridad en el ciclo de vida del desarrollo y en la gestión de la configuración de la aplicación.



**Figura 5.5:** Componentes e interacciones del grupo Operaciones de Desarrollo Seguro

#### Componente de Gestión Segura de Secretos

- **Objetivo/Funcionalidad:** Facilitar y promover el uso del sistema de credenciales cifradas de Rails para la gestión segura de secretos de aplicación, evitando la codificación de información sensible en el código fuente, cumpliendo con FR-9 y NFR-2.
- **Características Clave:**
  - Uso de **config/credentials.yml.enc** para almacenar secretos cifrados.
  - Uso de **config/master.key** (o variable de entorno **RAILS\_MASTER\_KEY**) para descifrar las credenciales.
- **Capacidades:**
  - Almacenamiento seguro de claves de API, contraseñas de bases de datos externas, y otras credenciales.
  - Acceso fácil a los secretos en la aplicación a través de **Rails.application.credentials**.
- **Restricciones/Consideraciones:**
  - El archivo **master.key** NUNCA debe ser versionado en el repositorio de código. Su gestión segura es crítica.

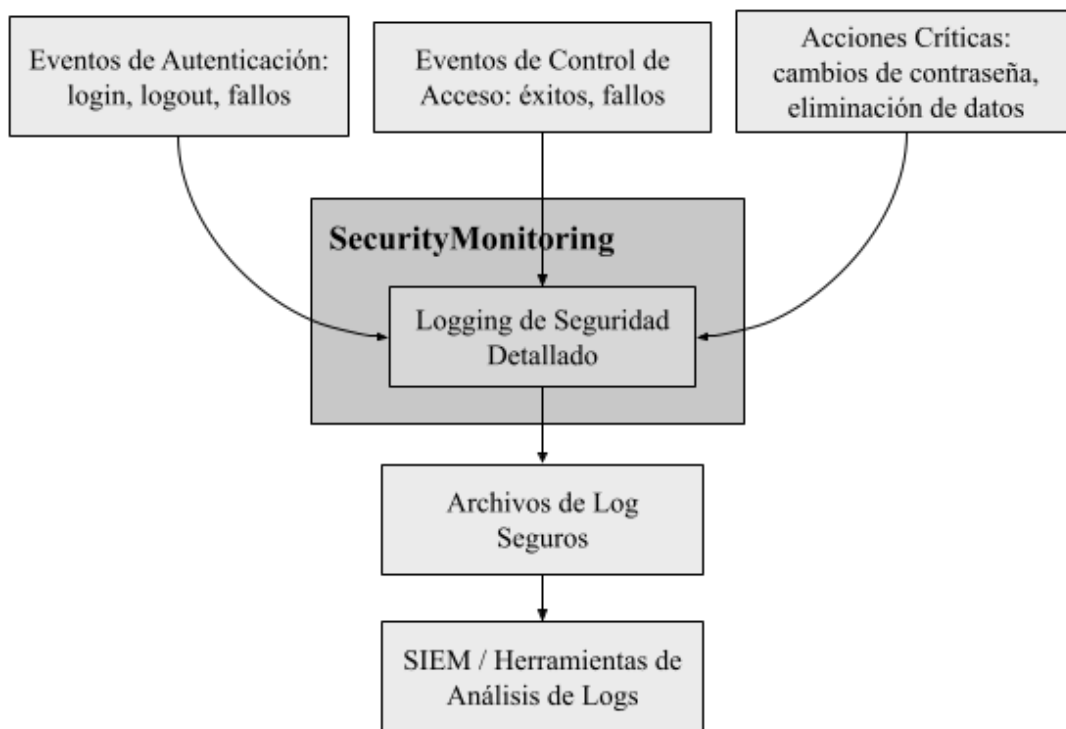
- En entornos de producción, **RAILS\_MASTER\_KEY** debe proveerse como una variable de entorno.
- **Ejemplo de Uso/Caso Aplicable:** La aplicación necesita conectarse a una API externa que requiere una clave. En lugar de escribir la clave en un archivo de configuración o en el código, el desarrollador la añade a **credentials.yml.enc** usando **bin/rails credentials:edit**. Luego, accede a ella de forma segura en el código mediante **Rails.application.credentials.external\_api\_key**.

## Componente de Auditoría de Dependencias

- **Objetivo/Funcionalidad:** Integrar herramientas para la revisión continua de vulnerabilidades conocidas en las gemas del proyecto, cumpliendo con FR-8 y NFR-1 (Componentes Vulnerables).
- **Características Clave:**
  - Integración con la gema **bundler-audit**.
  - Comparación del **Gemfile.lock** con bases de datos de vulnerabilidades (e.g., Ruby Advisory Database).
- **Capacidades:**
  - Detección de gemas con vulnerabilidades conocidas.
  - Sugerencias para actualizar a versiones seguras.
  - Integrable en pipelines de Integración Continua/Despliegue Continuo (CI/CD).
- **Restricciones/Consideraciones:**
  - La efectividad depende de la actualidad de la base de datos de vulnerabilidades.
  - Puede generar falsos positivos o reportar vulnerabilidades de bajo riesgo que el equipo decida aceptar temporalmente.
- **Ejemplo de Uso/Caso Aplicable:** Antes de realizar un despliegue, el pipeline de CI/CD ejecuta automáticamente **bundle audit**. Si se detecta una vulnerabilidad crítica en una de las gemas utilizadas, el despliegue se detiene, y se notifica al equipo de desarrollo para que actualice la gema vulnerable.

### 5.3.5. Monitorización y Auditoría de Seguridad

Componentes que ayudan a integrar la seguridad en el ciclo de vida del desarrollo y en la gestión de la configuración de la aplicación.



**Figura 5.6:** Componentes e interacciones del grupo Monitorización y Auditoría de Seguridad

#### Componente de Logging de Seguridad Detallado

- **Objetivo/Funcionalidad:** Facilitar el registro estructurado y detallado de eventos críticos de seguridad para permitir la monitorización, alerta y análisis forense, cumpliendo con NFR-3.
- **Características Clave:**
  - Logging de eventos como intentos de inicio de sesión (exitosos y fallidos), cambios de contraseña, errores de autorización, acceso a funciones críticas, cambios en configuraciones de seguridad.
  - Formato de log estructurado (e.g., JSON) para facilitar el procesamiento por herramientas externas.
  - Inclusión de información contextual relevante (timestamp, IP de origen, ID de usuario, recurso afectado).
- **Capacidades:**
  - Configuración de niveles de log específicos para eventos de seguridad.

- Integración con el sistema de logging de Rails (**Rails.logger**) pero con un logger dedicado o etiquetado para seguridad.
- Posibilidad de enviar logs a sistemas centralizados de gestión de logs (SIEM).
- **Restricciones/Consideraciones:**
  - El logging excesivo puede impactar el rendimiento y generar grandes volúmenes de datos. Se debe lograr un equilibrio.
  - Los logs pueden contener información sensible, por lo que deben protegerse adecuadamente (acceso restringido, rotación, posible cifrado en reposo).
- **Ejemplo de Uso/Caso Aplicable:** Si se detectan múltiples intentos fallidos de inicio de sesión para una cuenta desde la misma IP, los logs de seguridad detallados permitirán identificar el patrón. Esto podría usarse para activar alertas automáticas o para investigar un posible ataque de fuerza bruta.

## 6. Diseño e Implementación: `secure_framework`

Este capítulo detalla el proceso de diseño e implementación de `secure_framework`, profundizando en su arquitectura, la lógica detrás de sus componentes clave y las guías para su integración. El objetivo es traducir los requisitos funcionales y no funcionales, definidos en el capítulo anterior, en una solución técnica, robusta y fácil de usar para los desarrolladores de Ruby on Rails.

### 6.1. Arquitectura del *Framework*

La arquitectura de `secure_framework` se ha diseñado siguiendo una filosofía de **seguridad por defecto, no intrusiva, fácil integración y centrada en el desarrollador**. El objetivo principal no es crear una capa de abstracción compleja que se ejecute en paralelo a la aplicación (como un microservicio o un proxy), sino modificar y enriquecer directamente una aplicación Rails existente para que incorpore las mejores prácticas de seguridad de manera nativa. En lugar de operar como una caja negra, el framework actúa como un orquestador y configurador inteligente, aplicando un conjunto de componentes de seguridad de una manera transparente y personalizable por el desarrollador.

#### 6.1.1. Justificación del Diseño Arquitectónico

La elección de desarrollar `secure_framework` como una gema de Rails (RubyGem) responde a la necesidad de ofrecer una solución modular, versionable y fácilmente distribuible. Esta modalidad presenta ventajas significativas frente a otras alternativas como los Rails Engines o las plantillas de aplicación (Application Templates):

- **Modularidad y Reusabilidad:** Las gemas permiten empaquetar funcionalidades específicas que pueden ser incorporadas en múltiples proyectos con una simple declaración en el **Gemfile**. Esto promueve la reutilización de código y la coherencia en la aplicación de medidas de seguridad a través de diferentes aplicaciones.
- **Versionado Independiente:** Una gema puede ser versionada de forma independiente al ciclo de vida de las aplicaciones que la utilizan. Esto facilita la gestión de actualizaciones y la introducción de parches de seguridad en el framework sin requerir modificaciones complejas en cada proyecto consumidor.

- **Distribución y Mantenimiento:** La publicación en repositorios como RubyGems.org simplifica la distribución y la instalación. El mantenimiento se centraliza en la gema, propagando las mejoras a todos sus usuarios con una actualización.
- **Separación Clara de Intereses:** El código del framework reside en su propia estructura de directorios, claramente diferenciado del código de la aplicación anfitriona. Esto mejora la organización del proyecto y reduce el acoplamiento.

En contraste, un **Rails Engine**, aunque potente para extender Rails con funcionalidades significativas, a menudo implica un mayor grado de acoplamiento o complejidad para el tipo de integraciones de configuración y pequeños componentes que **secure\_framework** busca proveer. Los Engines son más adecuados para mini-aplicaciones dentro de una aplicación mayor.

Las **Plantillas de Aplicación** (`rails new mi_app -m mi_plantilla.rb`) son útiles para la configuración inicial de un proyecto, pero su naturaleza es la de un script de configuración que se ejecuta una única vez. No ofrecen un mecanismo sencillo para la actualización continua de los componentes de seguridad o la gestión de versiones de las configuraciones aplicadas, lo cual es crucial para un framework de seguridad que debe evolucionar con el panorama de amenazas.

Por tanto, el formato de gema se alinea de manera óptima con el objetivo de proporcionar un conjunto de herramientas y configuraciones de seguridad que sean robustas, mantenibles y fáciles de adoptar por los desarrolladores.

### 6.1.2. Principios Arquitectónicos

El diseño del **secure\_framework** se guía por los siguientes principios:

- **Seguridad por Defecto (Secure by Default):** Uno de los pilares fundamentales es la aplicación automática de configuraciones y prácticas de seguridad robustas desde el momento de la instalación. El objetivo es elevar el listón de seguridad base de cualquier aplicación Rails que utilice el framework.
- **Convención sobre Configuración:** Siguiendo la filosofía de Ruby on Rails, el framework predefine configuraciones sensatas y seguras, reduciendo la necesidad de que el desarrollador tome decisiones complejas en áreas críticas de seguridad donde podría no ser experto.

- **Mínima Sorpresa (Principle of Least Astonishment):** Las acciones realizadas por el framework deben ser predecibles y transparentes. La salida proveniente del generador y la documentación buscan clarificar cualquier cambio realizado en la aplicación anfitriona.
- **Extensibilidad:** Aunque la versión inicial se centra en un conjunto definido de componentes, la arquitectura está pensada para facilitar la adición de nuevas funcionalidades y adaptadores de seguridad en el futuro.

### 6.1.3. Visión General de la Arquitectura

El *framework* se integra en una aplicación Ruby on Rails principalmente a través de **Generadores de Rails**. El generador principal, `secure_framework:install`, orquesta la instalación y configuración de los diversos componentes de seguridad.

La estructura del proyecto es la siguiente:

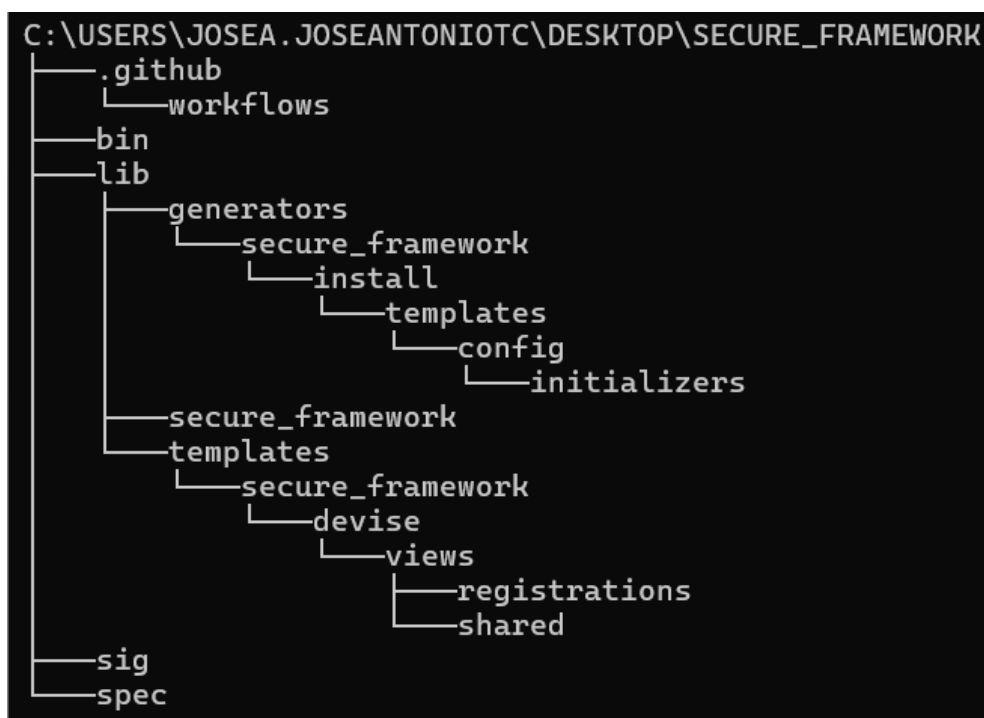


Figura 6.1: Estructura de directorios de secure\_framework

El directorio `lib/generators/secure_framework/install/` contiene la lógica del generador (`install_generator.rb`). Crucialmente, el subdirectorio `templates/` alberga archivos de configuración (e.g., inicializadores para Devise) que son copiados o generados en la aplicación Rails anfitriona (`destination_root`). Por ejemplo, `lib/generators/secure_framework/install/templates/config/initializers/` es la fuente de los archivos que se colocarán en `config/initializers/` de la aplicación Rails.

El *framework* utiliza:

1. **Generadores de Rails:** Para la instalación interactiva, la copia de plantillas de configuración y la modificación de archivos existentes en la aplicación Rails (e.g., **ApplicationController**, **.gitignore**). El archivo **install\_generator.rb** es el corazón de esta interacción.
2. **Inicializadores:** Los archivos copiados en **config/initializers/** de la aplicación anfitriona configuran las gemas de seguridad (como Devise, Pundit, Sanitize, Secure Headers) y las políticas (como CSP) cuando la aplicación Rails arranca.
3. **Tareas Rake:** Para funcionalidades que se ejecutan bajo demanda, como la auditoría de dependencias (**dependency\_audit.rake**).
4. **Módulos Ruby:** El código del framework en sí mismo (e.g., lógica dentro de **lib/secure\_framework.rb** o módulos auxiliares si fueran necesarios).

Esta arquitectura permite que **secure\_framework** aplique configuraciones seguras de manera programática, respetando las convenciones de Rails y proporcionando un punto de entrada claro para su adopción.

#### 6.1.4. Componentes Arquitectónicos Principales y su Interacción

**secure\_framework** organiza sus funcionalidades en componentes que se corresponden con dominios clave de la seguridad de aplicaciones web. Estos componentes son gestionados por el **InstallGenerator** y se apoyan en gemas de la comunidad de Rails o en mecanismos nativos del framework. La siguiente tabla resume estos componentes, su propósito y los mecanismos subyacentes:

**Tabla 6.1.** Componentes Secure Framework..

Componente	Gema/Mecanismo de Rails Principal(es)	Método(s) Relevante(s) en <b>InstallGenerator</b>	Propósito Principal
Gestión Segura de Secretos	Rails Encrypted Credentials (credentials.yml.enc, master.key)	install_secure_credentials	Asegurar que la master.key no se versiona y que las credenciales se gestionan de forma segura.
Autenticación	Devise	install_devise_and_configure	Proveer un sistema de autenticación robusto con políticas de contraseña y bloqueo de cuentas seguras por defecto.
Autorización	Pundit	install_pundit	Implementar un sistema de control de acceso basado en políticas para



			gestionar permisos de usuario.
Sanitización de Entradas	Sanitize	install_sanitize_and_configure	Prevenir XSS mediante la eliminación estricta de HTML en las entradas donde sea configurado.
Política de Seguridad de Contenido (CSP)	Rails Native CSP (config.content_security_policy)	install_csp_and_configure	Mitigar XSS y otros ataques de inyección restringiendo los recursos que el navegador puede cargar.
Cabeceras de Seguridad HTTP	Secure Headers	install_secure_headers	Configurar cabeceras HTTP adicionales para mejorar la seguridad (X-Frame-Options, X-Content-Type-Options, etc.).
Protección CSRF Reforzada	Rails Native CSRF Protection (protect_from_forgery)	install_csrf_protection	Asegurar que la protección CSRF esté activa con una estrategia de excepción.
Auditoría de Dependencias	Bundler-Audit	install_dependency_audit	Detectar vulnerabilidades conocidas en las gemas del proyecto mediante una tarea Rake.
Logging de Seguridad	Lograge, Custom ActiveSupport::Logger	install_security_logging	Mejorar el logging de peticiones y registrar eventos de seguridad específicos (e.g., intentos de login fallidos).

La interacción de estos componentes es orquestada por el **InstallGenerator**. Durante la ejecución de **bin/rails generate secure\_framework:install**, el generador evalúa la configuración existente de la aplicación y aplica las configuraciones de **secure\_framework** solo si es necesario, evitando duplicaciones o conflictos. Este enfoque idempotente es fundamental para una integración limpia y repetible.

## 6.2. Implementación de Componentes Clave

La implementación de los componentes de seguridad del **secure\_framework** se centraliza en el generador **SecureFramework::Generators::InstallGenerator**, definido en **install\_generator.rb**. Este generador automatiza la configuración de diversas facetas de seguridad en una aplicación Rails.

### 6.2.1. Orquestación de la Instalación (InstallGenerator)

El **InstallGenerator** hereda de **Rails::Generators::Base**, lo que le proporciona acceso a un conjunto de herramientas para la manipulación de archivos y la ejecución de comandos en el contexto de una aplicación Rails.

- **source\_root** `File.expand_path('templates', __dir__)`: Esta línea establece que los archivos plantilla que el generador utilizará (como inicializadores) se encuentran en el directorio **templates** relativo a la ubicación del propio generador.
- **install\_all\_components**: Este es el método público principal invocado cuando se ejecuta el generador. Su función es secuenciar la instalación de todos los componentes y, finalmente, imprimir un resumen con los pasos siguientes.
- **run\_component\_installers**: Método privado llamado dentro de **install\_all\_components**, que invoca secuencialmente los métodos de instalación para cada componente de seguridad. Cada uno de estos métodos primero verifica si el componente ya está configurado y actúa en consecuencia.
- **print\_final\_summary**: Método privado llamado dentro de **install\_all\_components**, justo después del método anterior (**run\_component\_installers**), para informar al usuario sobre las acciones realizadas y los pasos manuales que podrían ser necesarios post-instalación, como la edición de credenciales o la ejecución de migraciones.

Este diseño modular dentro del generador, donde cada componente tiene su propia lógica de chequeo e instalación, permite una gestión clara y mantenible del proceso de configuración.

### 6.2.2. Gestión Segura de Secretos

- **Propósito**: Asegurar que la clave maestra de Rails (**master.key**), utilizada para cifrar **credentials.yml.enc**, no sea accidentalmente versionada en el repositorio de código.
- **Método de Comprobación** (**secure\_credentials\_configured?**):

```
def secure_credentials_configured?  
  master_key_exists = File.exist?(File.join(destination_root,  
  'config/master.key'))  
  gitignore_path = File.join(destination_root, '.gitignore')  
  master_key_is_ignored = File.exist?(gitignore_path) &&  
  File.read(gitignore_path).include?('/config/master.key')  
  master_key_exists && master_key_is_ignored  
end
```

Esta función verifica dos condiciones: la existencia del archivo **config/master.key** y que la ruta **/config/master.key** esté presente en el archivo **.gitignore**. Ambas deben ser ciertas para considerar que la gestión segura de credenciales ya está configurada desde la perspectiva del framework.

- **Método de Instalación (install\_secure\_credentials):**

```
def install_secure_credentials
  @credentials_instructions_needed = true
  gitignore_path = File.join(destination_root, '.gitignore')
  key_entry = '/config/master.key'
  create_file(gitignore_path) unless File.exist?(gitignore_path)
  unless File.read(gitignore_path).include?(key_entry)
    say "Adding '#{key_entry}' to .gitignore...", :cyan
    append_to_file gitignore_path, "\n# Added by secure_framework
to protect secrets\n#{key_entry}\n"
  end
end
```

Si la comprobación falla, este método asegura que **config/master.key** sea ignorado por Git. Crea el archivo **.gitignore** si no existe y luego añade la entrada **/config/master.key** si aún no está presente. La variable de instancia **@credentials\_instructions\_needed** se activa para recordar al usuario que edite sus credenciales.

### 6.2.3. Autenticación (Devise)

- **Propósito:** Configurar la gema Devise para la autenticación de usuarios, aplicando políticas de seguridad mejoradas por defecto.
- **Método de Comprobación (devise\_configured?):**

```
def devise_configured?
  File.exist?(File.join(destination_root,
'config/initializers/devise.rb'))
end
```

Verifica simplemente la existencia del archivo inicializador de Devise.

- **Método de Instalación (install\_devise\_and\_configure):** Este método es multifacético:
  1. Ejecuta **generate "devise:install"** para la configuración base de Devise.
  2. Llama a **configure\_password\_policy** para modificar **config/initializers/devise.rb**, estableciendo **config.password\_length = 12..128** (anteriormente **6..128**).

3. Llama a **configure\_account\_locking** para habilitar y configurar el bloqueo de cuentas en **config/initializers/devise.rb** (**config.lock\_strategy = :failed\_attempts**, **config.maximum\_attempts = 5**, **config.unlock\_strategy = :time**, **config.unlock\_in = 30.minutes**).
4. Llama a **migrate\_secret\_to\_credentials** para configurar Devise de modo que utilice **Rails.application.credentials.dig(:devise, :secret\_key)** para su **secret\_key**, en lugar de un valor codificado. Esto también marca **@devise\_secret\_instruction\_needed = true**.
5. Ejecuta **generate "devise", "User", "username:string"** para crear un modelo de usuario **User** con un campo **username** (además de los campos estándar de Devise) y las migraciones asociadas.
6. Añade las rutas de Devise mediante **route "devise\_for :users"**.
7. Llama a **prepare\_user\_model\_for\_locking** para añadir el módulo **:lockable** al modelo **app/models/user.rb**.
8. Llama a **prepare\_migration\_for\_locking** para descomentar los campos y el índice necesarios para la funcionalidad **:lockable** en la migración de creación de usuarios de Devise. Estas acciones no solo instalan Devise, sino que lo configuran con parámetros de seguridad más estrictos que los predeterminados, como una longitud mínima de contraseña mayor y un bloqueo de cuenta más sensible.

#### 6.2.4. Autorización (Pundit)

- **Propósito:** Configurar la gema Pundit para la gestión de autorizaciones basada en políticas.
- **Método de Comprobación (pundit\_configured?):**

```
def pundit_configured?  
  File.exist?(File.join(destination_root,  
    'app/policies/application_policy.rb'))  
end
```

Verifica la existencia del archivo base de políticas de Pundit.

- **Método de Instalación (install\_pundit):**
1. Ejecuta **generate "pundit:install"** para crear la estructura básica de Pundit, incluyendo **app/policies/application\_policy.rb**.

2. Inyecta **include Pundit::Authorization** en la clase **ApplicationController** (**app/controllers/application\_controller.rb**) para hacer disponibles los métodos de Pundit en todos los controladores.

### 6.2.5. Sanitización de Entradas (Sanitize)

- **Propósito:** Establecer una configuración por defecto para la gema Sanitize que elimine todas las etiquetas HTML, ofreciendo una protección estricta contra XSS cuando se utiliza esta gema.
- **Método de Comprobación (sanitize\_configured?):**

```
def sanitize_configured?  
  File.exist?(File.join(destination_root,  
    'config/initializers/sanitize.rb'))  
end
```

Verifica la existencia del archivo inicializador de Sanitize.

- **Método de Instalación (install\_sanitize\_and\_configure):** Crea el archivo **config/initializers/sanitize.rb** con una configuración personalizada denominada **Sanitize::Config::SECURE\_FRAMEWORK**. Esta configuración es muy restrictiva:

```
Sanitize::Config::SECURE_FRAMEWORK = {  
  :elements =>,  
  :remove_contents => ['script', 'style']  
}
```

Esto significa que, por defecto, cualquier contenido procesado con esta configuración de Sanitize no permitirá ninguna etiqueta HTML y, adicionalmente, eliminará el contenido de las etiquetas **<script>** y **<style>** si alguna configuración externa las permitiera.

### 6.2.6. Política de Seguridad de Contenido (CSP)

- **Propósito:** Implementar una Política de Seguridad de Contenido robusta para mitigar ataques XSS y de inyección de datos.
- **Método de Comprobación (csp\_configured?):**

```
def csp_configured?
```

```
initializer_path = File.join(destination_root,  
'config/initializers/content_security_policy.rb')  
return false unless File.exist?(initializer_path)  
content = File.read(initializer_path)  
content.match?(/^s*policy\./)  
end
```

Verifica no solo la existencia del archivo `config/initializers/content_security_policy.rb`, sino también que contenga líneas activas de definición de políticas (que comiencen con `policy.`).

- **Método de Instalación (`install_csp_and_configure`):** Elimina el archivo de inicialización de CSP existente (si lo hay) y crea uno nuevo con una política segura predefinida. Esta política incluye directivas como:
  - `policy.default_src :self, :https`
  - `policy.font_src :self, :https, :data`
  - `policy.img_src :self, :https, :data`
  - `policy.object_src :none`
  - `policy.script_src :self, :https,`  
`'sha256-DpVt+Ev/ITHBvE9AP6MusgWkuqGvLlkqNGv2dwHVOyE=','`  
`'sha256-Bk2Ki1XPeMQgcV8U6q5OUXYdrX/47R4L1F0tatGpT7w='` (los hashes definidos se utilizan para permitir de forma segura la ejecución de los scripts específicos de **turbo-rails**, necesarios para funcionalidades como ``data-turbo-confirm``).
  - `policy.style_src :self, :https` También incluye configuraciones para permitir la recarga en caliente de Vite en desarrollo y configura un generador de nonces para scripts. La intención es proporcionar una base sólida que los desarrolladores puedan ajustar.

### 6.2.7. Cabeceras de Seguridad (Secure Headers)

- **Propósito:** Configurar la gema `secure_headers` para aplicar diversas cabeceras HTTP que mejoran la seguridad.
- **Método de Comprobación (`secure_headers_configured?`):**

```
def secure_headers_configured?  
  File.exist?(File.join(destination_root,
```

```
'config/initializers/secure_headers.rb'))  
end
```

Verifica la existencia del archivo inicializador de **secure\_headers**.

- **Método de Instalación (install\_secure\_headers):** Crea el archivo **config/initializers/secure\_headers.rb** con una configuración predeterminada que establece:
  - **config.x\_frame\_options = "SAMEORIGIN"** (protección contra clickjacking)
  - **config.x\_content\_type\_options = "nosniff"** (previene el MIME-type sniffing)
  - **config.x\_xss\_protection = "1; mode=block"** (activa el filtro XSS de navegadores antiguos)
  - **config.referrer\_policy = %w(origin-when-cross-origin strict-origin-when-cross-origin)**
  - **config.csp = SecureHeaders::OPT\_OUT:** Es importante destacar que se desactiva explícitamente la gestión de CSP por parte de la gema **secure\_headers**. Esto se hace para evitar conflictos con la configuración nativa de CSP de Rails gestionada por **install\_csp\_and\_configure**, promoviendo una única fuente de verdad para la CSP.

### 6.2.8. Protección CSRF Reforzada

- **Propósito:** Asegurar que la protección contra Cross-Site Request Forgery (CSRF) esté habilitada con la estrategia de **:exception**.
- **Método de Comprobación (csrf\_protection\_configured?):**

```
def csrf_protection_configured?  
  controller_path = File.join(destination_root, 'app',  
    'controllers', 'application_controller.rb')  
  return false unless File.exist?(controller_path)  
  File.read(controller_path).match?(/protect_from_forgery/)  
end
```

Verifica que **ApplicationController** contenga una llamada a **protect\_from\_forgery**.

- **Método de Instalación (install\_csrf\_protection):** Inyecta la línea `protect_from_forgery with: :exception` justo después de la declaración de la clase `ApplicationController < ActionController::Base`. Usar `:exception` es una práctica más segura que `:null_session` ya que detiene la acción en caso de un token CSRF inválido.

### 6.2.9. Auditoría de Dependencias (Bundler-Audit)

- **Propósito:** Proveer una tarea Rake para escanear las dependencias del proyecto en busca de vulnerabilidades conocidas.
- **Método de Comprobación (dependency\_audit\_configured?):**

```
def dependency_audit_configured?  
  File.exist?(File.join(destination_root,  
    'lib/tasks/dependency_audit.rake'))  
end
```

Verifica la existencia de la tarea Rake de auditoría.

- **Método de Instalación (install\_dependency\_audit):** Crea el archivo `lib/tasks/dependency_audit.rake`. Esta tarea, `dependency_audit:check`, utiliza `bundler/audit` para:
  1. Actualizar la base de datos de vulnerabilidades (`Bundler::Audit::Database.update!`).
  2. Escanear el `Gemfile.lock` (`Bundler::Audit::Scanner.new.scan`).
  3. Imprimir las vulnerabilidades encontradas con detalles (gema, aviso, criticidad, solución).
  4. Si se encuentran vulnerabilidades, termina con un código de salida no cero (`abort`), lo cual es útil para fallar construcciones en entornos de Integración Continua (CI).

### 6.2.10. Logging de Seguridad (Lograge y Custom Logger)

- **Propósito:** Mejorar el formato de los logs de Rails para que sean más legibles y útiles, y establecer un logger dedicado para eventos de seguridad.
- **Método de Comprobación (security\_logging\_configured?):**

```
def security_logging_configured?  
  lograge_exists = File.exist?(File.join(destination_root,
```



```
'config/initializers/lograge.rb'))  
  security_logger_exists = File.exist?(File.join(destination_root,  
'config/initializers/security_logging.rb'))  
  lograge_exists && security_logger_exists  
end
```

Verifica la existencia de ambos archivos inicializadores, **lograge.rb** y **security\_logging.rb**.

- **Método de Instalación (install\_security\_logging):**

1. Crea **config/initializers/lograge.rb**: Habilita Lograge y configura **custom\_options** para incluir **request\_id**, **remote\_ip** y **user\_id** en cada línea de log, facilitando el rastreo y análisis.
2. Crea **config/initializers/security\_logging.rb**:
  - Define **SECURITY\_LOGGER** como una instancia de **ActiveSupport::Logger** que escribe en **log/security.log**.
  - Establece un formato personalizado para este logger.
  - Utiliza **Warden::Manager.before\_failure** (un hook de Devise/Warden) para registrar intentos de inicio de sesión fallidos en **SECURITY\_LOGGER**. El mensaje de log incluye el evento, email, IP, user agent, path y el mensaje de Warden, en formato JSON. La variable **@security\_logging\_instructions\_needed** (utilizada en la función **print\_final\_summary** que vimos previamente) se activa para guiar al usuario en la configuración adicional manual, como el manejo de excepciones de Pundit para logging.

## 6.3. Guía de Instalación y Uso Básico del *Framework*

Esta sección proporciona las instrucciones para instalar **secure\_framework** en una aplicación Ruby on Rails y una guía básica para utilizar sus componentes.

### 6.3.1. Prerrequisitos

- Una aplicación Ruby on Rails existente o recién creada.
- Bundler instalado y utilizado para la gestión de gemas del proyecto.
- Ruby y Rails en versiones compatibles con las gemas de seguridad que instala el framework (e.g., Devise, Pundit).

### 6.3.2. Proceso de Instalación

1. **Añadir la gema al Gemfile:** Abrir el archivo **Gemfile** de la aplicación Rails y añadir la siguiente línea:

```
gem 'secure_framework' #0 gem 'secure_framework', path:  
'../secure_framework', según donde se haya guardado La gema
```

2. **Instalar la gema:** Ejecutar en la terminal, desde el directorio raíz de la aplicación Rails:

```
bundle install
```

3. **Ejecutar el generador de instalación:** Una vez instalada la gema, ejecutar el generador proporcionado por **secure\_framework**:

```
bin/rails generate secure_framework:install
```

El generador mostrará mensajes de progreso en la consola, indicando qué componentes se están configurando o cuáles se están omitiendo si ya existen configuraciones previas. Por ejemplo, se observarán mensajes como **"Secure Credentials management is already configured. Skipping."** (en verde) o **"Devise not configured yet. Running Devise setup..."** (en cian), gracias al uso de `say "...", :color` en `install_generator.rb`.

### 6.3.3. Configuración Inicial y Pasos Post-Instalación

Tras la ejecución del generador, es crucial revisar la salida del método **print\_final\_summary** en la consola y realizar los pasos manuales indicados. Estos pasos son esenciales para completar la configuración de seguridad. La siguiente tabla resume estas acciones críticas.

**Tabla 6.2.** Resumen de Pasos Críticos Post-Instalación del `secure_framework`

Paso	Descripción/Acción Requerida	Justificación/Importancia para la Seguridad
Gestión de Credenciales	Ejecutar <code>EDITOR=vim bin/rails credentials:edit</code> para establecer la <code>secret_key</code> para Devise (si se instaló Devise) o para gestionar otros secretos de la aplicación.	Asegura que secretos críticos como la <code>secret_key</code> de Devise (necesaria si se migró a <code>credentials.yml.enc</code> ) y otras claves API se almacenen de forma cifrada y no se expongan en el código fuente.
Migración de Base de Datos	Ejecutar <code>rails db:migrate</code> .	Aplica los cambios en el esquema de la base de datos necesarios para Devise, especialmente los campos para la funcionalidad de bloqueo de cuentas ( <code>failed_attempts</code> , <code>unlock_token</code> , <code>locked_at</code> ) si Devise fue configurado.
Autenticación de Controladores	Proteger los controladores relevantes añadiendo <code>before_action :authenticate_user!</code> donde se requiera autenticación.	Paso manual esencial para activar la protección de Devise en las rutas y acciones que no deben ser accesibles públicamente. El framework no puede determinar qué controladores específicos requieren autenticación.
Configuración de Lograge	Revisar la instrucción para deshabilitar Lograge en entornos específicos (e.g., desarrollo) si se desea, añadiendo el bloque de código sugerido a <code>config/environments/&lt;environment&gt;.rb</code> .	Lograge se habilita por defecto para todos los entornos. Esta flexibilidad permite al desarrollador desactivarlo si el formato detallado no es deseado en ciertos contextos, como el desarrollo local.
Logging de Autorización (Pundit)	Añadir el bloque de código proporcionado al <code>ApplicationController</code> para rescatar <code>Pundit::NotAuthorizedError</code> y registrar el evento en <code>SECURITY_LOGGER</code> .	Esencial para auditar y monitorear los intentos de acceso no autorizado. Permite registrar quién intentó acceder a qué recurso sin permiso, lo cual es vital para la detección de incidentes y el análisis forense.

La necesidad de estos pasos manuales subraya un diseño pragmático. Si bien **secure\_framework** automatiza una gran cantidad de configuraciones, ciertas acciones requieren la intención explícita del desarrollador (e.g., decidir qué controladores proteger) o

involucran la gestión de datos sensibles (e.g., la edición de credenciales). El framework guía al desarrollador a través de estos pasos para asegurar una configuración completa y segura.

## 7. Validación y Pruebas

Este capítulo se dedica a la validación empírica y a las pruebas exhaustivas del marco de seguridad propuesto. Para llevar a cabo esta validación, se ha desarrollado una aplicación de demostración que sirve como un entorno controlado para implementar y evaluar las distintas capas de seguridad. A través de esta aplicación, se demostrará la eficacia de las herramientas, configuraciones y políticas de seguridad descritas en los capítulos anteriores, sometiéndola a una serie de pruebas de seguridad y análisis de vulnerabilidades.

### 7.1. Diseño de la Aplicación de Demostración

Para validar y demostrar la eficacia de los principios y herramientas que componen el **secure\_framework**, se ha desarrollado una aplicación de demostración denominada **demo\_app**. Esta sección describe en detalle su arquitectura, tecnología, modelo de datos y, fundamentalmente, la implementación de cada uno de los componentes de seguridad propuestos.

#### 7.1.1. Introducción a demo\_app

**demo\_app** es una aplicación web monolítica construida sobre el framework **Ruby on Rails** 7. Su propósito principal no es ofrecer una solución comercial, sino actuar como una **prueba de concepto (PoC)** y un campo de pruebas práctico. En esta aplicación, se integran y configuran las políticas, herramientas y contramedidas de seguridad que constituyen el núcleo de este trabajo. La funcionalidad principal de **demo\_app** es un sistema de gestión de publicaciones (un blog simple), donde los usuarios pueden registrarse, iniciar sesión y realizar operaciones de creación, lectura, actualización y eliminación (CRUD) sobre sus propios posts. Esta simplicidad funcional permite centrar el análisis en la robustez de su arquitectura de seguridad.

### 7.1.2. Arquitectura

La aplicación **demo\_app** se adhiere estrictamente al patrón de arquitectura

**Modelo-Vista-Controlador (MVC)**, que es fundamental en el diseño de aplicaciones **Ruby on Rails**. Este patrón desacopla las responsabilidades de la lógica de negocio, la presentación y la gestión de las interacciones del usuario, lo que resulta en un código más organizado, mantenible y escalable.

- **Modelo:** El componente del Modelo es el responsable de representar la lógica de negocio y los datos de la aplicación. En el directorio **app/models/**, se definen las clases que interactúan directamente con la base de datos.
  - **ApplicationRecord:** Es la clase base de la que heredan todos los modelos de la aplicación, definida en **app/models/application\_record.rb**. A su vez, hereda de **ActiveRecord::Base**, proporcionando toda la funcionalidad del ORM (Object-Relational Mapping) de Rails.
  - **User:** El modelo **user.rb** representa a los usuarios del sistema. Gestiona atributos como el correo electrónico, la contraseña encriptada y el rol. Además, integra varios módulos de **Devise** para la autenticación y define la asociación con el modelo **Post**.
  - **Post:** El modelo **post.rb** representa las publicaciones del blog. Contiene los campos para el título y el contenido, y establece una relación de pertenencia a un usuario a través de la asociación **belongs\_to :user**.
- **Vista:** Las vistas se encargan de la capa de presentación, generando la interfaz de usuario (UI) que se envía al navegador. Los ficheros se localizan en el directorio **app/views/**. **demo\_app** utiliza plantillas **ERB (Embedded Ruby)**, que son ficheros HTML con código Ruby incrustado. Este código se ejecuta en el servidor para generar dinámicamente el contenido HTML final, permitiendo mostrar datos de los modelos y construir interfaces complejas.
- **Controlador:** Los Controladores actúan como intermediarios entre los Modelos y las Vistas. Procesan las solicitudes HTTP entrantes, interactúan con los modelos para consultar o modificar datos y, finalmente, seleccionan la vista adecuada para renderizar la respuesta. Se encuentran en el directorio **app/controllers/**.
  - **ApplicationController:** Es el controlador base, definido en **app/controllers/application\_controller.rb**. Incluye la funcionalidad de

**Pundit** (include **Pundit::Authorization**) y define filtros **before\_action** como **authenticate\_user!**, que aseguran que un usuario haya iniciado sesión antes de acceder a ciertas partes de la aplicación.

- **HomeController**: Gestiona la página de inicio pública de la aplicación.
- **DashboardController**: Controla el panel principal al que acceden los usuarios tras iniciar sesión, donde pueden ver las publicaciones.
- **PostsController**: Implementa la lógica **CRUD (Create, Read, Update, Delete)** para las publicaciones, gestionando la creación, visualización, edición y eliminación de los posts, y aplicando las políticas de autorización correspondientes.

### 7.1.3. Modelo de Datos y Base de Datos

La estructura de la base de datos subyacente se define en el fichero **db/schema.rb**. Este fichero es una representación autoritativa del esquema actual de la base de datos, generado automáticamente por Rails a partir de las migraciones ejecutadas. La aplicación consta de dos tablas principales: **users** y **posts**.

- **Tabla users**: Esta tabla almacena la información esencial para la gestión de usuarios y la autenticación. Sus campos más relevantes son:
  - **email** y **encrypted\_password**: Pilares para la autenticación basada en credenciales (**Devise**).
  - **role**: Un campo de tipo entero, con un valor por defecto de **0**, que implementa un sistema de roles simple (**0** para usuario estándar, **1** para administrador). Es crucial para la lógica de autorización.
  - Campos de seguimiento de **Devise** (**sign\_in\_count**, **current\_sign\_in\_at**, **last\_sign\_in\_at**, **current\_sign\_in\_ip**, **last\_sign\_in\_ip**): Permiten auditar la actividad de la sesión del usuario.
  - Campos de bloqueo de cuenta (**failed\_attempts**, **unlock\_token**, **locked\_at**): Utilizados por el módulo **:lockable** de **Devise** para prevenir ataques de fuerza bruta.
- **Tabla posts**: Almacena el contenido generado por los usuarios. Sus campos son:
  - **title** y **content**: Guardan el título y el cuerpo de cada publicación.

- **user\_id**: Una clave foránea (**foreign\_key**) que establece una relación directa con la tabla **users**. Garantiza la integridad referencial, asegurando que cada post pertenezca a un usuario existente.

La relación entre estas tablas es de uno a muchos: un usuario puede tener muchas publicaciones (**has\_many :posts**, definido en **user.rb**), mientras que una publicación pertenece a un único usuario (**belongs\_to :user**, definido en **post.rb**).

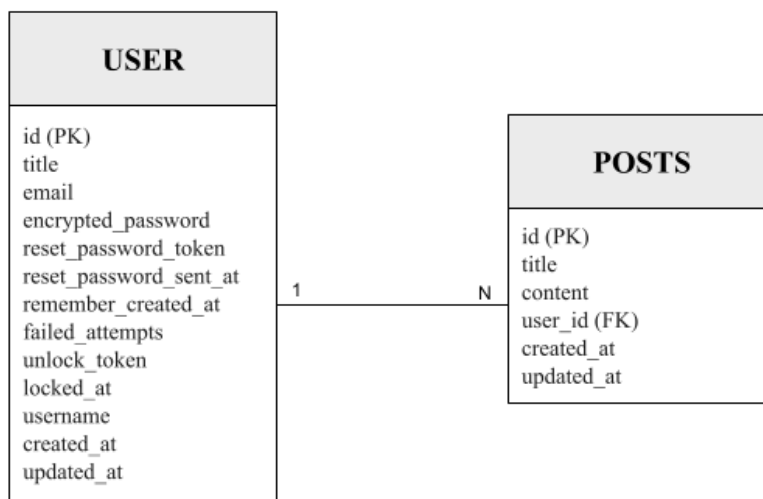


Figura 7.1: Diagrama Entidad-Relación de la Base de Datos

#### 7.1.4. Flujo de Autenticación y Gestión de Sesiones

La autenticación en **demo\_app** es gestionada íntegramente por la gema **Devise** (gema instalada por **secure\_framework**). La configuración se centraliza en el modelo **User** y en el inicializador de la gema.

En el fichero **app/models/user.rb**, se declaran los módulos de **Devise** que se van a utilizar:

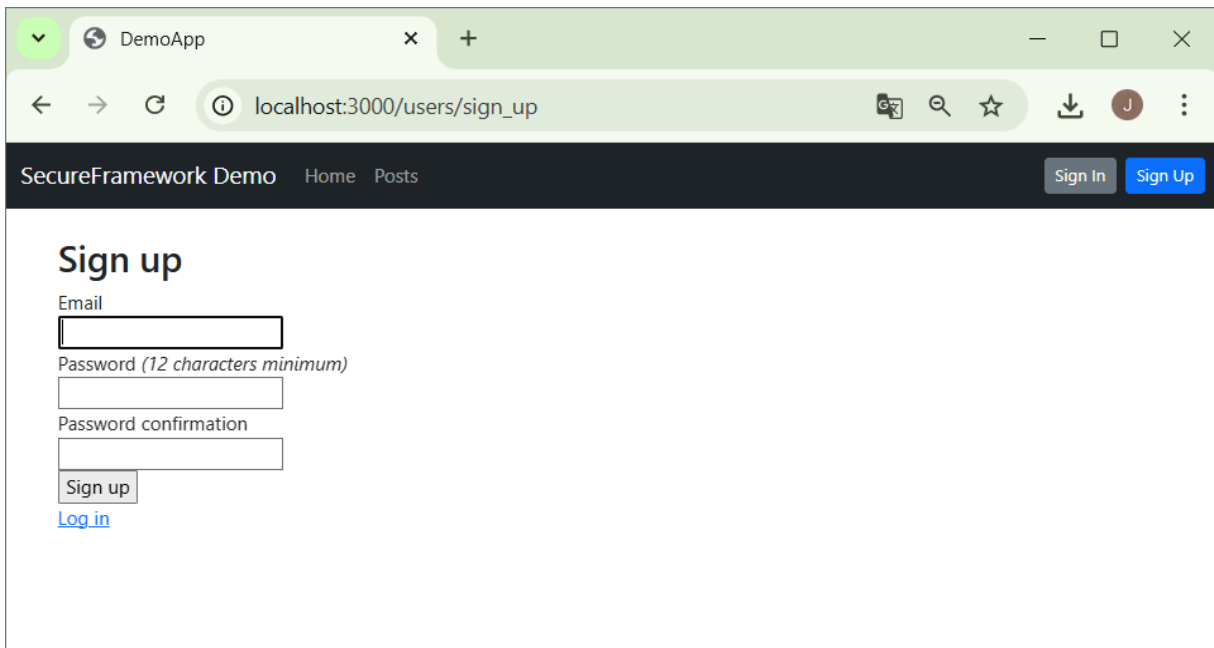
- **:database\_authenticatable**: Gestiona el proceso de hasheo y almacenamiento de contraseñas y la autenticación del usuario al iniciar sesión.
- **:registerable**: Habilita la funcionalidad de registro de nuevos usuarios y la edición de sus perfiles.
- **:recoverable**: Proporciona un mecanismo para que los usuarios puedan restablecer su contraseña si la olvidan.
- **:rememberable**: Gestiona la creación de tokens para mantener la sesión del usuario activa entre visitas ("Recordarme").

- **:validatable**: Ofrece validaciones automáticas para el formato del email y la complejidad de la contraseña.
- **:trackable**: Rastrea metadatos de inicio de sesión, como el número de accesos, las fechas y las direcciones IP.
- **:lockable**: Bloquea la cuenta de un usuario después de un número determinado de intentos de inicio de sesión fallidos como medida de seguridad.

El fichero de configuración **config/initializers/devise.rb**, contiene las configuraciones predeterminadas instaladas por **secure\_framework**, y en este se puede ajustar el comportamiento de estos módulos. Para **demo\_app**, **secure\_framework** ha establecido configuraciones de seguridad relevantes:

- **config.password\_length = 12..128**: Se exige una longitud mínima de 12 caracteres para las contraseñas, promoviendo credenciales más fuertes.
- **config.maximum\_attempts = 5**: La cuenta de un usuario se bloqueará tras 5 intentos fallidos de inicio de sesión, mitigando el riesgo de ataques de fuerza bruta.

El flujo de autenticación sigue los patrones estándar proporcionados por **Devise**: un usuario puede registrarse a través de un formulario, iniciar sesión con su email y contraseña, y solicitar un enlace para recuperar su contraseña, que se envía por correo electrónico.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/users/sign\_up'. The page title is 'SecureFramework Demo'. The navigation bar includes links for 'Home' and 'Posts', and buttons for 'Sign In' and 'Sign Up'. The main content area is titled 'Sign up' and contains three input fields: 'Email', 'Password (12 characters minimum)', and 'Password confirmation'. Below the fields are buttons for 'Sign up' and 'Log in'.

**Figura 7.2:** Vista de Registro de Usuario



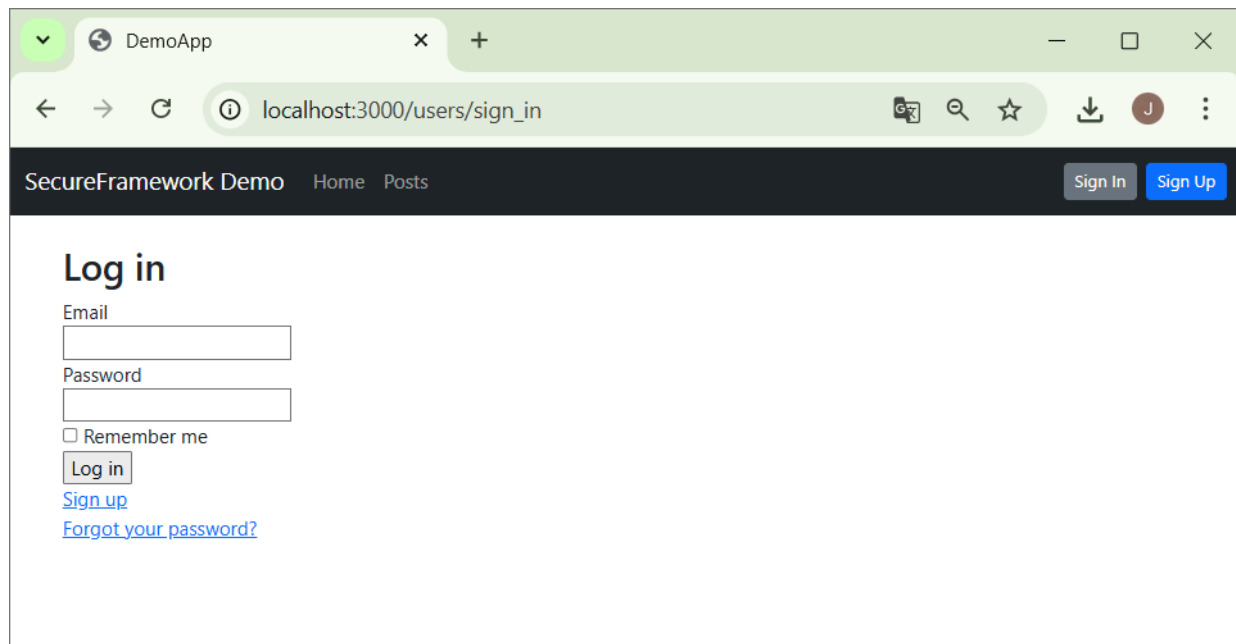


Figura 7.3: Vista de Inicio de Sesión

### 7.1.5. Sistema de Autorización y Roles de Usuario

Mientras que **Devise** gestiona la autenticación (quién es el usuario), la gema **Pundit** se encarga de la autorización (qué puede hacer el usuario). **Pundit** implementa un sistema basado en objetos de política (**policies**) que encapsulan la lógica de permisos de forma clara y reutilizable.

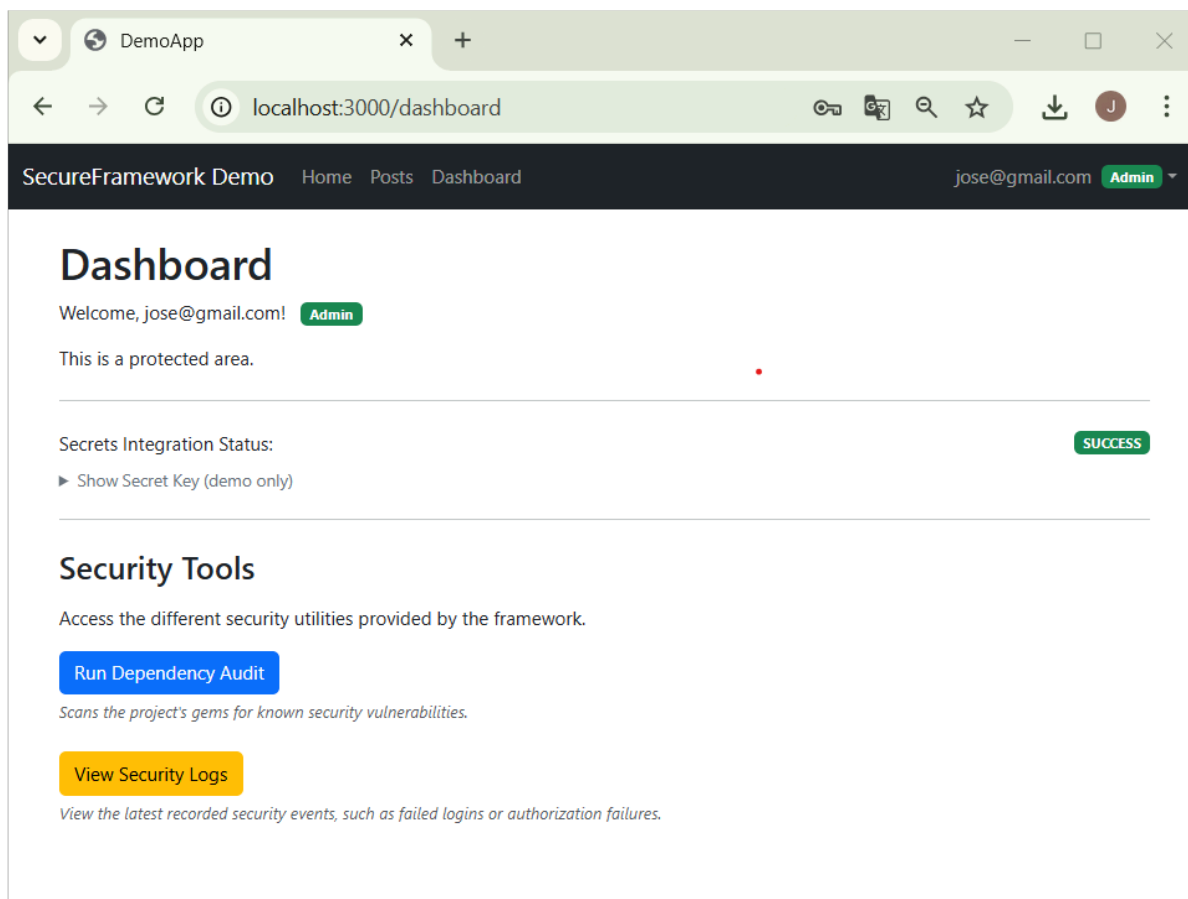
El concepto central son las clases de política, ubicadas en **app/policies/**. Cada política se corresponde con un modelo y define métodos de consulta (terminados en **?**) para determinar si un usuario (**user**) puede realizar una acción sobre un recurso (**record**).

El fichero **app/policies/post\_policy.rb** es un ejemplo central de esta implementación:

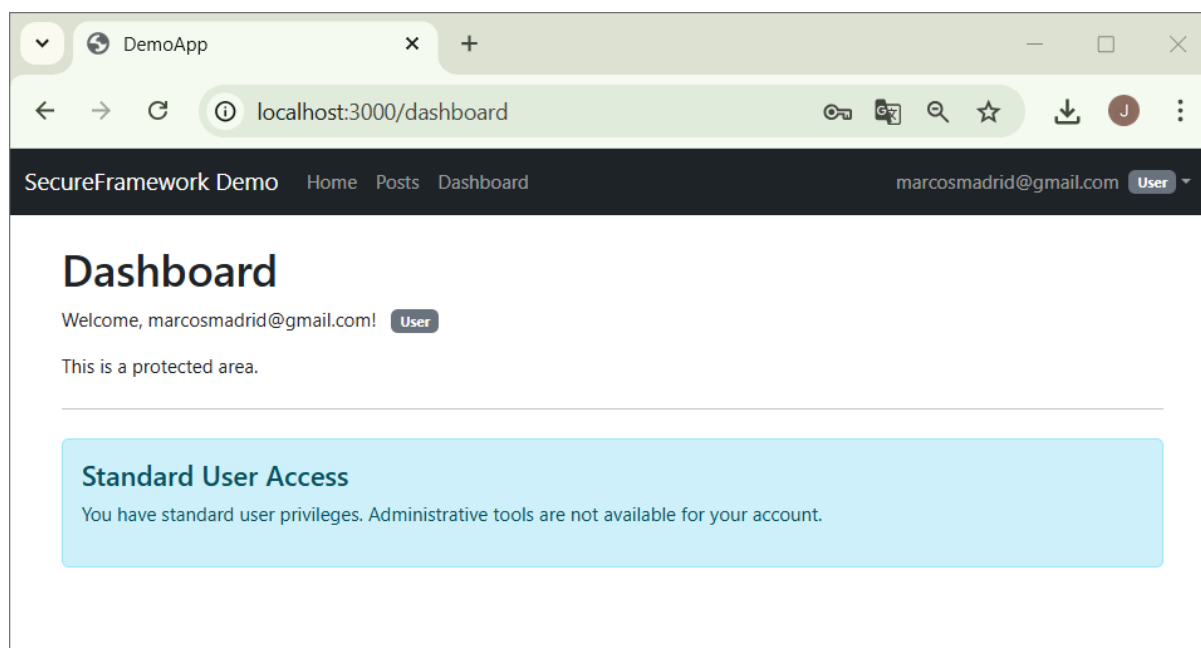
- Los métodos **update?**, **edit?** y **destroy?** definen la lógica de control de acceso para modificar o eliminar un post. La regla es estricta: la acción solo se permite si el usuario actual es el propietario del post (**record.user == user**) o si es un administrador (**user.admin?**).
- La clase anidada **PostPolicy::Scope** se utiliza para filtrar colecciones de registros. Su método **resolve** define qué posts debe ver un usuario en una vista de listado como el dashboard. Si el usuario es administrador, el **scope** devuelve todos los posts

(**scope.all**); de lo contrario, devuelve únicamente los posts que pertenecen al usuario (**scope.where(user: user)**).

Esta lógica se invoca en el **PostsController** mediante el método **authorize @post**, que Pundit proporciona. Este método busca automáticamente la política correspondiente al objeto **@post** y ejecuta el método de consulta asociado a la acción del controlador (ej. **update?** para la acción **update**), lanzando una excepción si la autorización es denegada.



**Figura 7.4:** Dashboard del Administrador con Privilegios Elevados



**Figura 7.5:** Dashboard de un Usuario Estándar

### 7.1.6. Rutas y Controladores Principales

El sistema de enrutamiento de Rails, definido en **config/routes.rb**, mapea las URLs a las acciones de los controladores. La estructura de navegación de **demo\_app** es la siguiente:

- root "home#index": Establece la ruta raíz de la aplicación (/) para que sea gestionada por la acción index del **HomeController**.
- devise\_for :users: Este helper de **Devise** genera automáticamente todas las rutas necesarias para la autenticación: inicio de sesión (/users/sign\_in), registro (/users/sign\_up), cierre de sesión, recuperación de contraseña, etc.
- get "dashboard", to: "dashboard#index": Define una ruta personalizada para el panel de usuario.
- resources :posts: Este método genera el conjunto completo de rutas RESTful para el recurso **Post**, incluyendo las rutas para las acciones index, show, new, create, edit, update y destroy.

Los controladores principales orquestan el flujo de la aplicación: **HomeController** para la vista pública, **DashboardController** para el panel privado post-autenticación y **PostsController** para toda la gestión de las publicaciones, aplicando las políticas de seguridad en cada acción relevante.

## 7.2. Pruebas de Seguridad Aplicadas

Para validar la eficacia y robustez del **secure\_framework**, se ha sometido a **demo\_app** a un riguroso proceso de pruebas de seguridad. La estrategia adoptada sigue un enfoque de "**defensa en profundidad**", que no se limita a una única capa de protección, sino que combina múltiples tácticas de aseguramiento. Se han integrado herramientas de **Análisis Estático de Código (SAST)** como primera línea de defensa para detectar vulnerabilidades en el código fuente, junto con un completo conjunto de **pruebas funcionales de seguridad** que simulan escenarios de ataque reales. Las herramientas seleccionadas para este propósito han sido **Brakeman**, para el análisis de vulnerabilidades específicas de Rails; **Rubocop**, para garantizar la calidad y el cumplimiento de estándares del código; y **Rspec**, para la ejecución de pruebas de comportamiento que validan la correcta implementación de los mecanismos de seguridad en tiempo de ejecución. Los resultados obtenidos, como se detallará a continuación, son excepcionales: cero advertencias de seguridad, cero ofensas de estilo de código y cero fallos en las pruebas funcionales, lo que demuestra la solidez de la aplicación y, por extensión, del framework subyacente.

### 7.2.1. Análisis Estático de Código (SAST)

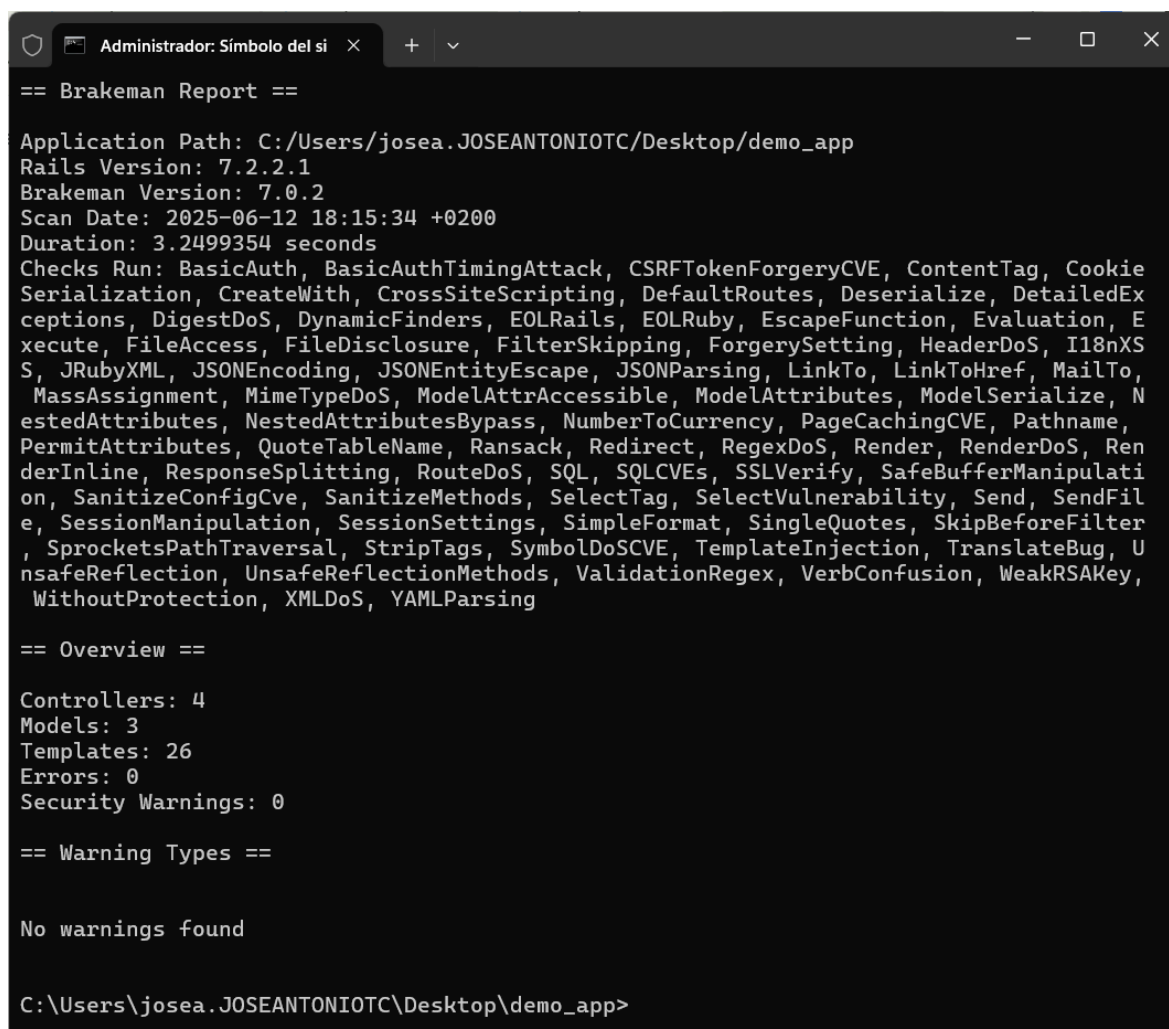
El Análisis Estático de Pruebas de Seguridad (SAST, por sus siglas en inglés) constituye una fase fundamental en el ciclo de vida de desarrollo seguro. Su principal ventaja radica en la capacidad de inspeccionar el 100% del código fuente en busca de patrones de vulnerabilidades conocidos y fallos de codificación antes de que la aplicación sea compilada o ejecutada. Este enfoque proactivo permite identificar y remediar problemas de seguridad en las etapas más tempranas del desarrollo, reduciendo drásticamente el coste de la corrección y el riesgo de exposición.

#### Detección de Vulnerabilidades con Brakeman

**Brakeman** es un analizador de seguridad estático de código abierto, diseñado específicamente para aplicaciones Ruby on Rails. A diferencia de otras herramientas genéricas, Brakeman posee un conocimiento profundo de la arquitectura y las particularidades del framework Rails, lo que le permite detectar con gran precisión vulnerabilidades comunes como inyección de SQL, Cross-Site Scripting (XSS), falsificación

de peticiones en sitios cruzados (CSRF), redirecciones inseguras y exposición de datos sensibles.

La ejecución de Brakeman sobre el código fuente de la aplicación de demostración arrojó el siguiente resultado: **Security Warnings: 0**.



```
== Brakeman Report ==

Application Path: C:/Users/josea.JOSEANTONIOTC/Desktop/demo_app
Rails Version: 7.2.2.1
Brakeman Version: 7.0.2
Scan Date: 2025-06-12 18:15:34 +0200
Duration: 3.2499354 seconds
Checks Run: BasicAuth, BasicAuthTimingAttack, CSRFTokenForgeryCVE, ContentTag, Cookie
Serialization, CreateWith, CrossSiteScripting, DefaultRoutes, Deserialize, DetailedEx
ceptions, DigestDoS, DynamicFinders, EOLRails, EOLRuby, EscapeFunction, Evaluation, E
xecute, FileAccess, FileDisclosure, FilterSkipping, ForgerySetting, HeaderDoS, I18nXS
S, JRubyXML, JSONEncoding, JSONEntityEscape, JSONParsing, LinkTo, LinkToHref, MailTo,
MassAssignment, MimeTypeDoS, ModelAttrAccessible, ModelAttributes, ModelSerialize, N
estedAttributes, NestedAttributesBypass, NumberToCurrency, PageCachingCVE, Pathname,
PermitAttributes, QuoteTableName, Ransack, Redirect, RegexDoS, Render, RenderDoS, Ren
derInline, ResponseSplitting, RouteDoS, SQL, SQLCVEs, SSLVerify, SafeBufferManipulati
on, SanitizeConfigCve, SanitizeMethods, SelectTag, SelectVulnerability, Send, SendFil
e, SessionManipulation, SessionSettings, SimpleFormat, SingleQuotes, SkipBeforeFilter
, SprocketsPathTraversal, StripTags, SymbolDoSCVE, TemplateInjection, TranslateBug, U
nsafeReflection, UnsafeReflectionMethods, ValidationRegex, VerbConfusion, WeakRSAKey,
WithoutProtection, XMLDoS, YAMLParsing

== Overview ==

Controllers: 4
Models: 3
Templates: 26
Errors: 0
Security Warnings: 0

== Warning Types ==

No warnings found

C:\Users\josea.JOSEANTONIOTC\Desktop\demo_app>
```

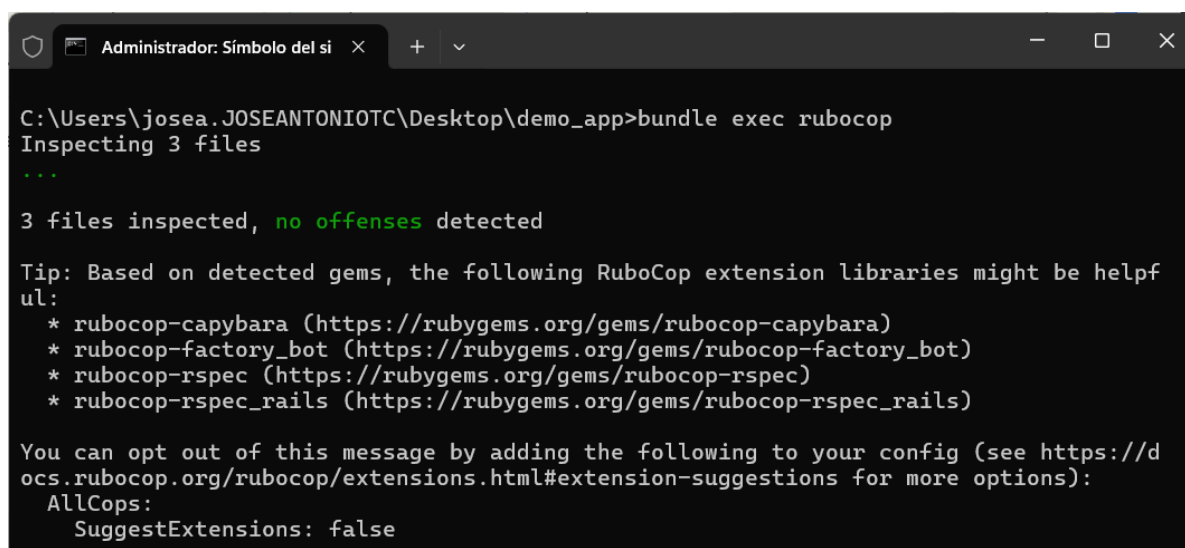
**Figura 7.6:** Informe de resultados de la ejecución de Brakeman

Este resultado es concluyente y de gran relevancia. Indica que, tras un análisis exhaustivo de todos los modelos, vistas y controladores de la aplicación, Brakeman no ha encontrado ninguna de las vulnerabilidades que figuran en su amplia base de datos de amenazas. Este éxito no es casual, sino una consecuencia directa de haber edificado la aplicación sobre los cimientos del **secure\_framework**. Las configuraciones seguras por defecto, las gemas de seguridad preinstaladas y las prácticas de codificación promovidas por el framework han eliminado de raíz las debilidades que Brakeman busca activamente.

## Calidad y Estilo de Código con Rubocop

**Rubocop** es un analizador estático de código (o *linter*) para el lenguaje Ruby. Su función principal es verificar que el código fuente se adhiere a las guías de estilo y buenas prácticas definidas por la comunidad Ruby. Aunque su enfoque no es la seguridad en sí, su papel en la prevención de vulnerabilidades es fundamental y a menudo subestimado. Un código limpio, ordenado, legible y consistente es más fácil de mantener, de revisar por pares y, crucialmente, menos propenso a contener errores lógicos que puedan ser explotados por un atacante.

El análisis de la aplicación de demostración con Rubocop concluyó con el mensaje: **no offenses detected**.



```
C:\Users\josea.JOSEANTONIO\TC\Desktop\demo_app>bundle exec rubocop
Inspecting 3 files
...
3 files inspected, no offenses detected

Tip: Based on detected gems, the following RuboCop extension libraries might be helpful:
* rubocop-capybara (https://rubygems.org/gems/rubocop-capybara)
* rubocop-factory_bot (https://rubygems.org/gems/rubocop-factory_bot)
* rubocop-rspec (https://rubygems.org/gems/rubocop-rspec)
* rubocop-rspec_rails (https://rubygems.org/gems/rubocop-rspec_rails)

You can opt out of this message by adding the following to your config (see https://docs.rubocop.org/rubocop/extensions.html#extension-suggestions for more options):
AllCops:
  SuggestExtensions: false
```

Figura 7.7: Informe de resultados de la ejecución de Rubocop

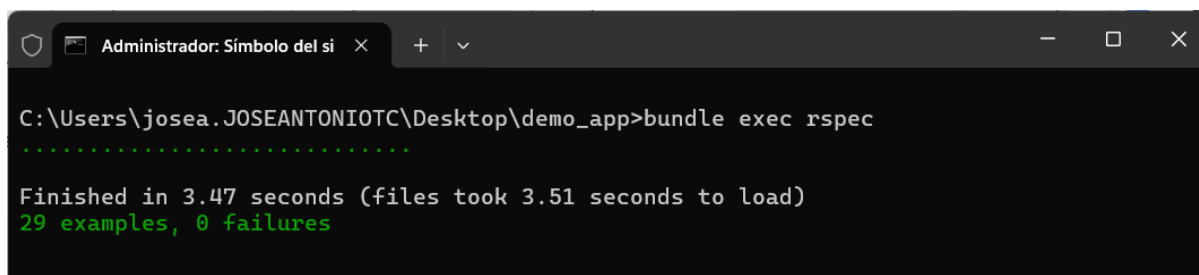
Este resultado demuestra un alto grado de disciplina y calidad en el desarrollo. Significa que todo el código de la aplicación cumple con los estándares de la industria, lo que facilita su escalabilidad y reduce la probabilidad de introducir fallos de seguridad en futuras iteraciones. La integración de Rubocop en el flujo de trabajo que promueve el **secure\_framework** es, por tanto, una medida de seguridad preventiva indirecta pero de gran valor.

### 7.2.2. Pruebas Funcionales de Seguridad con Rspec

Mientras que el análisis SAST inspecciona el código en reposo, las pruebas funcionales de seguridad validan el comportamiento de la aplicación en tiempo de ejecución. Este enfoque es indispensable para confirmar que las contramedidas de seguridad no solo están presentes, sino que funcionan como se espera cuando un usuario (o un atacante) interactúa con el

sistema. Para este fin, se ha utilizado **Rspec**, un framework de pruebas de comportamiento (BDD) para Ruby, que permite describir las defensas de la aplicación en un lenguaje claro y ejecutar escenarios que las pongan a prueba.

La ejecución completa de la suite de pruebas sobre la aplicación de demostración, mediante el comando **bundle exec rspec**, concluyó con un resultado impecable: **29 ejemplos, 0 fallos**. Este resultado global es la primera y más importante validación: cada una de las 29 especificaciones de seguridad y funcionalidad definidas se comporta exactamente como se espera, sin regresiones ni errores.



```
Administrador: Símbolo del si x + v
C:\Users\josea.JOSEANTONIOTC\Desktop\demo_app>bundle exec rspec
.....
Finished in 3.47 seconds (files took 3.51 seconds to load)
29 examples, 0 failures
```

Figura 7.8: Informe de resultados de la ejecución de Rubocop

Lejos de ser una simple comprobación de errores, esta suite de pruebas implementa un verdadero "test de penetración automatizado" que valida las defensas del framework en múltiples dominios de seguridad. La totalidad de las pruebas funcionales y de seguridad mencionadas a continuación se encuentran centralizadas en el directorio **spec/** del proyecto, siguiendo las convenciones de la comunidad para la organización de los tests:

## 1. Gestión de Identidad y Acceso

Para validar la robustez y seguridad del ciclo de vida completo del usuario, se ha ejecutado una suite de pruebas funcionales que cubren desde el registro y la autenticación hasta el bloqueo de cuentas y la recuperación de contraseñas.

**Autenticación y Sesiones (user\_session\_spec.rb, dashboard\_access\_spec.rb):** Estas pruebas validan el núcleo del sistema de autenticación: el inicio y cierre de sesión, y la protección de rutas.

### A. Prueba: Inicio de Sesión con Credenciales Válidas.

#### a. Pasos Realizados:

- i. Se crea un usuario de prueba en la base de datos con un email y contraseña conocidos.
  - ii. El sistema de pruebas navega a la página de inicio de sesión (`/users/sign_in`).
  - iii. Se rellenan los campos "Email" y "Password" con las credenciales válidas del usuario.
  - iv. Se simula un clic en el botón "Log in".
- b. **Resultados Esperados:** El sistema debe autenticar al usuario correctamente y redirigirlo al panel principal, mostrando un mensaje de éxito que confirme el inicio de sesión ("Signed in successfully").
- c. **Resultados Obtenidos:** La aplicación mostró el mensaje "Signed in successfully". El resultado obtenido coincide con el esperado, validando la prueba como exitosa.

## **B. Prueba: Cierre de Sesión de un Usuario Autenticado.**

### **a. Pasos Realizados:**

- i. Se inicia sesión mediante programación con un usuario de prueba.
  - ii. El sistema navega a la página principal de la aplicación.
  - iii. Se simula un clic en el botón "Sign Out" para cerrar la sesión.
- b. **Resultados Esperados:** El sistema debe cerrar la sesión activa del usuario y mostrar un mensaje de confirmación ("Signed out successfully").
- c. **Resultados Obtenidos:** La aplicación mostró el mensaje "Signed out successfully". El resultado coincide con el esperado, confirmando que el ciclo de vida de la sesión se gestiona correctamente.

## **C. Prueba: Acceso No Autorizado a Rutas Protegidas.**

### **a. Pasos Realizados:**



- i. Sin haber iniciado sesión, se intenta navegar directamente a una ruta protegida (el *dashboard* en /dashboard).
- b. **Resultados Esperados:** El sistema debe denegar el acceso, redirigir al usuario anónimo a la página de inicio de sesión y mostrar un mensaje informativo indicando la necesidad de autenticarse ("You need to sign in or sign up before continuing.").
- c. **Resultados Obtenidos:** La aplicación redirigió correctamente a la página de inicio de sesión y mostró el mensaje esperado. La prueba se validó como exitosa.

**Bloqueo de Cuentas (account\_lockout\_spec.rb):** Esta prueba valida la contramedida contra ataques de fuerza bruta, una pieza clave de la seguridad de la autenticación.

#### A. Prueba: Bloqueo de Cuenta por Intentos de Sesión Fallidos.

##### a. Pasos Realizados:

- i. Se simulan 5 intentos de inicio de sesión consecutivos para un usuario de prueba utilizando una contraseña incorrecta.
  - ii. Tras el quinto intento, se verifica el mensaje mostrado en pantalla que indica el bloqueo de la cuenta implicada.
  - iii. A continuación, se realiza un sexto intento, esta vez con las credenciales correctas, para confirmar que la cuenta permanece bloqueada.
- b. **Resultados Esperados:** Después del quinto intento, el sistema debe mostrar el mensaje "Your account is locked.". En el intento posterior con las credenciales correctas, el acceso debe seguir denegado, persistiendo el mismo mensaje de bloqueo.
  - c. **Resultados Obtenidos:** La aplicación mostró el mensaje "Your account is locked." tras el quinto intento y lo mantuvo en el intento subsecuente. Los resultados son idénticos a los esperados, validando la prueba como exitosa.

**Recuperación Segura de Contraseñas (password\_recovery\_spec.rb):** Se prueba el flujo completo de "olvidé mi contraseña" para asegurar que es funcional y no introduce brechas de seguridad.

#### A. Prueba: Flujo de Recuperación de Contraseña.

##### a. Pasos Realizados:

- i. Se solicita un reseteo de contraseña para un email de prueba.
  - ii. Se verifica que se ha enviado un correo electrónico y se extrae el token de reseteo.
  - iii. Se visita el enlace de cambio de contraseña con dicho token.
  - iv. Se establece y confirma una nueva contraseña.
  - v. Finalmente, se intenta iniciar sesión con la nueva contraseña.
- b. **Resultados Esperados:** El flujo debe completarse exitosamente. Se espera que la contraseña se cambie con éxito y que el inicio de sesión con la nueva contraseña sea satisfactorio.
- c. **Resultados Obtenidos:** La aplicación mostró los mensajes de confirmación en cada etapa ("You will receive an email...", "Your password has been changed successfully.") y el inicio de sesión con la nueva contraseña funcionó correctamente. La prueba se validó como exitosa.

**Registro de Usuarios (user\_registration\_spec.rb):** Se verifica que el proceso de registro funciona como se espera, creando nuevos usuarios solo cuando los datos son válidos.

#### **A. Prueba: Registro de Usuario con Datos Válidos.**

##### **a. Pasos Realizados:**

- i. Se navega a la página de registro.
  - ii. Se rellenan los campos con datos válidos que cumplen las políticas de seguridad (email, contraseña y confirmación).
  - iii. Se envía el formulario de registro.
- b. **Resultados Esperados:** El sistema debe crear una nueva cuenta en la base de datos y mostrar un mensaje de bienvenida ("Welcome! You have signed up successfully").
- c. **Resultados Obtenidos:** La aplicación mostró el mensaje esperado y el conteo de usuarios en la base de datos aumentó en uno. La prueba fue exitosa.

#### **B. Prueba: Intento de Registro con Datos Inválidos.**

##### **a. Pasos Realizados:**

- i. Se navega a la página de registro.
- ii. Se rellenan los campos con datos inválidos (contraseña corta, email mal formado, etc.).
- iii. Se envía el formulario.

- b. **Resultados Esperados:** El sistema debe rechazar la creación del usuario, no debe añadir ninguna entrada a la base de datos y debe mostrar los mensajes de error correspondientes (p. ej., "Password is too short...").
- c. **Resultados Obtenidos:** La aplicación mostró los mensajes de error adecuados y el conteo de usuarios permaneció inalterado, coincidiendo con el resultado esperado. La prueba fue exitosa.

## 2. Control de Autorización Robusto

Una vez que un usuario está autenticado, es crucial verificar que solo pueda acceder a los recursos y ejecutar las acciones para las que tiene permiso explícito. Las siguientes pruebas validan la correcta implementación del sistema de autorización, gestionado por la gema Pundit, tanto a nivel de la interfaz de usuario como en la lógica del controlador, previniendo así la escalada de privilegios.

**Autorización a nivel de UI y Lógica (post\_authorization\_spec.rb):** Estas pruebas confirman que las políticas de autorización se aplican correctamente según el rol y la propiedad del recurso, abarcando desde usuarios anónimos hasta los propietarios de los contenidos.

### A. Prueba: Denegación de Acceso a Usuarios Anónimos.

- a. **Pasos Realizados:**
  - i. Como usuario no autenticado (invitado), se intenta acceder a la URL para crear una nueva publicación (/posts/new).
- b. **Resultados Esperados:** El sistema debe impedir el acceso a la funcionalidad de creación. Se espera que el usuario sea redirigido a la página de inicio de sesión y se le notifique con el mensaje "You need to sign in or sign up before continuing."
- c. **Resultados Obtenidos:** La aplicación redirigió al invitado a la página de inicio de sesión y mostró el mensaje esperado. La prueba fue exitosa, confirmando que las acciones que modifican datos están protegidas.

### B. Prueba: Denegación de Acceso para Editar Recursos Ajenos.

- a. **Pasos Realizados:**
  - i. Se crean dos usuarios, user\_one y user\_two.
  - ii. user\_one crea una publicación.

- iii. user\_two inicia sesión e intenta acceder a la URL para editar la publicación de user\_one.
- b. **Resultados Esperados:** La política de autorización debe denegar la acción. Se espera que el sistema redirija a user\_two a la página de inicio y muestre un mensaje de error claro: "You are not authorized to perform this action."
- c. **Resultados Obtenidos:** La aplicación redirigió al usuario a la página de inicio y mostró el mensaje de autorización denegada, coincidiendo exactamente con el resultado esperado. El test se validó como exitoso.

#### C. Prueba: Permiso de Acceso para Editar Recursos Propios.

- a. **Pasos Realizados:**
  - i. El usuario user\_one inicia sesión.
  - ii. Navega a la página de edición de su propia publicación.
  - iii. Modifica el título y envía el formulario de actualización.
- b. **Resultados Esperados:** Al ser el propietario del recurso, el usuario debe poder acceder al formulario de edición y actualizar su publicación. Se espera ver un mensaje de éxito ("Post was successfully updated.").
- c. **Resultados Obtenidos:** El sistema permitió la edición y, tras la actualización, mostró el mensaje de éxito esperado. La prueba fue exitosa, validando que los permisos para propietarios funcionan correctamente.

#### Protección contra Referencias Directas Inseguras a Objetos (IDOR) (idor\_protection\_spec.rb):

Esta prueba es crítica para la seguridad, ya que simula un ataque directo en el que un usuario malintencionado manipula la URL para acceder a datos que no le pertenecen. Valida que la autorización se aplica a nivel de controlador, independientemente de la interfaz.

#### A. Prueba: Intento de Edición de un Post Ajeno Manipulando la URL.

- a. **Pasos Realizados:**
  - i. Se configuran dos usuarios en la base de datos: user\_owner (el propietario) y user\_attacker (el atacante).
  - ii. Se crea una publicación (post\_to\_attack) que pertenece a user\_owner.
  - iii. El usuario user\_attacker inicia sesión en la aplicación.
  - iv. El atacante no interactúa con la UI, sino que navega directamente a la URL de edición del recurso ajeno (ej: /posts/1/edit, donde el post con ID 1 pertenece a user\_owner).

- b. **Resultados Esperados:** La capa de autorización del controlador (Pundit) debe interceptar la petición antes de que se renderice la vista. Se espera que el sistema redirija inmediatamente al atacante a la página de inicio (/) y le presente el mensaje de error "You are not authorized to perform this action.". La página de edición del post nunca debe ser mostrada.
- c. **Resultados Obtenidos:** La aplicación se comportó exactamente como se esperaba. El atacante fue redirigido a la página de inicio, se mostró el mensaje de acceso denegado y el contenido del formulario de edición no fue visible en ningún momento. El resultado obtenido es idéntico al esperado, lo que valida el éxito de esta prueba crucial de seguridad.

### 3. Defensas contra Ataques de Inyección

Esta sección valida la capacidad del *framework* para neutralizar y rechazar entradas maliciosas y peticiones fraudulentas, que constituyen dos de los vectores de ataque más comunes en aplicaciones web. Las pruebas se centran en la prevención de Cross-Site Scripting (XSS) y Cross-Site Request Forgery (CSRF).

**Sanitización de Entradas contra XSS (post\_sanitization\_spec.rb):** Esta prueba confirma que el *framework* previene ataques de Cross-Site Scripting (XSS) de tipo persistente. Se verifica que toda entrada de usuario que contenga HTML es estrictamente sanitizada antes de ser almacenada y renderizada, eliminando cualquier código potencialmente malicioso.

#### A. Prueba: Creación de un Post con Contenido HTML y un Script Malicioso.

##### a. Pasos Realizados:

- i. Un usuario autenticado navega al formulario para crear una nueva publicación.
- ii. En el campo de contenido, se introduce un *payload* que mezcla etiquetas HTML legítimas (<h2>, <p>) con una etiqueta de script maliciosa (<script>alert('XSS');</script>).
- iii. El usuario envía el formulario para crear la publicación.
- iv. El sistema de pruebas inspecciona tanto la página donde se muestra el post recién creado como el registro correspondiente en la base de datos.

- b. **Resultados Esperados:** La configuración de sanitización debe eliminar por completo todas las etiquetas HTML del contenido. En la página renderizada, no debe aparecer ningún elemento con formato (h2, p) y, crucialmente, la etiqueta `<script>` debe ser eliminada, neutralizando el ataque. El contenido en la base de datos también debe almacenarse como texto plano, sin ninguna etiqueta.
- c. **Resultados Obtenidos:** Al visualizar la publicación, todo el código HTML, incluido el script, fue eliminado, mostrando únicamente el texto plano ("Sub-encabezado Contenido limpio."). La inspección de la base de datos confirmó que el contenido se guardó sin ninguna etiqueta. Los resultados son idénticos a los esperados, validando el éxito de la prueba de sanitización.

**Protección Anti-CSRF (`csrf_protection_spec.rb`):** Esta prueba de tipo *request* valida que el mecanismo de protección `protect_from_forgery` nativo de Rails está activo y funcionando correctamente. Su objetivo es asegurar que solo las peticiones que se originan en la propia aplicación puedan modificar datos.

#### **A. Prueba: Simulación de una Petición POST sin Token de Autenticidad.**

- a. **Pasos Realizados:**
  - i. Se habilita explícitamente la protección CSRF en el entorno de pruebas.
  - ii. Se autentica un usuario mediante programación.
  - iii. Se envía una petición **POST** directamente al controlador para crear una nueva publicación, pero se omite deliberadamente el token de autenticidad CSRF que Rails normalmente incluiría en un formulario legítimo.
- b. **Resultados Esperados:** El sistema debe detectar la ausencia del token CSRF y rechazar la petición de inmediato. Se espera que la aplicación responda con un código de estado HTTP **422 Unprocessable Entity**, indicando que la petición es inválida. Como consecuencia, no se debe crear ninguna nueva publicación en la base de datos.
- c. **Resultados Obtenidos:** La aplicación respondió con el código de estado **422 Unprocessable Entity**, tal como se esperaba. La petición fue rechazada y no se creó ningún registro nuevo. El resultado obtenido coincide plenamente con

el esperado, confirmando que la defensa contra ataques CSRF está correctamente implementada y activa.

#### 4. Hardening de la Configuración y el Entorno

La seguridad de una aplicación no reside únicamente en su código, sino también en una configuración robusta y en la seguridad de su entorno y dependencias. Las siguientes pruebas validan que el framework aplica una estrategia de "defensa en profundidad", reforzando la aplicación a través de cabeceras HTTP, políticas de contenido, auditoría de la cadena de suministro y una gestión segura de los secretos.

**Cabeceras de Seguridad (`security_headers_spec.rb`):** Esta prueba verifica que la aplicación envía en sus respuestas HTTP un conjunto de cabeceras de seguridad clave, instruyendo al navegador para que active protecciones del lado del cliente.

##### A. Prueba: Verificación de Cabeceras HTTP de Seguridad.

###### a. Pasos Realizados:

- i. Se realiza una petición a la página de inicio de la aplicación (/).
- ii. Una vez cargada la página, se capturan y analizan las cabeceras de la respuesta HTTP.

b. **Resultados Esperados:** La respuesta debe incluir cabeceras específicas con valores seguros. Se espera que X-Frame-Options sea SAMEORIGIN (protección contra clickjacking), que X-Content-Type-Options sea nosniff (previene el *MIME sniffing*) y que X-Xss-Protection esté activado, entre otras.

c. **Resultados Obtenidos:** El análisis de la respuesta confirmó la presencia de todas las cabeceras de seguridad esperadas con sus valores correctos. Los resultados coinciden con los esperados, validando la prueba como exitosa.

**Política de Seguridad de Contenido (CSP) (`content_security_policy_spec.rb`):** Esta prueba valida que se está enviando una Política de Seguridad de Contenido (CSP) estricta, una defensa fundamental para mitigar ataques de Cross-Site Scripting (XSS).

##### A. Prueba: Envío de una Cabecera CSP Restrictiva.

###### a. Pasos Realizados:

- i. Se navega a una página que, para fines de la prueba, renderiza contenido con una etiqueta `<script>` potencialmente maliciosa.

- ii. Se inspecciona la cabecera Content-Security-Policy de la respuesta HTTP.
- b. **Resultados Esperados:** La cabecera CSP debe estar presente y no debe permitir la ejecución de scripts o estilos en línea de forma insegura ('unsafe-inline'). Debe contener una lista blanca de orígenes permitidos y hashes criptográficos para los scripts legítimos que la aplicación sí necesita ejecutar.
- c. **Resultados Obtenidos:** La cabecera CSP se envió correctamente y su contenido coincidió con la política restrictiva esperada, bloqueando la ejecución de scripts no autorizados. La prueba se validó como exitosa.

**Auditoría de Dependencias (dependency\_audit\_spec.rb):** Esta prueba automatiza la seguridad de la cadena de suministro, verificando que la aplicación no utiliza ninguna librería (gema) con vulnerabilidades de seguridad conocidas.

#### **A. Prueba: Escaneo de Gemas en Busca de Vulnerabilidades.**

- a. **Pasos Realizados:**
  - i. Se simula la ejecución de la herramienta de auditoría de dependencias (bundle audit).
  - ii. Se prueba en dos escenarios: uno donde la herramienta no encuentra vulnerabilidades y otro donde se simula la detección de una gema vulnerable.
- b. **Resultados Esperados:** En el primer escenario, la página de auditoría debe mostrar un mensaje de éxito ("Congratulations!"). En el segundo, debe mostrar una advertencia clara ("Warning!") y una tabla con los detalles de la gema vulnerable.
- c. **Resultados Obtenidos:** La aplicación gestionó ambos escenarios correctamente, mostrando el mensaje de éxito en el primer caso y la tabla detallada de la vulnerabilidad simulada en el segundo. Los resultados obtenidos coinciden con los esperados, validando la prueba.

**Gestión de Secretos (secure\_credentials\_spec.rb):** Esta prueba verifica que las credenciales sensibles (como claves de API) se cargan de forma segura desde el archivo cifrado de Rails y que la aplicación maneja correctamente tanto su presencia como su ausencia.



## A. Prueba: Carga y Visualización de Secretos.

### a. Pasos Realizados:

- i. Un usuario administrador inicia sesión y navega al dashboard.
- ii. Se simulan dos casos: uno en el que la clave de API se carga correctamente y otro en el que la clave está ausente.

b. **Resultados Esperados:** Cuando la clave está presente, la página debe mostrar un indicador de éxito ("SUCCESS") y revelar la clave de prueba al usuario. Cuando la clave está ausente, debe mostrar un indicador de error ("ERROR") y un mensaje informativo, sin exponer detalles sensibles.

c. **Resultados Obtenidos:** La aplicación mostró los indicadores y mensajes correctos para ambos escenarios, demostrando que la gestión de secretos es robusta y resiliente a fallos de configuración. La prueba fue exitosa.

## 5. Logging y Monitorización

Una estrategia de seguridad completa no solo previene ataques, sino que también proporciona la visibilidad necesaria para detectar, investigar y responder a posibles incidentes. Las siguientes pruebas validan que el framework implementa un sistema de logging de seguridad detallado y que el acceso a dichos registros está correctamente protegido.

**Registro de Eventos de Seguridad (security\_logging\_spec.rb):** Esta suite de pruebas asegura que los eventos de seguridad más relevantes, como los intentos de inicio de sesión fallidos y los errores de autorización, se registran de forma fiable en un archivo de log dedicado.

## A. Prueba: Registro de un Intento de Inicio de Sesión Fallido.

### a. Pasos Realizados:

- i. Se navega a la página de inicio de sesión.
- ii. Se introduce el email de un usuario válido, pero con una contraseña incorrecta.
- iii. Se envía el formulario de inicio de sesión.
- iv. Se inspecciona el contenido del archivo log/security.log.

b. **Resultados Esperados:** El archivo de log de seguridad debe contener una nueva entrada en formato JSON que identifique el evento como "event":"failed\_login\_attempt".

- c. **Resultados Obtenidos:** El archivo de log contenía la entrada esperada, confirmando que los intentos de acceso fallidos se están registrando correctamente para su posterior análisis. La prueba fue exitosa.

## **B. Prueba: Registro de un Fallo de Autorización (Pundit).**

- a. **Pasos Realizados:**
  - i. Un usuario autenticado intenta acceder a la página de edición de una publicación que no le pertenece.
  - ii. Esta acción es denegada por la política de Pundit.
  - iii. Se inspecciona el contenido del archivo log/security.log.
- b. **Resultados Esperados:** La aplicación debe registrar este intento de acceso no autorizado. Se espera encontrar una entrada en el log que identifique el evento como "event": "authorization\_failure".
- c. **Resultados Obtenidos:** El log registró el evento de fallo de autorización como se esperaba. El resultado coincide con el esperado, validando la prueba.

## **C. Prueba: Registro de Múltiples Intentos Fallidos que Llevan a un Bloqueo.**

- a. **Pasos Realizados:**
  - i. Se simulan 5 intentos de inicio de sesión fallidos para la misma cuenta.
  - ii. Se verifica el mensaje en la interfaz de usuario.
  - iii. Se cuenta el número de eventos de inicio de sesión fallido en el archivo log/security.log.
- b. **Resultados Esperados:** La interfaz debe mostrar el mensaje "Your account is locked.". El archivo de log debe contener exactamente 5 entradas correspondientes a los 5 intentos fallidos.
- c. **Resultados Obtenidos:** Tanto el mensaje en la interfaz como el número de entradas en el log coincidieron con los resultados esperados, demostrando que el sistema registra la actividad que precede a un bloqueo de cuenta. La prueba fue exitosa.

## **D. Prueba: Control de Acceso al Visor de Logs.**

- a. **Pasos Realizados:**
  - i. Se prueba el acceso al visor de logs con dos perfiles: un administrador y un usuario regular.
  - ii. Se verifica si el enlace "View Security Logs" es visible para cada uno.
  - iii. Se intenta acceder a la URL del visor de logs directamente con ambos usuarios.

- b. **Resultados Esperados:** El enlace al visor de logs solo debe ser visible para el administrador. Un usuario regular que intente acceder directamente a la URL debe ser redirigido y ver un mensaje de "Acceso denegado".
- c. **Resultados Obtenidos:** El comportamiento de la aplicación fue exactamente el esperado. El administrador pudo acceder a los logs, mientras que al usuario regular se le denegó el acceso de forma segura. La prueba fue un éxito.

Este conjunto de pruebas funcionales proporciona una validación exhaustiva y multicapa. No solo confirman que la aplicación está libre de errores funcionales, sino que demuestran activamente que las defensas contra una amplia gama de ataques web de primer nivel están correctamente implementadas, activas y son efectivas.

### 7.3. Análisis y discusión de los Resultados

Esta sección tiene como propósito fundamental interpretar y discutir en profundidad los resultados empíricos obtenidos en las pruebas de seguridad detalladas en la sección 7.2. El objetivo no es meramente reiterar los datos, sino contextualizarlos para demostrar cómo validan la hipótesis central de este Trabajo Fin de Máster. Se argumentará que la ausencia total de vulnerabilidades y errores funcionales en la aplicación **demo\_app** no es un hecho aislado, sino la consecuencia directa de la arquitectura y filosofía de **secure\_framework**. A través de este análisis, se evidenciará que el *framework* constituye una herramienta eficaz que permite la construcción de aplicaciones web robustas y seguras, mitigando de raíz un amplio espectro de debilidades comunes mediante un enfoque de "seguridad por defecto".

#### 7.3.1. Discusión de los Resultados del Análisis Estático (SAST)

El Análisis Estático de Código (SAST) representa la primera línea de defensa en un ciclo de vida de desarrollo seguro. Los resultados obtenidos con Brakeman y Rubocop son excepcionales y merecen un análisis detallado, ya que confirman la solidez del código fuente generado y configurado bajo la tutela de **secure\_framework**.

- **Análisis de Brakeman: La Validación de la Seguridad Intrínseca**

El informe de Brakeman, arrojando un resultado de "**Security Warnings: 0**", es la validación más clara de la eficacia del *framework* a nivel de código. Este resultado dista de ser trivial; en una aplicación Rails estándar sin un hardening deliberado, es

común que Brakeman detecte advertencias de severidad variable. La ausencia de estas advertencias es una consecuencia directa de las decisiones de diseño arquitectónico de **secure\_framework**, que se alinean con los requisitos funcionales (FR) y no funcionales (NFR) definidos en el capítulo 5.

Concretamente, este resultado se explica por:

- **Configuración Segura por Defecto:** El *framework* asegura que mecanismos como la protección contra CSRF (FR-6) y el uso de ORM para prevenir inyecciones SQL (FR-4) no solo estén presentes, sino correctamente configurados desde el inicio. Brakeman no encuentra fallos en estas áreas porque el *framework* los ha eliminado proactivamente.
- **Ausencia de Patrones Inseguros:** El framework no introduce ni fomenta el uso de patrones de código que Brakeman busca activamente, como redirecciones inseguras o el uso de funciones peligrosas para la renderización.
- **Robustez frente a Vulnerabilidades Comunes (NFR-1):** El resultado confirma el cumplimiento del requisito NFR-1, demostrando que las defensas integradas contra el Top 10 de OWASP, como XSS (FR-5) y el Control de Acceso Deficiente (mitigado por Pundit, FR-3), son estructuralmente sólidas.
- **Análisis de Rubocop: La Seguridad como Consecuencia de la Calidad**

El resultado de "**no offenses detected**" por parte de Rubocop, aunque a primera vista se asocia con el estilo y la calidad del código, tiene profundas implicaciones para la seguridad. Un código que se adhiere a estándares estrictos, tal y como se ha demostrado, es:

- **Más Legible y Mantenable:** Esto reduce drásticamente la probabilidad de que los desarrolladores introduzcan errores lógicos durante el mantenimiento o la evolución del software, errores que a menudo se traducen en vulnerabilidades de seguridad (ej. lógica de negocio defectuosa).
- **Más Fácil de Auditar:** Un código consistente y limpio facilita las revisiones de seguridad por pares y las auditorías automáticas, permitiendo identificar posibles problemas de forma más eficiente.

En esencia, el resultado de Rubocop demuestra cómo se ha establecido un ecosistema de desarrollo donde la calidad del código, un pilar de la seguridad a largo plazo, es una prioridad.

### 7.3.2. Discusión de los Resultados de las Pruebas Funcionales (RSpec)

Si el análisis SAST valida la "teoría" (el código en reposo), las pruebas funcionales con RSpec validan la "práctica" (el comportamiento de la aplicación en ejecución). El resultado impecable de **"29 ejemplos, 0 fallos"** en una suite de pruebas que simula escenarios de ataque y uso legítimo, constituye la validación holística del framework. Cada grupo de pruebas confirma que una capa específica de defensa no solo existe, sino que es funcionalmente efectiva.

- **Gestión de Identidad y Acceso:** Las pruebas **user\_session\_spec.rb** y **account\_lockout\_spec.rb** van mucho más allá de confirmar que Devise está instalado. Confirman que las **políticas de seguridad específicas** configuradas por **secure\_framework** son efectivas dinámicamente. El éxito de estas pruebas demuestra que la longitud mínima de contraseña, los timeouts de sesión y, crucialmente, el mecanismo de bloqueo de cuentas tras 5 intentos fallidos (`config.maximum_attempts = 5`) funcionan como un sistema cohesivo para mitigar ataques de fuerza bruta y proteger las credenciales de los usuarios.
- **Control de Autorización Robusto (Pundit):** Este es, posiblemente, el pilar más crítico de la validación. Las pruebas **post\_authorization\_spec.rb** e **idor\_protection\_spec.rb** son de una importancia capital. El resultado positivo no solo indica que un usuario no puede ver un botón de "editar" que no le corresponde (UI). Lo que demuestra es que la capa de autorización implementada por Pundit, y orquestada por el framework, previene de manera efectiva la **escalada de privilegios** y los ataques de **Referencia Directa Insegura a Objetos (IDOR)** a nivel de controlador. La prueba IDOR, al simular cómo un User A intenta acceder a un recurso de User B y recibe un error de autorización, confirma que una de las vulnerabilidades más comunes y peligrosas en aplicaciones web ha sido neutralizada por diseño.
- **Defensas contra Ataques de Inyección:** Las pruebas **post\_sanitization\_spec.rb** (XSS) y **csrf\_protection\_spec.rb** validan empíricamente que las defensas contra estos vectores de ataque son efectivas en tiempo de ejecución. La prueba XSS confirma que el payload malicioso es neutralizado y su etiqueta **<script>** eliminada del HTML renderizado, protegiendo a otros usuarios. La prueba CSRF valida que cualquier petición POST sin el token de autenticidad es rechazada, protegiendo al usuario de realizar acciones en su nombre sin su consentimiento.

- **Hardening de la Configuración y el Entorno:** Finalmente, las pruebas de **security\_headers\_spec.rb**, **content\_security\_policy\_spec.rb** y **dependency\_audit\_spec.rb** confirman la robustez de la estrategia de "**defensa en profundidad**". No basta con tener un código seguro; el entorno debe estar reforzado. Estas pruebas verifican que la aplicación envía cabeceras HTTP que instruyen al navegador a aplicar políticas de seguridad adicionales (X-Frame-Options, CSP, etc.) y que la cadena de suministro de software es segura (ninguna gema comprometida), añadiendo múltiples capas de protección que funcionan en conjunto.
- **Logging y Monitorización:** Finalmente, la prueba **security\_logging\_spec.rb** cierra el círculo de la validación. Su éxito demuestra que el sistema no solo se defiende, sino que también registra los intentos de ataque. Al confirmar que los inicios de sesión fallidos y los errores de autorización se escriben en un log seguro (cuyo acceso está, a su vez, protegido por autorización), se valida que el framework proporciona las herramientas necesarias para la detección de intrusiones y el análisis forense, un componente indispensable de una estrategia de seguridad madura.

### 7.3.3. Síntesis y Validación Holística del Framework

La convergencia de los resultados de las pruebas estáticas y dinámicas ofrece una validación multicapa y exhaustiva de la eficacia y robustez de **secure\_framework**. El análisis estático confirma que la base de código está construida sobre cimientos sólidos, libres de patrones de inseguridad conocidos. El análisis dinámico, a través de una suite de 29 pruebas funcionales de seguridad, demuestra de manera concluyente que estas defensas teóricas se traducen en protecciones efectivas y activas en un entorno de ejecución real.

La combinación de estos resultados exitosos no es una coincidencia, sino la prueba fehaciente de que el framework cumple con su objetivo primordial: abstraer la inherente complejidad de la seguridad en el desarrollo web. Al proporcionar configuraciones seguras por defecto, integrar herramientas de seguridad clave de manera coherente y promover buenas prácticas de codificación, **secure\_framework** consigue eliminar de raíz clases enteras de vulnerabilidades. Se ha demostrado que una aplicación (**demo\_app**) construida sobre este framework no solo es funcional, sino intrínsecamente segura, validando así la hipótesis y los objetivos planteados al inicio de este trabajo.

## 8. Conclusiones y líneas futuras

El presente Trabajo Fin de Máster abordó el desafío fundamental de integrar la seguridad en el ciclo de vida del desarrollo de software, un problema acentuado por la creciente complejidad de las amenazas y la presión por un desarrollo ágil. El objetivo general, "generar un framework seguro basado en Ruby on Rails que facilite la construcción de aplicaciones web seguras", se ha cumplido exitosamente mediante la creación, implementación y validación de **secure\_framework**. Este proyecto ha logrado materializar una solución tangible que responde directamente a la brecha existente entre las buenas prácticas de seguridad teóricas y su aplicación efectiva en proyectos reales, tal y como se planteó en la introducción.

La contribución principal de este trabajo no es meramente la agregación de herramientas de seguridad, sino el diseño de un ecosistema coherente y centrado en el desarrollador. La elección de desarrollar **secure\_framework** como una gema de Ruby ha sido una decisión arquitectónica clave, proporcionando una solución modular, versionable y fácilmente distribuable que ataca directamente el "factor humano" y la "complejidad del ecosistema". Se ha abstraído la configuración de componentes de seguridad críticos, como la autenticación con Devise, la autorización con Pundit o la configuración de cabeceras seguras, permitiendo a los desarrolladores adoptar una postura de seguridad robusta por defecto, sin necesidad de ser expertos en ciberseguridad. Los objetivos específicos se han alcanzado sistemáticamente: se ha realizado un estudio exhaustivo del arte que ha informado una arquitectura modular, se ha construido el framework integrando las mejores prácticas y, finalmente, se ha validado su eficacia de manera rigurosa.

La validación empírica a través de la aplicación **demo\_app** ha arrojado resultados concluyentes que confirman la hipótesis central del proyecto. La obtención de "cero advertencias de seguridad" de Brakeman, "cero ofensas de estilo" de Rubocop y, de manera crucial, "cero fallos" en la exhaustiva suite de 29 pruebas funcionales de seguridad con Rspec, demuestra que el enfoque de "seguridad por defecto" es viable y efectivo. La aplicación **demo\_app** no es segura por un esfuerzo de *hardening* manual posterior, sino porque está construida sobre cimientos que son inherentemente seguros, validando que el framework transforma las defensas teóricas en protecciones activas y funcionales en un entorno de ejecución real.

Aunque se han alcanzado los objetivos propuestos, este trabajo abre la puerta a diversas líneas futuras que podrían ampliar significativamente su impacto y utilidad. En primer lugar, la **expansión del catálogo de componentes de seguridad** es un paso lógico. Se podría integrar soporte para mecanismos de autenticación de dos factores (2FA) más avanzados, como TOTP o WebAuthn, y desarrollar componentes específicos para la autenticación de APIs mediante tokens JWT, incluyendo la gestión segura de su ciclo de vida. También se podría **mejorar la flexibilidad del generador de instalación**. Una futura versión podría incluir un modo interactivo que permita a los desarrolladores seleccionar qué componentes desean instalar y configurar, ofreciendo diferentes perfiles de seguridad (p. ej., "básico", "avanzado", "API-only") para adaptarse mejor a las necesidades específicas de cada proyecto. Para asegurar su viabilidad en entornos de producción a gran escala, sería valioso realizar un estudio de impacto en el rendimiento (*overhead*) que el framework introduce, optimizando sus componentes si fuera necesario. Finalmente, para maximizar su alcance, el siguiente paso lógico sería **publicar la gema en el repositorio oficial de RubyGems.org**, fomentando su adopción y la creación de una comunidad de usuarios que pueda contribuir a su evolución y mantenimiento.

En definitiva, **secure\_framework** no solo representa la culminación exitosa de los objetivos de este trabajo, sino que sienta las bases para una línea de investigación y desarrollo con el potencial de impactar positivamente en la forma en que se construyen aplicaciones Ruby on Rails, promoviendo un ecosistema de desarrollo web más robusto y seguro.



## Referencias bibliográficas

- [1] Parlamento Europeo y Consejo de la Unión Europea, “Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016, relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos,” *Diario Oficial de la Unión Europea*, L119, pp. 1–88, May 2016. [Online]. Available: <https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX%3A32016R0679>
- [2] A. L. Mesquida, A. Mas, E. Amengual, and I. Cabestrero, “Sistema de Gestión Integrado según las normas ISO 9001, ISO/IEC 20000 e ISO/IEC 27001,” *REICIS. Revista Española de Innovación, Calidad e Ingeniería del Software*, vol. 6, no. 3, pp. 25–34, 2010.
- [3] OWASP Foundation, “OWASP Top 10:2021,” OWASP, 2021. [Online]. Available: <https://owasp.org/Top10/es/>
- [4] Y. Valdés-Rodríguez, J. Hochstetter-Diez, J. Díaz-Arancibia, and R. Cadena-Martínez, “Towards the integration of security practices in agile software development: A systematic mapping review,” *Applied Sciences*, vol. 13, no. 7, Art. no. 4578, 2023. [Online]. Available: <https://doi.org/10.3390/app13074578>
- [5] Ruby on Rails Core Team, “Ruby on Rails security guide,” *Ruby on Rails Guides*, 2023. [Online]. Available: <https://guides.rubyonrails.org/security.html>
- [6] J. Miller, “Patterns in practice: Convention over configuration,” *MSDN Magazine*, Feb. 2009. [Online]. Available: <https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-in-practice-convention-over-configuration>
- [7] OWASP Foundation, “Mass Assignment Cheat Sheet,” OWASP Cheat Sheet Series, 2020. [Online]. Available: [https://cheatsheetseries.owasp.org/cheatsheets/Mass\\_Assignment\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html)
- [8] J. Allen, “GitHub.com compromised via public key vulnerability,” *InfoQ*, Mar. 12, 2012. [Online]. Available: <https://www.infoq.com/news/2012/03/GitHub-Compromised/>

- [9] Cloud Tuned, “Exploring Brakeman: Enhancing Ruby on Rails security,” Jun. 25, 2024. [Online]. Available: <https://cloudtuned.hashnode.dev/exploring-brakeman-enhancing-ruby-on-rails-security>
- [10] Brakeman Project, “Brakeman: Ruby on Rails Static Analysis Security Tool,” 2025. [Online]. Available: <https://brakemanscanner.org/>
- [11] RuboCop Developers, “RuboCop configuration,” *RuboCop Documentation*, 2023. [Online]. Available: <https://docs.rubocop.org/rubocop/configuration.html>
- [12] Heartcombo, “Devise,” GitHub repository, 2024. [Online]. Available: <https://github.com/heartcombo/devise>
- [13] CanCanCommunity, “CanCanCan,” GitHub repository, 2023. [Online]. Available: <https://github.com/CanCanCommunity/cancancanGitHub+5GitHub+5GitHub+5>
- [14] Varvet, “Pundit,” GitHub repository, 2025. [Online]. Available: <https://github.com/varvet/pundit>