Vol.8 No.1 (2024): Journal Scientific MQRInvestigar ISSN: 2588–0659
https://doi.org/10.56048/MQR20225.8.1.2024.236-259

Incidence of software design patterns on web application security

Incidencia de los patrones de diseño de software en la seguridad de aplicaciones web

Autores:

Ing. Mesías-Valencia, Juan José
Pontificia Universidad Católica del Ecuador Sede Ambato | PUCESA
Maestrante de Ciberseguridad
Ambato - Ecuador
jjmesias@pucesa.edu.ec
https://orcid.org/0009-0003-7145-2338

Mg. Cevallos-Muñoz, Fausto Danilo
Pontificia Universidad Católica del Ecuador Sede Ambato | PUCESA
Tutor Externo del Centro de Posgrados
Ambato - Ecuador
fcevallos@fagadevs.com
https://orcid.org/0009-0009-1627-4827

Fechas de recepción: 01-DIC-2023 aceptación: 08-ENE-2024 publicación: 15-MAR-2024

https://orcid.org/0000-0002-8695-5005
http://mqrinvestigar.com/

Resumen

Los patrones de diseño son soluciones recurrentes para problemas de diseño comunes, estas han ganado reconocimiento como herramientas fundamentales para estructurar y organizar el código de manera eficiente. En este contexto, surge la pregunta de cómo estos patrones pueden influir en la seguridad de las aplicaciones web, que a menudo están expuestas a una amplia gama de amenazas y vulnerabilidades.

Investigar la influencia de los patrones de diseño de software es importante ya que, al brindarnos enfoques para la validación de datos, la autenticación y la segregación de responsabilidades puede ayudarnos a identificar y prevenir vulnerabilidades más comunes y específicas, por lo tanto, disminuir la posibilidad de un ataque en la aplicación.

Las aplicaciones web al brindar diferentes servicios a usuarios manejan información sensible, por lo que su seguridad es fundamental para el usuario final. El objetivo de la investigación es determinar cómo los patrones de diseño de software contribuyen a mitigar vulnerabilidades en aplicaciones web.

Controlar y mitigar las vulnerabilidades es tarea diaria de los desarrolladores por lo que causan costos a la mantenibilidad de software, un aspecto importante de la investigación es denotar que al momento de desarrollar aplicaciones basada en patrones de diseño se puede solventar futuras incidencias de seguridad gracias a las estructuras y pautas bien definidas que guían el desarrollo por patrones de diseño.

Palabras clave: Ciberseguridad; Patrones de diseño; OWASP; Mitigación; Vulnerabilidad

Abstract

Design patterns are recurring solutions to common design problems; they have gained recognition as fundamental tools for efficiently structuring and organizing code. In this context, the question arises of how these patterns can influence the security of web applications, which are often exposed to a wide range of threats and vulnerabilities.

Investigating the influence of software design patterns is crucial since they provide approaches for data validation, authentication, and responsibility segregation. This can help identify and prevent common and specific vulnerabilities, thereby reducing the likelihood of an attack on the application. Web applications handle sensitive information as they provide various services to users, making their security crucial for end users. The research aims to determine how software design patterns contribute to mitigating vulnerabilities in web applications. Controlling and mitigating vulnerabilities is a daily task for developers and incurs costs in software maintainability. An essential aspect of the research is highlighting that, when developing applications based on design patterns, future security incidents can be addressed thanks to well-defined structures and guidelines that guide pattern-based development. Design patterns are recurring solutions to common design problems; they have gained recognition as fundamental tools for efficiently structuring and organizing code. In this context, the question arises of how these patterns can influence the security of web applications, which are often exposed to a wide range of threats and vulnerabilities. Investigating the influence of software design patterns is crucial since they provide approaches for data validation, authentication, and responsibility segregation. This can help identify and prevent common and specific vulnerabilities, thereby reducing the likelihood of an attack on the application. Web applications handle sensitive information as they provide various services to users, making their security crucial for end users. The research aims to determine how software design patterns contribute to mitigating vulnerabilities in web applications.

Controlling and mitigating vulnerabilities is a daily task for developers and incurs costs in software maintainability. An essential aspect of the research is highlighting that, when developing applications based on design patterns, future security incidents can be addressed thanks to well-defined structures and guidelines that guide pattern-based development.

Keywords: Cybersecurity; Design patterns; OWASP; Mitigation; Vulnerability

Introducción

La era contemporánea se halla caracterizada por una dependencia, ya sea parcial o total, de la tecnología, y en este panorama, el internet emerge como uno de los más significativos inventos. Este fenómeno ha revolucionado diversos aspectos de la vida humana, inicialmente como un medio global de comunicación, para posteriormente transformarse en la nueva Biblioteca de Alejandría. Así, el internet se erige como una entidad que alberga la totalidad de la historia y el conocimiento de la especie humana, redefiniendo la manera en que accedemos y compartimos información.

Internet se ha convertido en una herramienta versátil que abarca una amplia gama de actividades. Desde la búsqueda básica de información sobre cualquier tema hasta la interacción entre individuos a través de servicios de mensajería, la adquisición de productos mediante plataformas de comercio electrónico, y la realización de transacciones bancarias en cuestión de minutos. Su utilidad abarca prácticamente todos los aspectos de la vida cotidiana, consolidándose como una red integral que facilita y agiliza diversas tareas y operaciones.

Cuando los ingenieros se embarcan en el desarrollo de servicios web, su enfoque a menudo se centra en la búsqueda de nuevas tecnologías, lenguajes de programación avanzados y una arquitectura eficiente. Sin embargo, en muchos casos descuidan aspectos críticos como la estructura interna, la calidad del código y, sobre todo, la seguridad. Esta falta de atención a estos aspectos esenciales puede resultar en aplicaciones vulnerables a ataques o infiltraciones no deseadas.

Para identificar los riesgos y vulnerabilidades, se realiza una revisión concisa de los organismos internacionales encargados de destacar las amenazas más prevalentes en la web, recopilando así una lista actualizada de posibles peligros. Entre todas las organizaciones, OWASP se destaca como la más relevante. El Open Web Application Security Project, u OWASP, es una comunidad abierta dedicada a permitir que organizaciones e individuos diseñen, desarrollen, compren y mantengan aplicaciones en las que se pueda confiar. Sus herramientas, documentos, foros y capítulos son gratuitos y están abiertos a cualquier persona interesada en mejorar la seguridad de las aplicaciones. Desde sus inicios, OWASP ha producido una serie de recursos extremadamente valiosos para la seguridad de aplicaciones web [1].

Al recopilar información sobre riesgos y vulnerabilidades de OWASP, se emprende una búsqueda específica de patrones de diseño que puedan mitigar los problemas previamente identificados. Este proceso implica una exhaustiva exploración de diversos tipos y categorías de patrones centrados en la seguridad web, con el objetivo de determinar cuál es más adecuado para abordar cada riesgo particular.

Con base en los datos recopilados, se lleva a cabo la identificación del patrón de diseño que constituye la mejor solución para cada vulnerabilidad detectada. Utilizando esta información, se procede a la fase de diseño, codificación y prueba de prototipos, siguiendo

las mejores prácticas identificadas. Además, se emplean herramientas externas para evaluar la calidad y seguridad del código desarrollado. Finalmente, se presentan los resultados y conclusiones derivados de este trabajo.

Material y Métodos

La metodología empleada en este trabajo sigue el enfoque de desarrollo en cascada, caracterizado por un flujo secuencial de fases, donde cada una debe completarse antes de pasar a la siguiente. La metodología consiste en llevar a cabo una etapa inicial de análisis en el cual se identifican las vulnerabilidades más comunes reportadas por la OWASP y los patrones de diseño tradicionales disponibles e implementables. La siguiente fase comprende la etapa de diseño en donde el objetivo es identificar y seleccionar el patrón de diseño más apropiado para mitigar las vulnerabilidades seleccionadas. Posteriormente continúa la etapa de implementación en donde en un módulo de manejo de usuarios y acceso de los mismos programado en C# con NET CORE se implementan las vulnerabilidades identificadas y en una nueva versión de la misma aplicación en donde se aplican las correcciones estructurales dictadas por los patrones de diseño seleccionados. Finalmente la etapa de verificación consiste en evaluar el estado inicial de la aplicación con las herramientas NDepend y SonarQube y evaluar el estado final de la aplicación con las misma herramientas posterior a la corrección propuesta. Finalmente, se presentarán los resultados y conclusiones derivados de la investigación.

En base a la documentación revisada se pudo identificar aspectos esenciales como las vulnerabilidades abordadas por la OWASP además de los patrones tradicionales con los que se puede desarrollar la presente investigación.

OWASP Top 10. Es un documento de concientización estándar para desarrolladores y seguridad de aplicaciones web. Representa un amplio consenso sobre los riesgos de seguridad más críticos para las aplicaciones web [2]. Según el top 10 estos son los riesgos más comunes y que generan mayor preocupación en la comunidad.

A01:2021 - Broken Access Control. El control de acceso aplica una política tal que los usuarios no pueden actuar fuera de los permisos previstos. Las fallas generalmente conducen a la divulgación no autorizada de información, modificación o destrucción de todos los datos o a la realización de una función comercial fuera de los límites del usuario [3].

A02:2021 - Cryptographic Failures. Una falla criptográfica es una vulnerabilidad crítica de seguridad de una aplicación web que expone datos confidenciales de la aplicación en un algoritmo criptográfico débil o inexistente. Pueden ser contraseñas, registros médicos de pacientes, secretos comerciales, información de tarjetas de crédito, direcciones de correo electrónico u otra información personal del usuario [4].

A03:2021 - Injection. Cubre vulnerabilidades y fallas de seguridad como inyecciones SQL, NoSQL, comando OS, mapeo relacional de objetos (ORM), LDAP y lenguaje de

expresión (EL), inyección de biblioteca de navegación de gráficos de objetos (OGNL) y ataque XSS o Cross-Site Scripting [5].

A04:2021 - Insecure Design. El diseño inseguro es una categoría amplia que representa diferentes debilidades, expresadas como "diseño de control faltante o ineficaz". El diseño inseguro no es la fuente de todas las demás categorías de riesgo principales. Existe una diferencia entre diseño inseguro e implementación insegura. Diferenciamos entre fallas de diseño y defectos de implementación por una razón: tienen diferentes causas fundamentales y remediaciones. Un diseño seguro aún puede tener defectos de implementación que generan vulnerabilidades que pueden explotarse. Un diseño inseguro no puede solucionarse mediante una implementación perfecta ya que, por definición, nunca se crearon los controles de seguridad necesarios para defenderse contra ataques específicos. Uno de los factores que contribuye al diseño inseguro es la falta de un perfil de riesgo empresarial inherente al software o sistema que se está desarrollando y, por tanto, la imposibilidad de determinar qué nivel de diseño de seguridad se requiere [6].

A05:2021 - Security Misconfiguration. La aplicación puede ser vulnerable si carece de reforzamiento de seguridad adecuado en cualquier parte de su pila de aplicaciones o si los permisos en los servicios en la nube están configurados de manera incorrecta. Además, la habilitación o instalación de funciones innecesarias, el mantenimiento de cuentas predeterminadas con contraseñas sin cambios, y la revelación de mensajes de error detallados a los usuarios pueden exponer la aplicación a riesgos de seguridad. Es crucial asegurarse de que las funciones de seguridad más recientes estén habilitadas y configuradas de manera segura, que las configuraciones en los diversos componentes (servidores de aplicaciones, marcos de aplicaciones, bibliotecas, bases de datos, etc.) utilicen valores seguros y que se envíen encabezados y directivas de seguridad adecuadas. Mantener el software actualizado y seguir un proceso de configuración de seguridad repetible son prácticas esenciales para mitigar riesgos en los sistemas [7].

A06:2021 - Vulnerable and Outdated Components. La potencial vulnerabilidad se evidencia en diversas situaciones, como la falta de conocimiento sobre las versiones de todos los componentes utilizados (tanto en el lado del cliente como en el del servidor), incluyendo dependencias anidadas. Además, la exposición a riesgos se incrementa si el software empleado es vulnerable, incompatible o desactualizado, abarcando desde el sistema operativo hasta las bibliotecas. La falta de búsquedas regulares de vulnerabilidades y la ausencia de suscripción a boletines de seguridad relacionados también contribuyen a la vulnerabilidad. Igualmente, la no realización oportuna y basada en el riesgo de reparaciones y actualizaciones en la plataforma subyacente, marcos y dependencias, así como la falta de pruebas de compatibilidad para bibliotecas actualizadas, pueden exponer a la persona o la organización a periodos prolongados de riesgo innecesario. La protección insuficiente de las configuraciones de los componentes agrega otro nivel de vulnerabilidad [8].

A07:2021 - Identification and Authentication. La seguridad contra ataques vinculados a la autenticación se basa en la confirmación de identidad, autenticación y gestión

adecuada de sesiones de usuario. Las vulnerabilidades en la autenticación pueden surgir si la aplicación permite ataques automatizados, como el relleno de credenciales o la fuerza bruta, y si acepta contraseñas predeterminadas, débiles o comunes. Además, debilidades en los procesos de recuperación de credenciales y contraseñas olvidadas, el uso de almacenes de datos de contraseñas inseguros y la falta de autenticación multifactor contribuyen a la exposición. Otros riesgos incluyen la exposición del identificador de sesión en la URL, la utilización incorrecta de identificadores de sesión y la invalidez inadecuada de sesiones de usuario o tokens de autenticación durante el cierre de sesión o períodos de inactividad [9].

A08:2021 - Software and Data Integrity Failures. Las fallas de integridad del software y de los datos se relacionan con código e infraestructura que no protegen contra violaciones de integridad. Un ejemplo de esto es cuando una aplicación depende de complementos, bibliotecas o módulos de fuentes, repositorios y redes de entrega de contenido (CDN) que no son de confianza. Una canalización de CI/CD insegura puede generar la posibilidad de acceso no autorizado, código malicioso o compromiso del sistema. Por último, muchas aplicaciones ahora incluyen la funcionalidad de actualización automática, donde las actualizaciones se descargan sin suficiente verificación de integridad y se aplican a la aplicación previamente confiable. Los atacantes podrían potencialmente cargar sus propias actualizaciones para distribuirlas y ejecutarlas en todas las instalaciones. Otro ejemplo es cuando los objetos o datos se codifican o serializan en una estructura que un atacante puede ver y modificar y es vulnerable a una deserialización insegura [10].

A09:2021 - Security Logging and Monitoring Failures. El propósito de esta categoría es facilitar la detección, escalada y respuesta eficaz ante posibles infracciones activas. La detección de infracciones es imposible sin un adecuado sistema de registro y monitoreo. Cuando los procesos de registro, detección, monitoreo y respuesta activa son insuficientes en cualquier momento, se compromete la capacidad de la organización para abordar eficientemente las amenazas y eventos de seguridad [11].

A10:2021 - Server-Side Request Forgery (SSRF). Las fallas de SSRF ocurren cada vez que una aplicación web busca un recurso remoto sin validar la URL proporcionada por el usuario. Permite a un atacante obligar a la aplicación a enviar una solicitud diseñada a un destino inesperado, incluso cuando está protegida por un firewall, VPN u otro tipo de lista de control de acceso a la red (ACL). A medida que las aplicaciones web modernas brindan a los usuarios finales funciones convenientes, recuperar una URL se convierte en un escenario común. Como resultado, la incidencia de la SSRF está aumentando. Además, la gravedad de SSRF es cada vez mayor debido a los servicios en la nube y la complejidad de las arquitecturas.

Con el objetivo de mitigar las vulnerabilidades analizadas, los desarrolladores de software han instituido una serie de prácticas destinadas a prevenir problemas de seguridad en los sistemas mediante la creación y aplicación de patrones de diseño [12].

Patrones de diseño. En ingeniería de software, los patrones de diseño se presentan como esquemas o plantillas derivadas de prácticas bien estructuradas para abordar problemas comunes que suelen repetirse. Estas estructuras también pueden entenderse como las estrategias óptimas utilizadas para resolver eficazmente desafíos en el diseño de software. Adoptando un enfoque sistemático, los patrones de diseño ofrecen soluciones reutilizables y flexibles, proporcionando orientación sobre cómo resolver problemas específicos. Cada patrón de diseño puede incluir uno o más objetos o interfaces estándar necesarios para su implementación exitosa [13].

La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1994, publicaron Patrones de diseño: elementos de software orientado a objetos reutilizables, en el que aplicaron el concepto de patrones de diseño a la programación. El libro presentaba 23 patrones que solucionan diversos problemas del diseño orientado a objetos y se convirtió muy rápidamente en un éxito de ventas. Debido a su largo nombre, la gente empezó a llamarlo “el libro de la pandilla de los cuatro”, que pronto se redujo a simplemente “el libro de GoF” [14].

La clasificación de los patrones de diseño se basa en dos criterios fundamentales. El primero, denominado "propósito", describe la función que cumple el patrón, mientras que el segundo, llamado "alcance", especifica si el patrón se aplica principalmente a clases o a objetos. Así, los patrones se distribuyen en tres categorías distintas: creacional, estructural y conductual [15].

Patrones creacionales. Los patrones de diseño creacional son patrones de diseño que tratan con mecanismos de creación de objetos, tratando de crear objetos de una manera adecuada a la situación. La forma básica de creación de objetos podría generar problemas de diseño o agregar complejidad al diseño. Los patrones de diseño creacional resuelven este problema controlando de alguna manera la creación de este objeto [16], algunos ejemplos de estos patrones incluyen:

- **Factory.** Se trata de un patrón de diseño creacional que proporciona una interfaz en una superclase para la creación de objetos, permitiendo a las subclases modificar el tipo de objetos que se generarán [17].
- **Abstract Factory.** Se refiere a un patrón de diseño creacional que posibilita la generación de familias de objetos interrelacionados sin la necesidad de especificar sus clases concretas [18].
- **Builder.** Se trata de un patrón de diseño creacional que posibilita la construcción de objetos complejos de manera gradual y paso a paso. Este patrón facilita la creación de diferentes tipos y representaciones de un objeto mediante el mismo código de construcción [19].
- **Prototype.** Se refiere a un patrón de diseño creacional que posibilita la copia de objetos existentes sin que el código dependa de sus clases, también conocido como clonación [20].

- **Singleton.** Se trata de un patrón de diseño creacional que asegura que una clase cuente únicamente con una instancia, al mismo tiempo que proporciona un punto de acceso global a dicha instancia [21].

Patrones estructurales. Los patrones de diseño estructural son estrategias que simplifican el diseño al identificar formas efectivas de establecer relaciones entre entidades [22]. Ejemplos de estos patrones incluyen:

- **Adapter.** Es un patrón de diseño estructural que permite que objetos con interfaces incompatibles colaboren [23].
- **Bridge.** Se refiere a un patrón de diseño estructural que posibilita la subdivisión de una clase extensa o un conjunto de clases estrechamente vinculadas en dos jerarquías distintas (abstracción e implementación), permitiendo su desarrollo independiente [24].
- **Composite.** Se trata de un patrón de diseño estructural que facilita la composición de objetos en estructuras de árbol, permitiendo posteriormente trabajar con estas estructuras como si fueran objetos individuales [25].
- **Decorator.** Se trata de un patrón de diseño estructural que posibilita la incorporación de nuevos comportamientos a objetos al situarlos dentro de objetos envolventes especiales que encapsulan dichos comportamientos [26].
- **Facade.** Es un patrón de diseño estructural que suministra una interfaz simplificada para una biblioteca, un marco, o cualquier conjunto complejo de clases [27].
- **Flyweight.** Es un patrón de diseño estructural que permite optimizar el uso de la cantidad de RAM disponible al compartir partes comunes de estado entre varios objetos, en lugar de mantener todos los datos en cada objeto [28].
- **Proxy.** Se trata de un patrón de diseño estructural que facilita la creación de un sustituto o marcador de posición para otro objeto. Un proxy gestiona el acceso al objeto original, permitiéndole realizar acciones antes o después de que la solicitud alcance al objeto original [29].

Patrones Conductuales. Los patrones de diseño de comportamiento son estrategias que identifican patrones de comunicación frecuentes entre objetos e implementan dichos patrones. Al hacerlo, estos patrones aumentan la flexibilidad para llevar a cabo esta comunicación [30].

- **Chain of Responsibility.** Es un patrón de diseño de comportamiento que posibilita la transferencia de solicitudes a lo largo de una cadena de controladores. Cada controlador, al recibir una solicitud, toma la decisión de procesarla o pasarla al siguiente controlador en la cadena [31].

- **Command.** Se trata de un patrón de diseño de comportamiento que transforma una solicitud en un objeto independiente que encapsula toda la información relacionada con la solicitud. Esta transformación permite pasar solicitudes como argumentos de método, pospone o encola la ejecución de una solicitud, y facilita operaciones que pueden deshacerse [32].

- **Iterator.** Es un patrón de diseño de comportamiento que posibilita la iteración de elementos en una colección sin exponer su representación subyacente (como una lista, pila o árbol) [33].

- **Mediator.** Es un patrón de diseño de comportamiento que ayuda a disminuir las dependencias caóticas entre objetos. Este patrón restringe las comunicaciones directas entre los objetos y los obliga a colaborar únicamente a través de un objeto mediador [34].

- **Memento.** Se trata de un patrón de diseño de comportamiento que posibilita la conservación y restauración del estado previo de un objeto sin exponer los detalles de su implementación [35].

- **Observer.** Es un patrón de diseño de comportamiento que posibilita la definición de un mecanismo de suscripción para informar a varios objetos sobre cualquier evento que ocurra en el objeto que están observando [36].

- **State.** Es un patrón de diseño de comportamiento que habilita a un objeto para modificar su comportamiento cuando su estado interno experimenta cambios. Esto da la impresión de que el objeto está cambiando de clase [37].

- **Strategy.** Es un patrón de diseño de comportamiento que facilita la definición de una familia de algoritmos, encapsulando cada uno en una clase independiente y permitiendo que sus objetos sean intercambiables entre sí [38].

- **Template Method.** Es un patrón de diseño de comportamiento que establece la estructura básica de un algoritmo en la superclase, pero permite que las subclases anulen pasos específicos del algoritmo sin alterar su estructura [39].

- **Visitor.** Es un patrón de diseño de comportamiento que posibilita la separación de los algoritmos de los objetos sobre los que actúan [40].

SonarQube. SonarQube es una herramienta de revisión de código automática y autoadministrada que le ayuda sistemáticamente a ofrecer código limpio. Como elemento central de nuestra solución Sonar, SonarQube se integra en su flujo de trabajo existente y detecta problemas en su código para ayudarlo a realizar inspecciones continuas del código de sus proyectos. El producto analiza más de 30 lenguajes de programación diferentes y se integra en su canal de integración continua (CI) de plataformas DevOps para garantizar que su código cumpla con estándares de alta calidad [41].

NDepend. Es una herramienta .NET que proporciona una visión profunda de las bases de código. La herramienta permite a los desarrolladores, arquitectos y ejecutivos tomar decisiones inteligentes sobre los proyectos. La comunidad lo denomina la "navaja suiza" para los programadores de .NET [42].

Resultados

Una vez establecida la metodología y los materiales, se establece el caso de estudio que comprende un módulo para el manejo de usuarios. Módulo en el que se implementan vulnerabilidades específicamente identificadas.

En base a los riesgos expuestos por OWASP, se identifican los tres más críticos según la propia organización, considerados como los más perjudiciales para la seguridad. Según las pruebas realizadas por OWASP, el 94% de las aplicaciones evaluadas presentaban alguna forma de acceso de control expuesto, destacando como la problemática más crítica. En segundo lugar, se encuentran las fallas relacionadas con criptografía, que exponen información sensible o comprometen el sistema. Finalmente, en el tercer puesto, el 94\% de las aplicaciones probadas mostraron algún tipo de vulnerabilidad relacionada con inyección, siendo las más conocidas la inyección SQL, NoSQL, comando OS, mapeo relacional de objetos (ORM), LDAP y lenguaje de expresión (EL), así como la inyección de biblioteca de navegación de gráficos de objetos (OGNL) y el ataque XSS o Cross-Site Scripting [43].

Vulnerabilidades de control de acceso. El control de acceso roto se refiere a la capacidad de un atacante para obtener acceso o privilegios adicionales sin necesidad de autenticarse como otro usuario. Una manera en que se podría haber comprometido el control de acceso es mediante la implementación de políticas excesivamente permisivas. Si las políticas no se adhieren al principio de privilegio mínimo, se otorga más acceso del deseado, lo que lleva a un control de acceso inadecuado [44].

Criptografía débil y ataque de fuerza bruta. Un cifrado que posibilite un acceso no autorizado debe ser lo suficientemente robusto como para resistir ataques de fuerza bruta por parte de entidades no autorizadas. Sin embargo, la noción de cifrado ``débil" es dinámica y está en constante evolución. La diferencia entre descifrar una encriptación fuerte y débil mediante un ataque de fuerza bruta radica en el nivel de recursos computacionales dedicados a dicho ataque, y estos recursos experimentan un aumento constante. De hecho, el costo asociado a los ataques de fuerza bruta a la criptografía disminuye exponencialmente con el tiempo, de acuerdo con la ley de Moore [45].

Cross-Site Scripting. XSS pertenece a la categoría de ataques de inyección de código y representa una de las vulnerabilidades de seguridad más críticas en aplicaciones web. En este tipo de ataque, el atacante introduce de manera cuidadosa código JavaScript malicioso a través de los parámetros de entrada en el lado del cliente. El propósito de este acto es desencadenar acciones perjudiciales por parte de las aplicaciones web, con el objetivo de lograr distintos objetivos, como el robo de cookies y tokens de sesión, o para llevar a cabo otros tipos de ataques. La raíz del ataque XSS suele radicar en un filtrado

inadecuado del texto de entrada en el lado del cliente, permitiendo al atacante insertar fácilmente código malicioso en las páginas web basadas en OSN (redes sociales en línea). Estos scripts maliciosos se ejecutan en el navegador web del usuario, comprometiendo así la seguridad del cliente [46].

En el proceso de identificación de las vulnerabilidades más recurrentes, se inicia un análisis detallado con el objetivo de determinar el patrón de diseño más eficaz para mitigar cada problema identificado. Se presentarán los resultados de esta investigación mediante una tabla que considera las recomendaciones proporcionadas por la OWASP para abordar cada vulnerabilidad específica, permitiendo así una comparación detallada con los diversos patrones de diseño disponibles. Este enfoque sistemático facilitará la selección de los patrones más adecuados para cada contexto particular, respaldando la efectividad de las soluciones propuestas.

Con el propósito de demostrar las vulnerabilidades seleccionadas y su respectiva mitigación, se presentará un prototipo de código desarrollado en el lenguaje C\# Net CORE 8.0, diseñado para simular escenarios en los cuales se implementarán los patrones de diseño seleccionados. A través del uso de herramientas externas, se llevará a cabo una evaluación comparativa entre la versión inicial del código y la versión mejorada después de la aplicación del patrón correspondiente, específico para la vulnerabilidad identificada en el código propuesto. Este enfoque permitirá explicar de manera detallada el impacto derivado de la aplicación de buenas prácticas de programación, respaldando la eficacia de los patrones de diseño seleccionados.

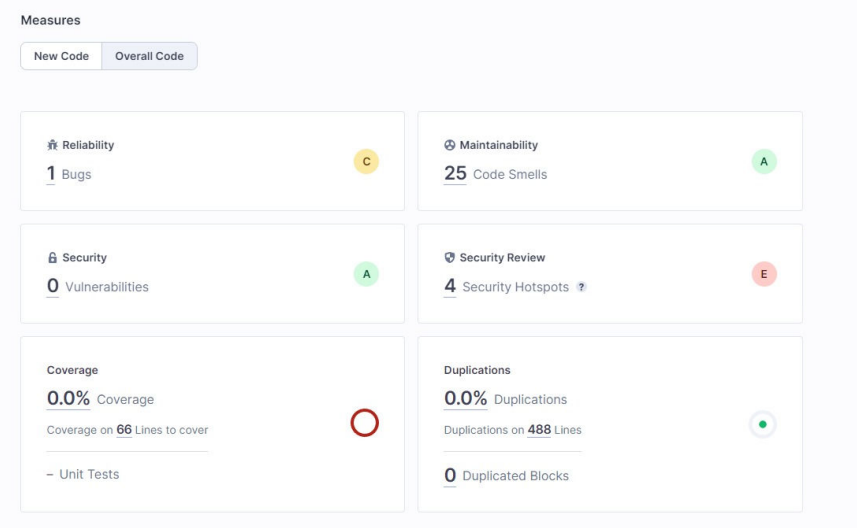
Como punto de partida, se llevó a cabo un análisis de la aplicación mediante el empleo de las herramientas SonarQube y NDepend, las cuales facilitan la realización de un análisis estático del código. A continuación, se presentan los resultados obtenidos.

El análisis con la herramienta SonarQube proporciona una visión integral del estado del código, destacando aspectos como bugs, código no óptimo, vulnerabilidades y puntos críticos de seguridad.

El análisis realizado con la herramienta NDepend proporciona un gráfico que evidencia la relación entre la "Abstractness" (Abstracto) y la "Instability" (Inestabilidad).

Figura 1

Análisis Inicial con SonarQube

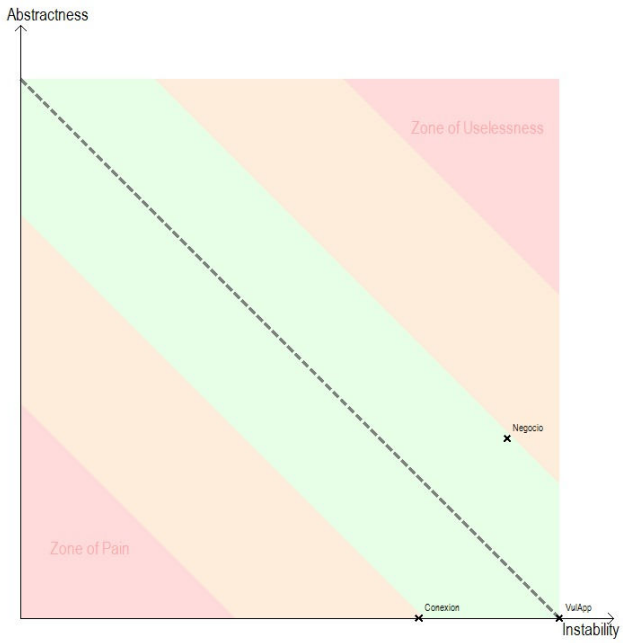


Fuente: SonarQube

El análisis revela la presencia de cuatro revisiones de seguridad significativas, junto con la identificación de un bug en el código.

Figura 2

Análisis Inicial con NDepend



Fuente: Ndepend

El gráfico ilustra que la aplicación se sitúa en el cuadrante inferior derecho, indicando que posee baja abstracción pero es frecuentemente utilizada, lo que facilita su mantenimiento. No obstante, la gráfica también señala que la aplicación se encuentra en los límites admisibles, sugiriendo la posibilidad de entrar en un estado denominado "zona de dolor" y "zona de inutilidad".

A continuación se muestra una tabla con los resultados obtenidos de la comparación entre las vulnerabilidades y los patrones de diseño.

Cuadro 1

Resultado del caso denominado Broken Access Control

Vulnerabilidad	**Patrón de diseño sugerido**
Broken Access Control	Chain of Responsibility
Criterio de Implementación	
Patrón de diseño de comportamiento el cual establece que cada petición debe pasar a través de una cadena de validadores o "manejadores", los cuales definen si pasa a la siguiente fase o no.	

La vulnerabilidad conocida como Broken Access Control se aborda de manera efectiva mediante la implementación del patrón de diseño Chain of Responsibility. Este patrón establece una cadena de objetos que manejan las solicitudes de forma secuencial. Cada objeto en la cadena decide si asume la solicitud o la transfiere al siguiente en la cadena. Este enfoque resulta beneficioso para verificar y autorizar el acceso en una serie de pasos. La aplicación de Chain of Responsibility posibilita una gestión más flexible y escalable de las reglas de autorización, ya que cada eslabón puede especializarse en una tarea específica sin generar un fuerte acoplamiento en el código. Además, facilita la extensión y modificación de las reglas de acceso sin afectar otras partes del sistema.

Figura 3

Código sin patrón de diseño (Caso Broken-Access-Control)

```
[Authorize]
[HttpGet(Name = "DameTodosUsuarios")]
[ProducesResponseType(typeof(Unit), StatusCodes.Status200OK)]
0 referencias
public async Task<IActionResult> DameTodosUsuarios()
{
    var result = await _usuarioRepo.DameTodosUsuarios();
    return Ok(result);
}
```

Fuente: Juan Mesías

Figura 4

Código con patrón de diseño (Caso Broken-Access-Control)

```
[Authorize]
[ApiAdmin]
[HttpGet(Name = "DameTodosUsuarios")]
[ProducesResponseType(typeof(Unit), StatusCodes.Status200OK)]
0 referencias
public async Task<IActionResult> DameTodosUsuarios()
{
    var result = await _usuarioRepo.DameTodosUsuarios();
    return Ok(result);
}
```

Fuente: Juan Mesías

En base a la definición del patrón “Chain of Responsibility” se crearon atributos adaptables y secuenciales para cada acción o para cada controlador de la solución, lo que permite ir encadenando funciones de validación que en caso de ser correctas continuarán con el proceso normalmente, caso contrario simplemente el acceso será denegado. En este caso el atributo “ApiAdmin” valida el rol del usuario que quiere acceder a la API.

Cuadro 2

Resultado del caso denominado Fuerza bruta e inyección SQL

Vulnerabilidad	**Patrón de diseño sugerido**
Fuerza Bruta	State (Rate Limiting)
Criterio de Implementación	
Patrón de diseño de conducta que limita el número de peticiones al servidor, además de otros recursos que influyen en la interacción con el servidor.	

La vulnerabilidad conocida y clasificada como ataques de fuerza bruta, encuentra una solución efectiva mediante la aplicación del patrón de diseño denominado "State". Este patrón permite que un objeto modifique su comportamiento en respuesta a cambios en su estado interno. En el contexto de la seguridad y la autenticación, el patrón State puede emplearse para representar el estado de un usuario durante el proceso de inicio de sesión. Por ejemplo, después de un número específico de intentos fallidos, el objeto podría cambiar su estado, imponiendo restricciones adicionales o aumentando la demora entre intentos. Es relevante destacar que la mitigación efectiva de ataques de fuerza bruta se logra comúnmente mediante prácticas como la Limitación de Tasa (Rate Limiting), una estrategia que controla la frecuencia de ciertas operaciones para prevenir abusos o ataques.

Figura 5

Código sin patrón de diseño (Caso Fuerza Bruta e Inyección SQL)

```
0 referencias
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    if (env.IsDevelopment())
    {
        app.UseSwagger();
        app.UseSwaggerUI();
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app
        .UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        });

}
```

Fuente: Juan Mesías

Figura 6

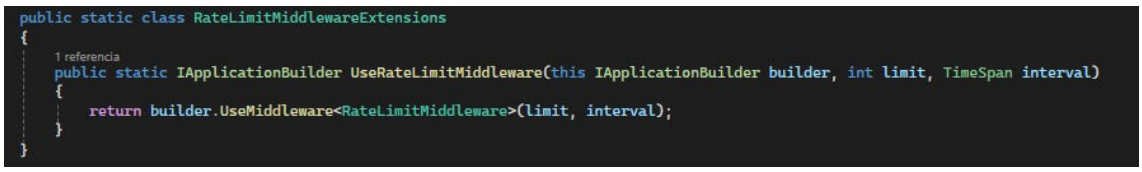
Código con patrón de diseño (Caso Fuerza Bruta e Inyección SQL)

```
0 referencias
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{

    if (env.IsDevelopment())
    {
        app.UseSwagger();
        app.UseSwaggerUI();
        app.UseDeveloperExceptionPage();
    }

    app.UseRateLimitMiddleware(100, TimeSpan.FromMinutes(1));

    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app
        .UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        });

}
```

Fuente: Juan Mesías

Figura 7

Código con patrón de diseño (Caso Fuerza Bruta e Inyección SQL - Implementación con middleware)



```
public static class RateLimitMiddlewareExtensions
{
    1 referencia
    public static IApplicationBuilder UseRateLimitMiddleware(this IApplicationBuilder builder, int limit, TimeSpan interval)
    {
        return builder.UseMiddleware<RateLimitMiddleware>(limit, interval);
    }
}
```

Fuente: Juan Mesías

En base a la definición del patrón "Rate Limiter" establece que se debe configurar el número de peticiones que son procesadas por una dirección IP en un determinado rango de tiempo, de esta manera se implementó un middleware que a través de la caché almacena y verificará el número de peticiones de un cliente en un minuto en caso de exceder este número de peticiones se le denegara el acceso a la funcionalidad.

Cuadro 3

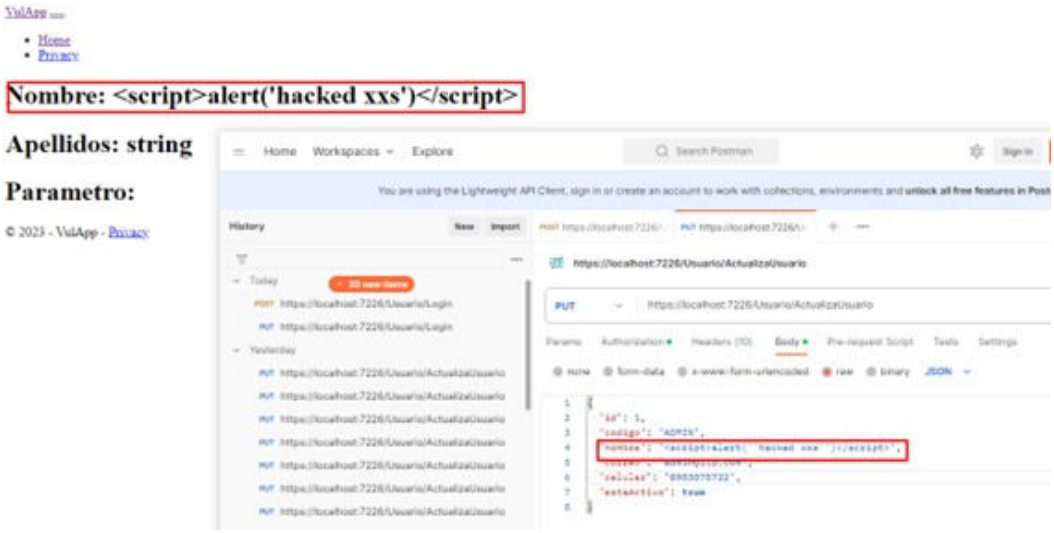
Resultado del caso denominado Cross-Site Scripting

Vulnerabilidad	**Patrón de diseño sugerido**
Cross-Site Scripting	Proxy
Criterio de Implementación	
Patrón de diseño estructural que permite la validación del objeto enviado previo al procesamiento de la petición.	

La vulnerabilidad de Cross-Site Scripting (XSS) puede abordarse eficazmente mediante la aplicación del patrón de diseño Proxy, el cual actúa como un intermediario que supervisa y controla el acceso a otro objeto, permitiendo o denegando el acceso según condiciones predefinidas. El Proxy desempeña un papel crucial al realizar la validación de entrada, asegurándose de que los datos ingresados por el usuario cumplan con criterios específicos antes de pasarlos al módulo crucial. Además, el Proxy puede implementar mecanismos adicionales para reforzar la seguridad, como el escape de caracteres para prevenir la introducción de caracteres especiales y el uso de encabezados HTTP de seguridad. Esta combinación de técnicas ayuda a mitigar el riesgo de ataques XSS al establecer un control más riguroso sobre los datos de entrada y sus interacciones.

Figura 7

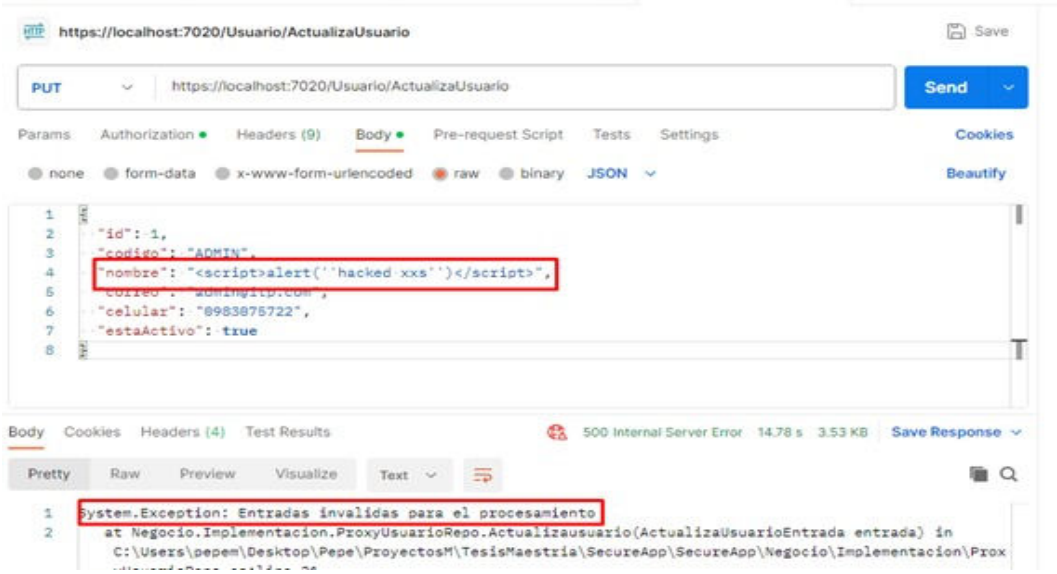
Código sin patrón de diseño (Caso Cross Site Scripting)



Fuente: Juan Mesías

Figura 8

Código con patrón de diseño (Caso Cross Site Scripting)



Fuente: Juan Mesías

Figura 8

Código con patrón de diseño (Caso Cross Site Scripting - Implementación)

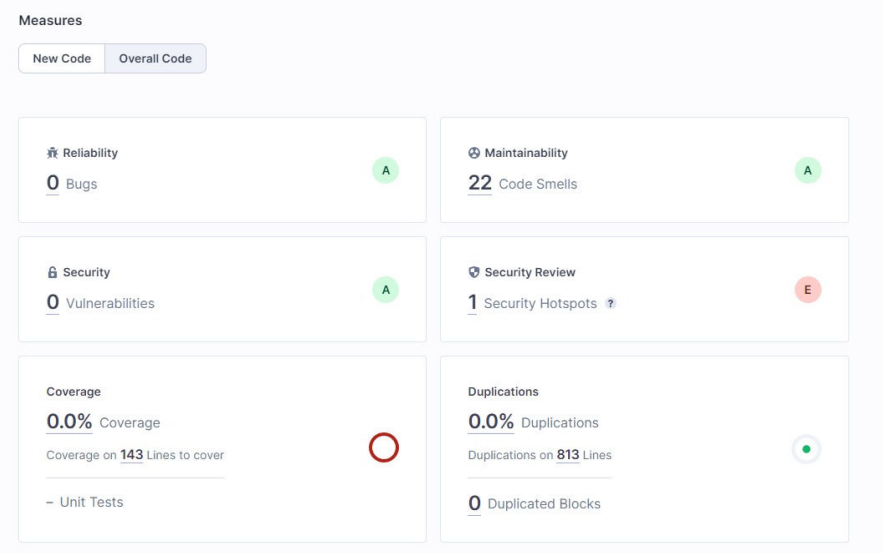
```
public class ProxyUsuarioRepo : IUsuarioRepo
{
    private readonly UsuarioRepo _usuarioRepo;
    private readonly DapperContext _dapperContext;
    private readonly IConfiguration _configuration;
    0 referencias
    public ProxyUsuarioRepo(UsuarioRepo usuariorepo, DapperContext context, IConfiguration config)
    {
        _usuarioRepo = usuariorepo;
        _dapperContext = context;
        _configuration = config;
    }

    2 referencias
    public async Task<int> Actualizausuario(ActualizaUsuarioEntrada entrada)
    {
        await Task.CompletedTask;


        if (SonEntradasSeguras([entrada.Codigo, entrada.Nombre, entrada.Correo, entrada.Celular]))
            await _usuarioRepo.Actualizausuario(entrada);
        else
            throw new Exception("Entradas invalidas para el procesamiento");

        return 0;
    }
```

Fuente: Juan Mesías

En base a la definición del patrón “Proxy” en donde se indica que para agregar un control o una funcionalidad adicional a la función original se puede implementar un intermediario el cual sea el encargado de realizar el control con una implementación extra sin modificar el código original.

Figura 9

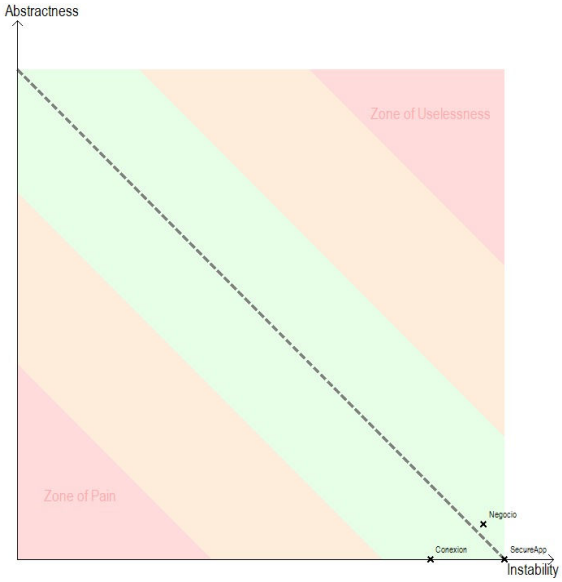
Análisis Final con SonarQube



Fuente: Juan Mesías

Figura 9

Análisis Final con NDepend



Fuente: Juan Mesías

El gráfico ilustra que la aplicación se sitúa en el cuadrante inferior derecho. Sin embargo, tras la aplicación de las recomendaciones, los componentes de la misma tienden a sesgar hacia un punto intermedio entre las variables establecidas.

Conclusiones

Con el desarrollo de la presente investigación se pudo identificar que los patrones de diseño proveen de soluciones prácticas y funcionales para una o varias vulnerabilidades existentes. La investigación se desarrolló con la utilización de herramientas proveídas por un framework de desarrollo con un enfoque sobre los patrones de diseño de esta manera se controló la aplicación de una manera comportamental y estructural para evitar vulnerabilidades importantes.

En el desarrollo de la investigación se identificó que los patrones de diseño tienen una segmentación muy bien definida por lo que según la interpretación del desarrollador y basándose en la brecha de seguridad que desee solventar en la aplicación, estos pueden ser utilizados de diferentes maneras con diferentes enfoques e incluso un patrón puede cubrir una o más necesidades de seguridad. Los patrones de diseño no nos dictan por regla que su implementación debe ser del lado del cliente o del servidor todo se engloba en la percepción del desarrollador.

Se identificó que los patrones más influyentes sobre las vulnerabilidades seleccionadas son los patrones que definen una estructura (Estructurales) y proponen un comportamiento establecido (comportamentales) sobre la aplicación, de esta manera se logró sustentar la mitigación de las vulnerabilidades identificadas.

Referencias bibliográficas

[1] C. Ltd, *Mastering OWASP*. Cybellium Ltd, 2023.

[2] S. C. M. Arante, *Auditoría de la Seguridad Informática*. Ra-Ma Editorial, 2022.

[3] "A01 Broken Access Control," *OWASP Top 10:2021*. https://owasp.org/Top10/A01_2021-Broken_Access_Control/ (accessed Dec. 19, 2023).

[4] "A02 Cryptographic Failures," *OWASP Top 10:2021*. https://owasp.org/Top10/A02_2021-Cryptographic_Failures/ (accessed Dec. 19, 2023).

[5] "A03 Injection," *OWASP Top 10:2021*. https://owasp.org/Top10/A03_2021-Injection/ (accessed Dec. 19, 2023).

[6] "A04 Insecure Design," *OWASP Top 10:2021*. https://owasp.org/Top10/A04_2021-Insecure_Design/ (accessed Dec. 19, 2023).

[7] "A05 Security Misconfiguration," *OWASP Top 10:2021*. https://owasp.org/Top10/A05_2021-Security_Misconfiguration/ (accessed Dec. 19, 2023).

[8] "A06 Vulnerable and Outdated Components," *OWASP Top 10:2021*. https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/ (accessed Dec. 19, 2023).

[9] "A07 Identification and Authentication Failures," *OWASP Top 10:2021*. https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/ (accessed Dec. 19, 2023).

[10] "A08 Software and Data Integrity Failures," *OWASP Top 10:2021*. https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/ (accessed Dec. 19, 2023).

[11] "A09 Security Logging and Monitoring Failures," *OWASP Top 10:2021*. https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/ (accessed Dec. 19, 2023).

[12] "A10 Server Side Request Forgery (SSRF)," *OWASP Top 10:2021*. https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_(SSRF) (accessed Dec. 19, 2023).

[13] L. Mehra, *Software Design Patterns for Java Developers: Expert-led Approaches to Build Re-usable Software and Enterprise Applications (English Edition)*. BPB Publications, 2021.

[14] S. Holzner, *Design Patterns For Dummies*. John Wiley & Sons, 2006.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1995.
[16] "Design Patterns and Refactoring." https://sourcemaking.com/design_patterns/creational_patterns (accessed Dec. 19, 2023).
[17] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/factory-method (accessed Dec. 19, 2023).
[18] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/abstract-factory (accessed Dec. 19, 2023).
[19] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/builder (accessed Dec. 19, 2023).
[20] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/prototype (accessed Dec. 19, 2023).
[21] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/singleton (accessed Dec. 19, 2023).
[22] "Design Patterns and Refactoring." https://sourcemaking.com/design_patterns/structural_patterns (accessed Dec. 19, 2023).
[23] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/adapter (accessed Dec. 19, 2023).
[24] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/bridge (accessed Dec. 19, 2023).
[25] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/composite (accessed Dec. 19, 2023).
[26] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/decorator (accessed Dec. 19, 2023).
[27] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/facade (accessed Dec. 19, 2023).
[28] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/flyweight (accessed Dec. 19, 2023).
[29] "Alexander Shvets," Jan. 01, 2022. https://refactoring.guru/design-patterns/proxy (accessed Dec. 19, 2023).
[30] "Design Patterns and Refactoring." https://sourcemaking.com/design_patterns/behavioral_patterns (accessed Dec. 19, 2023).

[31] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/chain-of-responsibility (accessed Dec. 19, 2023).
[32] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/command (accessed Dec. 19, 2023).
[33] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/iterator (accessed Dec. 19, 2023).
[34] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/mediator (accessed Dec. 19, 2023).
[35] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/memento (accessed Dec. 19, 2023).
[36] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/observer (accessed Dec. 19, 2023).
[37] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/state (accessed Dec. 19, 2023).
[38] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/strategy (accessed Dec. 19, 2023).
[39] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/template-method (accessed Dec. 19, 2023).
[40] “Alexander Shvets,” Jan. 01, 2022. https://refactoring.guru/design-patterns/visitor (accessed Dec. 19, 2023).
[41] “SonarQube 10.3.” https://docs.sonarsource.com/sonarqube/latest/ (accessed Dec. 19, 2023).
[42] “Features,” *NDepend*. https://www.ndepend.com/features/ (accessed Dec. 19, 2023).
[43] “OWASP Top Ten,” *OWASP Foundation*. https://owasp.org/www-project-top-ten/ (accessed Dec. 19, 2023).
[44] D. Shields, *AWS Security*. Simon and Schuster, 2022.
[45] N. R. Council, D. on E. and P. Sciences, C. S. and T. Board, and C. to S. N. C. Policy, *Cryptography’s Role in Securing the Information Society*. National Academies Press, 1996.

Conflicto de intereses:
Los autores declaran que no existe conflicto de interés posible.
Financiamiento:
No existió asistencia financiera de partes externas al presente artículo.
Agradecimiento:
N/A
Nota:
El artículo no es producto de una publicación anterior.