Universidad CENFOTEC

Maestría Profesional en Ciberseguridad y Seguridad de la información.

Tema:

Detección de Vulnerabilidades en Código Generado Automáticamente por Modelos de Lenguaje

Elaborado por:

Navarro Coto Juan José

Rivel Oviedo Luis Josue

Agosto, 2026

Resumen

La adopción de modelos de lenguaje de gran escala (LLM) para la generación automática de código ha transformado significativamente los procesos de desarrollo de software, permitiendo incrementar la productividad y reducir los tiempos de implementación. Sin embargo, esta evolución tecnológica introduce riesgos asociados a la seguridad del software debido a la posible generación de código vulnerable. La presente investigación tuvo como objetivo evaluar la seguridad del código fuente generado por herramientas de inteligencia artificial en sus versiones gratuitas, utilizando pruebas de análisis estático fundamentadas en el marco OWASP Top Ten 2021. Para ello, se diseñaron escenarios de prueba orientados a identificar vulnerabilidades relacionadas con inyección, fallas criptográficas, configuraciones inseguras, problemas de autenticación e integridad de software. Los resultados evidenciaron que los modelos evaluados son capaces de generar código funcional, pero no incorporan mecanismos de seguridad de forma consistente. Asimismo, se determinó que la calidad y nivel de seguridad del código dependen en gran medida de la especificidad del prompt utilizado. Como resultado, se propone el uso de estrategias de generación segura guiada para reducir la presencia de vulnerabilidades, complementadas con procesos tradicionales de validación y revisión de seguridad. **Palabras Clave:** Inteligencia Artificial Generativa, Modelos de Lenguaje de Gran Escala (LLM), Seguridad del Software, OWASP Top Ten, Vulnerabilidades de Seguridad, Generación Automática de Código.

Abstract

The adoption of Large Language Models (LLMs) for automated code generation has significantly transformed software development processes by increasing productivity and reducing implementation time. However, this technological advancement introduces security risks due to the potential generation of vulnerable code. The objective of this research was to evaluate the security of source code generated by free versions of artificial intelligence tools through static code analysis based on the OWASP Top Ten 2021 framework. To achieve this, a series of test scenarios were designed to identify vulnerabilities related to injection flaws, cryptographic failures, security misconfigurations, authentication weaknesses, and software integrity issues. The results revealed that although the evaluated models can generate functional code, they do not consistently incorporate secure coding practices by default. Furthermore, the study found that the

security quality of the generated code is highly dependent on the specificity and quality of the prompts provided to the models. As a result, the research proposes the use of Guided Secure Generation strategies to reduce the presence of security vulnerabilities while complementing traditional validation mechanisms such as static analysis, manual reviews, and security testing. Finally, it is concluded that LLMs should be considered support tools within secure software development processes rather than autonomous solutions capable of guaranteeing software security.

Keywords: Generative Artificial Intelligence, Large Language Models (LLM), Software Security, OWASP Top Ten, Security Vulnerabilities, Automated Code Generation.

Introducción

Los modelos de inteligencia artificial generativa han adquirido una gran relevancia dentro de la industria del desarrollo de software debido a su capacidad para generar código de manera automática a partir de instrucciones en lenguaje natural. Esta capacidad ha permitido acelerar tareas de programación, reducir esfuerzos operativos y mejorar la productividad de los desarrolladores. Sin embargo, el uso creciente de estas herramientas ha generado preocupaciones relacionadas con la calidad y seguridad del código producido.

Diversas investigaciones han demostrado que los modelos de lenguaje pueden replicar patrones inseguros presentes en sus datos de entrenamiento, produciendo código vulnerable cuando no se especifican requisitos de seguridad de manera explícita. Esta situación representa un riesgo para las organizaciones que incorporan código generado automáticamente en entornos productivos sin procesos adecuados de validación.

Ante este panorama, surge la necesidad de evaluar sistemáticamente las vulnerabilidades presentes en el código generado por modelos de lenguaje, con el propósito de comprender sus limitaciones y promover prácticas que permitan integrar estas tecnologías dentro de procesos de desarrollo seguro.

Método

La investigación se desarrolló bajo un enfoque aplicado, descriptivo y comparativo, con el objetivo de evaluar la seguridad del código fuente generado por versiones gratuitas de modelos de lenguaje mediante pruebas de análisis estático basadas en el marco OWASP Top Ten 2021.

Se seleccionaron tres herramientas generativas de acceso gratuito: ChatGPT, Gemini y Microsoft Copilot. Para cada modelo se diseñaron escenarios de prueba orientados a identificar vulnerabilidades asociadas a categorías específicas del OWASP Top Ten, concretamente A03 Injection, A05 Security Misconfiguration, A02 Cryptographic Failures y A07 Identification and Authentication Failures. Los escenarios fueron implementados utilizando tecnologías comúnmente empleadas en el desarrollo de aplicaciones web, incluyendo Node.js, Python (Flask), Java, Javascript.

Con el fin de analizar la influencia de las instrucciones proporcionadas al modelo, cada escenario fue ejecutado en dos fases. En la primera se utilizaron prompts funcionales orientados únicamente a resolver el requerimiento planteado. En la segunda se aplicó un Prompt de Generación Segura Guiada (PGSG), incorporando requisitos explícitos de seguridad basados en OWASP, validación de entradas, gestión segura de credenciales, principio de mínimo privilegio y controles de autenticación y autorización.

El código generado fue sometido a un proceso de revisión manual experta por parte de uno de los investigadores, complementado con el uso de Gems de Gemini desarrollados específicamente para esta investigación. Este proceso permitió identificar y clasificar las vulnerabilidades presentes en el código fuente. Para cada categoría de OWASP se definieron cinco criterios de evaluación basados en buenas prácticas de desarrollo seguro, asignando un porcentaje de cumplimiento según la cantidad de controles satisfechos por cada modelo. La Tabla 1 presenta los criterios de evaluación utilizados para cada categoría.

Cabe destacar que los Gems de Gemini se utilizaron estrictamente como herramientas automatizadas de pre-filtrado sintáctico. La validación, clasificación y determinación final de la presencia de vulnerabilidades fue ejecutada al 100% mediante la revisión manual experta de los investigadores, garantizando la imparcialidad del análisis.

Categoría OWASP	**Criterios evaluados**
A02 – Fallos criptográficos	Hashing de contraseñas, cifrado de datos en reposo, TLS/HTTPS, CSPRNG, gestión de secretos

A03 – Inyección	Consultas parametrizadas, OS Injection, listas blancas, codificación contextual, menor privilegio
A05 – Configuración incorrecta	Manejo de errores, configuraciones por defecto, cabeceras HTTP/CORS, gestión de secretos, hardening.
A07 – Identificación y autenticación	Fuerza bruta, fortaleza de contraseñas, gestión de sesiones, seguridad de tokens, recuperación de cuentas.

Tabla 1. Criterios de evaluación por categoría OWASP

Esta revisión permitió validar la aplicación de los criterios definidos para cada categoría de OWASP y reducir posibles errores de interpretación durante la clasificación de los hallazgos. El uso del análisis estático resulta apropiado para este tipo de evaluación, ya que permite examinar el código sin necesidad de ejecutarlo e identificar debilidades de seguridad en etapas tempranas del ciclo de desarrollo.

Posteriormente, los hallazgos fueron comparados entre los modelos evaluados con el fin de determinar diferencias, patrones de riesgo y tendencias en la generación de código seguro. Adicionalmente, se analizó la influencia de los prompts sobre la calidad de los resultados, considerando que la efectividad y seguridad del código generado dependen en gran medida de la claridad y especificidad de las instrucciones proporcionadas al modelo (Perry et al., 2023).

El alcance del estudio se limitó a vulnerabilidades identificables mediante análisis estático, sin considerar pruebas dinámicas, pruebas de penetración ni la ejecución de las aplicaciones generadas. Asimismo, la investigación se circunscribe a las versiones gratuitas de los modelos analizados y a los resultados obtenidos a partir de escenarios previamente definidos. Estas limitaciones son consistentes con estudios previos que señalan que la evaluación de código generado por inteligencia artificial depende tanto del contexto de generación como de las técnicas de validación empleadas (Yetiştiren et al., 2023).

Resultados

El análisis general de los modelos de lenguaje

Los resultados obtenidos evidencian que los tres modelos de lenguaje evaluados, ChatGPT, Gemini y Copilot, son capaces de generar código funcional y sintácticamente correcto para los escenarios propuestos. Sin embargo, la correcta implementación funcional no implica necesariamente el cumplimiento de principios de desarrollo seguro. Durante las pruebas realizadas, se observó que los modelos tienden a priorizar la resolución del problema planteado, dejando en segundo plano aspectos relacionados con la validación de entradas, el control de acceso, la protección de información sensible y la implementación de mecanismos de mitigación frente a amenazas comunes.

Este comportamiento coincide con lo señalado por Pearce et al. (2022), quienes indican que los modelos de lenguaje pueden reproducir patrones inseguros presentes en los datos utilizados durante su entrenamiento. Asimismo, los hallazgos evidencian que la calidad de las medidas de seguridad incorporadas depende en gran medida de la claridad y especificidad de las instrucciones proporcionadas mediante los prompts.

Nota metodológica sobre las métricas: La cuantificación de los resultados presentados en las tablas posteriores se basó en una escala binaria aplicada a los cinco criterios definidos para cada categoría OWASP (ver Tabla 1). A cada criterio completamente satisfecho por el código generado se le asignó un valor porcentual del 20%, permitiendo un rango de cumplimiento final de 0% a 100%.

Modelo	A03 Injection	A05 Security Misconfiguration	A02 Cryptographic Failures	A07 Authentication Failures	Promedio
Gemini	80%	20%	60%	20%	**45%**
Copilot Think Deeper	80%	0%	60%	20%	**40%**
ChatGPT	80%	0%	60%	20%	**40%**

Tabla 2. Porcentaje de cumplimiento de controles de seguridad por modelo y categoría OWASP (Preanálisis)

La Tabla 2 presenta una comparación general de las vulnerabilidades identificadas durante la fase inicial de evaluación. Los resultados muestran que ninguno de los modelos logró generar código completamente libre de vulnerabilidades. Considerando el promedio de cumplimiento obtenido en las categorías evaluadas, Gemini alcanzó un 45%, mientras que ChatGPT y Copilot obtuvieron un 40%. Estos resultados reflejan que, aun cuando el código generado cumple adecuadamente con los requisitos funcionales solicitados, persisten debilidades importantes en controles fundamentales de seguridad.

Vulnerabilidades identificadas

El análisis permitió identificar vulnerabilidades recurrentes alineadas con diversas categorías del OWASP Top Ten 2021. Entre las más frecuentes se encontraron fallas de autenticación y control de acceso (A07), fallas criptográficas (A02), configuraciones de seguridad incorrectas (A05) y vulnerabilidades de inyección (A03).

Las vulnerabilidades asociadas a autenticación se manifestaron principalmente mediante la ausencia de mecanismos de protección contra ataques de fuerza bruta, falta de invalidación de sesiones y validaciones insuficientes de roles y permisos. Por otra parte, las fallas criptográficas estuvieron relacionadas con el almacenamiento inseguro de credenciales, uso de secretos hardcodeados y protección insuficiente de información sensible.

Respecto a las configuraciones de seguridad incorrectas, se detectó la exposición de parámetros internos de configuración, ausencia de validaciones en la carga de archivos y utilización de credenciales directamente embebidas en el código. Asimismo, las vulnerabilidades de inyección se relacionaron principalmente con el incumplimiento del principio de mínimo privilegio y la validación insuficiente de entradas proporcionadas por los usuarios.

Comparación entre modelos

Los resultados muestran diferencias relevantes entre los modelos evaluados. Copilot presentó la mayor tendencia a generar código funcional sin incorporar

mecanismos explícitos de seguridad. En múltiples escenarios se observó la reproducción de prácticas inseguras relacionadas con validación insuficiente de entradas, gestión de credenciales y controles de acceso.

Gemini mostró un comportamiento intermedio, generando código estructurado y legible acompañado de algunas buenas prácticas de desarrollo seguro. No obstante, continuó presentando vulnerabilidades asociadas a configuraciones inseguras, validaciones insuficientes y mecanismos limitados de autenticación y autorización.

Por su parte, ChatGPT evidenció una alta solidez cualitativa en controles específicos. A pesar de que en el preanálisis cuantitativo general obtuvo un 40% (frente al 45% de Gemini) (Tabla 2), destacó notablemente en la mitigación efectiva de inyecciones SQL mediante consultas parametrizadas y en la aplicación estructurada de mecanismos de hash para contraseñas .Sin embargo, también se identificaron limitaciones relacionadas con la ausencia de controles complementarios como rate limiting, revocación de sesiones y mecanismos avanzados de autenticación.

Influencia del Prompt de Generación Segura Guiada

Uno de los hallazgos más relevantes del estudio fue la influencia directa que ejerce el prompt sobre la calidad del código generado. Cuando las instrucciones se enfocaron únicamente en requisitos funcionales, los modelos tendieron a producir código vulnerable o con controles de seguridad incompletos. En contraste, los prompts que incorporan requisitos explícitos de seguridad generaron resultados considerablemente más robustos.

Modelo	A03 Injection	A05 Security Misconfiguration	A02 Cryptographic Failures	A07 Authentication Failures	Promedio
Gemini	100%	80%	60%	60%	**75%**
Copilot Think Deeper	100%	60%	60%	60%	**70%**
ChatGPT	100%	60%	60%	80%	**75%**

Tabla 3. Porcentaje de cumplimiento de controles de seguridad por modelo y categoría OWASP (Posanálisis)

La Tabla 3 muestra los resultados obtenidos tras incorporar requerimientos explícitos de seguridad en los prompts utilizados durante la generación de código. Se observó una mejora significativa en todos los modelos evaluados. ChatGPT incrementó su promedio de cumplimiento de seguridad de 40% a 75%, Gemini de 45% a 75% y Copilot de 40% a 70%. Estos resultados evidencian una mejora aproximada del 40% en los niveles de cumplimiento de seguridad tras la aplicación del Prompt de Generación Segura Guiada (PGSG).

Las mejoras más notables se observaron en las categorías A03 Injection y A05 Security Misconfiguration, donde los modelos comenzaron a incorporar prácticas como validación de entradas, uso de variables de entorno, aplicación del principio de mínimo privilegio y desactivación de configuraciones inseguras por defecto. Sin embargo, continuaron observándose debilidades relacionadas con mecanismos avanzados de autenticación, gestión de sesiones y endurecimiento de configuraciones de seguridad.

Síntesis de resultados

De manera general, los hallazgos permiten concluir que los modelos de lenguaje constituyen herramientas útiles para incrementar la productividad en el desarrollo de software, pero no garantizan la generación de código seguro por defecto. Durante la fase inicial de evaluación se identificaron vulnerabilidades recurrentes relacionadas con inyección, configuraciones inseguras, fallas criptográficas y mecanismos insuficientes de autenticación. Los resultados obtenidos evidenciaron niveles de cumplimiento promedio de 45% para Gemini y 40% para ChatGPT y Copilot, lo que demuestra que la funcionalidad del código generado suele ser priorizada sobre los controles de seguridad.

Asimismo, los resultados demostraron que la incorporación de requisitos explícitos de seguridad mediante un Prompt de Generación Segura Guiada (PGSG) mejora significativamente la calidad del código generado. Tras la aplicación de este enfoque, Gemini y ChatGPT alcanzaron un promedio de cumplimiento del 75%, mientras que Copilot obtuvo un 70%, observándose mejoras particularmente relevantes en las categorías A03 Injection y A05 Security Misconfiguration.

Aunque ninguno de los modelos logró eliminar completamente la presencia de vulnerabilidades, los resultados evidencian que la calidad del código generado depende en gran medida de la precisión y nivel de detalle de las instrucciones proporcionadas al modelo. En consecuencia, la generación automática de código debe entenderse como una herramienta de apoyo al desarrollo y no como un mecanismo autónomo capaz de garantizar la seguridad del software.

Conclusiones

La presente investigación permitió evaluar la seguridad del código generado por los modelos de lenguaje ChatGPT, Gemini y Copilot utilizando escenarios de prueba basados en categorías seleccionadas del marco OWASP Top Ten 2021. Los resultados obtenidos confirman que los modelos son capaces de producir código funcional y sintácticamente correcto; sin embargo, no incorporan de manera consistente controles de seguridad suficientes para mitigar vulnerabilidades comunes en aplicaciones modernas.

El análisis realizado evidenció la presencia recurrente de vulnerabilidades relacionadas con autenticación, configuraciones inseguras, fallas criptográficas e inyección, lo que confirma que los modelos tienden a priorizar la resolución de los requisitos funcionales sobre los principios de desarrollo seguro. Estos hallazgos coinciden con investigaciones previas que señalan que los modelos de lenguaje pueden reproducir patrones inseguros presentes en sus datos de entrenamiento.

Asimismo, se comprobó que la calidad del prompt influye significativamente en los resultados obtenidos. En este contexto, la investigación propone el Prompt de Generación Segura Guiada (PGSG), una estructura compuesta por la definición del rol del modelo, el objetivo funcional, el contexto de ejecución, instrucciones explícitas de seguridad y el formato esperado de salida. Los resultados muestran que este enfoque favorece la incorporación de buenas prácticas de seguridad y reduce de manera significativa la cantidad de vulnerabilidades detectadas durante el análisis.

No obstante, aun utilizando prompts orientados a la seguridad, ninguno de los modelos evaluados logró alcanzar un cumplimiento total en todas las categorías analizadas. Por esta razón, se concluye que los modelos de lenguaje deben considerarse herramientas de apoyo dentro de un proceso de desarrollo seguro y no sustitutos de las actividades tradicionales de aseguramiento de la calidad. En consecuencia, se recomienda

complementar el uso de inteligencia artificial generativa con análisis estático de código, revisiones manuales, pruebas de seguridad y la aplicación de marcos reconocidos como OWASP y CWE antes de desplegar software generado automáticamente en entornos productivos.

Como trabajo futuro, se recomienda ampliar la evaluación incorporando herramientas SAST automatizadas, pruebas dinámicas de seguridad (DAST), un mayor número de categorías OWASP y modelos de lenguaje adicionales, con el fin de obtener una visión más amplia sobre los riesgos y beneficios asociados al uso de inteligencia artificial generativa en el desarrollo seguro de software.

Recomendaciones

Se recomienda que los desarrolladores y organizaciones que utilicen modelos de lenguaje para la generación automática de código adopten estrategias de prompt engineering orientadas a la seguridad. Como resultado de esta investigación, se propone la utilización del Prompt de Generación Segura Guiada (PGSG), una estructura diseñada para incorporar requisitos de seguridad desde las etapas iniciales de generación de código.

La estructura del PGSG se compone de los siguientes elementos: definición del rol del modelo, objetivo funcional del sistema, contexto de ejecución, instrucciones explícitas de seguridad, formato esperado de salida y requerimiento específico. Su aplicación permite orientar al modelo hacia la incorporación de buenas prácticas alineadas con marcos de referencia como OWASP Top Ten y CWE.

A continuación, se presenta un ejemplo simplificado de la estructura propuesta:

Rol: Actúa como un desarrollador senior con experiencia construyendo aplicaciones robustas, confiables y seguras.

Objetivo del código: Generar una aplicación web en Flask funcional, clara y lista para utilizarse en un entorno real.

Contexto: La aplicación permitirá a los usuarios subir archivos y descargarlos mediante una URL.

Instrucciones de seguridad: Validar archivos y entradas de usuario, prevenir comportamientos inseguros relacionados con rutas y nombres de archivos, controlar el

tamaño de los archivos, manejar errores sin exponer información sensible, implementar controles de acceso cuando corresponda y aplicar buenas prácticas basadas en OWASP y CWE.

Formato de salida: Código funcional acompañado de una explicación breve y clara.

Requerimiento específico: Crear endpoints para subir y descargar archivos.

Los resultados obtenidos durante la investigación sugieren que el uso de estructuras como el PGSG contribuye a reducir la cantidad de vulnerabilidades presentes en el código generado y favorece la incorporación temprana de controles de seguridad. No obstante, esta estrategia debe complementarse con mecanismos de validación independientes, tales como análisis estático (SAST), pruebas dinámicas (DAST) y revisiones manuales, debido a que los modelos de lenguaje continúan presentando limitaciones inherentes a su naturaleza probabilística.

Finalmente, se recomienda que futuras investigaciones evalúen la efectividad del PGSG en diferentes lenguajes de programación, arquitecturas de software y versiones avanzadas de modelos de lenguaje, con el propósito de fortalecer su aplicabilidad dentro de entornos de desarrollo seguro.

Referencias

• Apu Murillo, Y. (2025). *Pentesting Vibe-Coded Web Applications and APIs: How AI-Generated Code Shifts Risk, Evidence, and Testing Strategy*.

• Bouzid, R., & Khoury, R. (2025). Assessing the Effectiveness of ChatGPT in Secure Code Development: A Systematic Literature Review. *ACM Computing Surveys*, *57*(12). https://doi.org/10.1145/3744553

• Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code*. arXiv preprint arXiv:2107.03374.

• Kaniewski, S., Holstein, D., Schmidt, F., & Heer, T. (2024). *Vulnerability handling of AI-generated code: existing solutions and open challenges*. arXiv preprint arXiv:2408.08549.

• Khanmohammadi, K., Roy, P., Khoury, R., Hamou-Lhadj, A., & Konan, W. P. (2024). *WildCode: An Empirical Analysis of Code Generated by ChatGPT*.

• Nie, Y., Wang, Z., Yang, Y., Jiang, R., Tang, Y., Davies, X., ... & Guo, W. (2024). *SECODEPLT: A Unified Benchmark for Evaluating the Security Risks and Capabilities of Code Agents*. arXiv preprint arXiv:2410.11096.

• Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *IEEE Symposium on Security and Privacy*.

• Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants? *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3576915.3623157

• Tihanyi, N., Bisztray, T., Jai, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023). The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. *PROMISE '23*. https://doi.org/10.1145/3617555.3617874

• Yan, H., Vaidya, S. S., Zhang, X., & Yao, Z. (2025). *Guiding AI to Fix Its Own Flaws: An Empirical Study on LLM-Driven Secure Code Generation*.

• Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778v2*