Estudio de vulnerabilidades inducidas en el proceso de optimización de compiladores (Lenguaje C)

1st Dilan Esteban Rey Sepulveda
2190397
Escuela de Ingeniería de sistemas.
Bucaramanga, Santander

2nd David Alexander Vasquez Vivas
2190053
Escuela de Ingeniería de sistemas.
Bucaramanga, Santander

3rd Cristian Leonardo Rueda Quintanilla
2172916
Escuela de Ingeniería de sistemas.
Bucaramanga, Santander

***Abstract*—Este artículo tiene como objetivo analizar las vulnerabilidades introducidas por las optimizaciones del compilador del lenguaje C, sus repercusiones y las soluciones existentes para mitigarlas. Además, se profundizará en una propuesta novedosa: el compilador seguro, una herramienta que busca optimizar el código escrito por el programador sin comprometer la seguridad del sistema.**

***Index Terms*—Compiladores, C, Optimización.**

I. Introducción

En el complejo mundo de la programación, la seguridad del código es un aspecto fundamental. El lenguaje C, por su potencia y versatilidad, se ha convertido en una herramienta indispensable para el desarrollo de sistemas operativos, aplicaciones críticas y una amplia gama de software. Sin embargo, el poder del compilador C, encargado de traducir el código escrito en humano a lenguaje de máquina, puede verse comprometido por la introducción de vulnerabilidades/bugs durante el proceso de optimización, que pueden llegar a causar comportamientos no deseados en nuestros sistema.

Adicional esto salta a la luz un dato no menos importante, en promedio los bugs que son producto de errores en la optimización por parte del compilador GCC son muy tardíos en su proceso de correción (llegando a tardarse hasta al rededor de 11.16 meses en encontrar la solución a uno de estos bugs) [3].

Así pues, se vuelve de suma importancia el obtener un mejor entendimiento del funcionamiento y la naturaleza de las vulnerabilidades/bugs de optimización en los compiladores, estudiando las soluciones existentes y profundizando en una propuesta emergente que llama mucho la atención: el compilador seguro.

II. El problema de las vulnerabilidades

Durante el proceso de optimización, el compilador C puede realizar transformaciones en el código original con el fin de mejorar su rendimiento. Sin embargo, estas optimizaciones pueden generar suposiciones sobre el comportamiento del programa que no siempre son precisas, basándose en la intención del programador rather than on the C standard itself. En ciertos escenarios, estas suposiciones pueden conducir a comportamientos no definidos, creando brechas de seguridad que pueden ser explotadas por atacantes para obtener acceso no autorizado a sistemas o provocar su colapso.

A. Impacto de las vulnerabilidades

Las vulnerabilidades introducidas por el compilador C pueden tener un impacto significativo en la seguridad de los sistemas. Los atacantes pueden aprovechar estas brechas para:

- **Ejecutar código arbitrario:** Un atacante podría introducir código malicioso en el programa, tomando control del sistema y realizando acciones no deseadas.
- **Acceder a datos confidenciales:** Las vulnerabilidades podrían permitir el acceso a información sensible, como contraseñas o datos financieros.
- **Provocar denegación de servicio:** Un atacante podría inundar el sistema con solicitudes maliciosas, lo que lo llevaría a un estado inoperante.

III. El comportamiento indefinido

Las vulnerabilidades por comportamiento indefinido en los lenguajes de programación, particularmente en aquellos basados en C, existen debido a la filosofía de diseño de estos lenguajes. Los diseñadores de C querían crear un lenguaje de bajo nivel eficiente, lo cual permite a los compiladores generar código altamente optimizado para diferentes conjuntos de instrucciones . Sin embargo, esta eficiencia tiene un costo: la introducción de comportamientos indefinidos. [4]

El comportamiento indefinido ocurre cuando el estándar del lenguaje de programación no especifica qué debe suceder en ciertas situaciones. En el contexto de C, esto puede incluir operaciones como el desbordamiento de puntero, el desbordamiento de entero con signo, la 'desreferencia' de puntero nulo, la división por cero, el desplazamiento sobredimensionado y el desbordamiento del búfer. Cuando se utiliza una construcción que el estándar no define, el resultado puede ser impredecible y varía dependiendo del compilador, el sistema operativo o incluso el hardware. [4]

A. Consecuencias del comportamiento indefinido

Una de las consecuencias más graves del comportamiento indefinido es que puede llevar a vulnerabilidades de seguridad en el software. Por ejemplo, consideremos el caso del desbordamiento de puntero. En C, si se suma o resta un entero

de un puntero y el resultado no apunta al mismo objeto o justo después del final del objeto, el comportamiento no está definido según el estándar . Esto significa que el compilador puede asumir que tal situación nunca ocurre y, por lo tanto, eliminar cualquier comprobación de desbordamiento de puntero que haya implementado el programador para proteger el código. Esta eliminación incorrecta de comprobaciones puede introducir vulnerabilidades graves, como el desbordamiento del búfer, que pueden ser explotadas por atacantes para ejecutar código malicioso. [4]

IV. Soluciones existentes

Para abordar las vulnerabilidades del compilador C, se han propuesto diversas soluciones:

- **Análisis estático de código:** Esta técnica examina el código fuente para identificar posibles vulnerabilidades. Sin embargo, puede generar falsos positivos y no tener en cuenta las optimizaciones del compilador.
- **Análisis dinámico de código:** Esta técnica monitorea la ejecución del programa para detectar vulnerabilidades en tiempo real. Si bien es más precisa que el análisis estático, puede ralentizar el programa y no encontrar vulnerabilidades en todas las rutas de ejecución.
- **Pruebas:** Las pruebas exhaustivas pueden ayudar a identificar vulnerabilidades, pero no garantizan su ausencia total.
- **Modificación de las optimizaciones del compilador:** Desactivar ciertas optimizaciones puede prevenir vulnerabilidades, pero a costa de un impacto significativo en el rendimiento.
- **Funciones seguras:** Utilizar alternativas seguras a las funciones vulnerables puede ser útil, pero puede requerir modificaciones en el código fuente.
- **Estándares de codificación segura:** Seguir estándares de codificación segura puede mejorar la seguridad del código, pero exige reescribir el código desde cero de acuerdo con los estándares.

V. Métodos tradicionales: Limitaciones y necesidad de un nuevo enfoque

Los métodos tradicionales de seguridad del software, como las herramientas de análisis dinámico, no siempre son capaces de identificar las vulnerabilidades introducidas por la optimización. Incluso herramientas como Sanitizer (UBSan), que instrumenta el código fuente antes de la optimización para evitar la eliminación de código inestable, pueden tener limitaciones. Deshabilitar las optimizaciones por completo tampoco es una solución viable, ya que puede afectar significativamente al rendimiento del programa compilado.

VI. El compilador seguro: Un enfoque integral a la seguridad del código C

En este contexto, surge el concepto de un compilador seguro. Un compilador seguro se diferencia de los compiladores tradicionales en su enfoque hacia la optimización. Su objetivo principal es preservar el código inestable durante la compilación, evitando así la introducción de vulnerabilidades debidas a la optimización agresiva.

Un compilador seguro ideal cumple con los siguientes requisitos:

- **Preservación de código inestable:** El compilador debe garantizar que el código inestable permanezca intacto durante la optimización, evitando la introducción de vulnerabilidades.
- **Rendimiento aceptable:** La preservación del código inestable no debe introducir una ralentización significativa en el rendimiento del programa compilado.
- **Mínima modificación del código fuente:** El compilador seguro debería funcionar como un reemplazo directo del compilador estándar en el sistema de compilación, evitando la necesidad de modificaciones extensas en el código fuente.

VII. Implementación del compilador seguro

En base a la investigación realizada en el artículo [1] sobre el impacto de la optimización del código C en la seguridad.

Los resultados de la investigación indican que el compilador seguro ofrece una herramienta valiosa para mitigar las vulnerabilidades introducidas por la optimización del código C. El compilador seguro implementado en este estudio ofrece tres niveles de seguridad, cada uno con diferentes balances entre seguridad y rendimiento.

A. Especificación de niveles de seguridad

- **Nivel de seguridad 3:** Permite la construcción de la mayoría de las aplicaciones sin errores. En este nivel, el compilador implementa opciones asociadas con las mejores prácticas adoptadas en grandes proyectos (Linux kernel, Firefox, PostgreSQL). Estas opciones incluyen -fPIE, -fstack-protector-strong, -fno-strict-aliasing, y el uso de funciones fortificadas habilitadas por la opción FORTIFY_SOURCE. Las funciones fortificadas se portaron de la biblioteca Musl y se almacenan en archivos de encabezado como parte del compilador seguro. Además, el diagnóstico -Wclobbered se modificó debido a un alto número de falsos positivos en su versión de GCC. Se espera que el uso del nivel de seguridad 3 resulte en una leve disminución de la velocidad del programa compilado en comparación con un programa compilado con GCC 9.3.0, usando el nivel de optimización -O2.
- **Nivel de seguridad 2:** Agrega banderas de control de optimización y configuraciones de optimización diseñadas específicamente para el compilador seguro, junto con las opciones estándar necesarias. Incluye -fkeep-oversized-shifts y -fpreserve-memory-writes. La primera prohíbe la propagación y el plegado constantes cuando el segundo argumento del operador de desplazamiento es mayor o igual al ancho de su tipo, preservando la operación de desplazamiento en el código ensamblador resultante. La segunda bandera ordena al compilador que preserve todos los efectos secundarios de todas las operaciones de escritura que afectan el área de memoria desde la

cual se realiza al menos una operación de lectura. Esto elimina vulnerabilidades causadas por código inestable. La transición del nivel 3 al 2 no debería aumentar significativamente el número de programas que no pueden ser compilados, aunque el rendimiento puede disminuir.

- **Nivel de seguridad 1:** La diferencia principal es el uso de UBSan [2] con opciones adicionales como -fsanitize=function, -fsanitize=null, y -fsanitize=return. Este nivel también utiliza una versión modificada de ASLR que permite la aleatorización de ubicaciones de funciones en un archivo de programa y el orden de las variables locales en el marco de una función. Agrega verificaciones estrictas, lo que hace imposible construir muchas aplicaciones sin modificar su código fuente, sacrificando rendimiento (reducción de velocidad hasta 2 a 3 veces) y la capacidad de construir aplicaciones insuficientemente confiables. Este nivel está diseñado para casos donde la seguridad tiene la máxima prioridad.

VIII. Metodología

A. Proceder

En la investigación [1] se generó un conjunto de pruebas con fragmentos de código para 203 casos especificos listados en el anexo J del estándar C11 de dicha investigación con el fin de estudiar el impacto en el rendimiento de cada uno de los niveles del compilador seguro. Los resultados de dichas pruebas se analizarán en la siguiente sección.

IX. Experimentos

A. Impacto en el rendimiento: Un análisis detallado por tarea

El impacto del compilador seguro en el rendimiento varía según el tipo de tarea. Las tareas intensivas en memoria o en codificación de video pueden verse más afectadas por las optimizaciones del compilador. En general, el nivel 3 del compilador seguro ofrece un equilibrio entre seguridad y rendimiento, con una ralentización máxima del 20% en las tareas evaluadas. Los niveles 2 y 1 ofrecen un mayor nivel de seguridad, pero a costa de una mayor ralentización, especialmente en tareas específicas como la codificación de vídeo. [1]

Scenario	Baseline (s)	Safe3 (s)	Safe3 slowed	Safe2 (s)	Safe2 slowed	Safe1 (s)	Safe1 slowed
GNU Go	4.67	4.85	3.85%	5.23	11.99%	9.32	99.57%
LAME	3.45	3.55	2.89%	3.55	2.83%	9.89	186.31%
fannkuch	2.03	2.42	19.21%	2.42	19.21%	3.51	72.91%
x264	7.91	7.90	−0.23%	8.06	1.78%	18.30	131.21%
zlib	2.25	2.29	2.00%	2.28	1.33%	2.80	24.44%

Fig. 1: Impacto en rendimiento en los niveles del compilador seguro. Tomado de [1]

B. Resultados

- En 17 de los 203 tipos de comportamiento indefinido, los compiladores usaron este conocimiento para optimizar el código.
- Para 9 tipos, aunque podrían afectar la optimización, no se encontró evidencia de que los compiladores utilizan esta posibilidad.
- Los otros tipos de comportamiento indefinido fueron detectados como advertencias o errores, o compilados en código con semántica predecible.

Fig. 2: Resultados set de pruebas.

C. Recomendaciones para una selección adecuada del nivel de seguridad

La elección del nivel de seguridad adecuado para el compilador seguro debe realizarse en función del equilibrio entre seguridad y rendimiento, considerando las características específicas de la aplicación y el contexto de desarrollo:

- **Evaluación previa:** Se recomienda realizar una evaluación previa del impacto del compilador seguro en el rendimiento de la aplicación específica antes de adoptar un nivel de seguridad definitivo.
- **Monitoreo del rendimiento:** Es importante monitorear el rendimiento de la aplicación después de implementar el compilador seguro para detectar posibles problemas de rendimiento y ajustar el nivel de seguridad si es necesario.
- **Comunicación con el equipo de desarrollo:** Si se observan ralentizaciones significativas, es recomendable comunicarse con el equipo de desarrollo para analizar posibles optimizaciones del código fuente o ajustes en la configuración del compilador.

X. Conclusiones

A raíz de los resultados analizados se llegó a las siguientes conclusiones:

- El comportamiento indefinido en lenguajes como C es un compromiso entre eficiencia y seguridad. Si bien permite a los compiladores generar código altamente optimizado, también introduce riesgos significativos si no se maneja adecuadamente. Los desarrolladores deben ser conscientes de estas potenciales vulnerabilidades y utilizar herramientas y prácticas que mitiguen los riesgos asociados con el comportamiento indefinido.
- El compilador seguro ofrece una herramienta valiosa para mitigar las vulnerabilidades introducidas por la optimización del código C. Sin embargo, es importante considerar el impacto en el rendimiento al seleccionar el nivel de seguridad adecuado. La elección del nivel ideal dependerá de las prioridades del proyecto, considerando la necesidad de seguridad y el impacto en el rendimiento.

References

[1] Baev, R.V., Skvortsov, L.V., Kudryashov, E.A. et al. Preventing Vulnerabilities Caused by Optimization of Code with Undefined Behavior. Program Comput Soft 48, 445–454 (2022). [En línea]. Disponible: https://doi-org.bibliotecavirtual.uis.edu.co/10.1134/S0361768822070027

[2] UndefinedBehaviorSanitizer — Clang 19.0.0git documentation. (s. f.).[En línea]. Disponible: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[3] Zhide Zhou, Zhilei Ren, Guojun Gao, He Jiang, An empirical study of optimization bugs in GCC and LLVM, Journal of Systems and Software, Volume 174, 2021, 110884, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110884. [En línea]. Disponible: https://doi.org/10.1016/j.jss.2020.110884.

[4] Evaluación de vulnerabilidades inducidas por el compilador Michael J. Hohnka , Jodi A. Miller , Kenrick M. Dacumos , Timothy J. Fritton , Julia D. Erdley y Lyle N. Long. Revista de sistemas de información aeroespacial 2019 16:10 , 409-426. [En línea]. Disponible: https://arc.aiaa.org/doi/abs/10.2514/1.I010699