El caso de las hojas de ruta seguras para la memoria

Por qué tanto los ejecutivos de alto nivel como los expertos técnicos deben tomarse en serio la codificación segura para la memoria

Publicación: diciembre de 2023

Agencia de Ciberseguridad y Seguridad de Infraestructura de Estados Unidos (United States Cybersecurity and Infrastructure Security Agency)
Agencia de Seguridad Nacional de Estados Unidos (United States National Security Agency)
Oficina Federal de Investigaciones de Estados Unidos (United States Federal Bureau of Investigation)
Centro Australiano de Ciberseguridad (Australian Cyber Security Centre) de la Dirección de Señales de Australia (Australian Signals Directorate)
Centro Canadiense de Ciberseguridad (Canadian Centre for Cyber Security)
Centro Nacional de Ciberseguridad del Reino Unido (United Kingdom National Cyber Security Centre)
Centro Nacional de Ciberseguridad de Nueva Zelanda (New Zealand National Cyber Security Centre)
Equipo de Respuesta a Emergencias Informáticas de Nueva Zelanda (Computer Emergency Response Team New Zealand)

Resumen ejecutivo

Las vulnerabilidades de seguridad de la memoria son el tipo más prevalente de vulnerabilidad de software divulgada.[1,2,3] Son una clase de errores de codificación comunes y conocidos que los agentes maliciosos explotan de forma rutinaria. Estas vulnerabilidades representan un problema importante para la industria del software, ya que hacen que los fabricantes publiquen continuamente actualizaciones de seguridad y que sus clientes apliquen correcciones con frecuencia. Estas vulnerabilidades persisten a pesar de que los fabricantes de software históricamente han gastado importantes recursos en intentar reducir su prevalencia e impacto a través de diversos métodos, incluido el análisis, la aplicación de correcciones, la publicación de código nuevo y la inversión en programas de capacitación para desarrolladores. Las organizaciones de clientes gastan importantes recursos en responder a estas vulnerabilidades a través de onerosos programas de administración de correcciones y actividades de respuesta a incidentes.

Los lenguajes de programación seguros para la memoria (MSL, por sus siglas en inglés) pueden eliminar las vulnerabilidades de seguridad de la memoria. Por lo tanto, la transición a los MSL probablemente disminuiría, en gran medida, la necesidad de invertir en actividades destinadas a reducir estas vulnerabilidades o minimizar su impacto. Además, las inversiones para migrar códigos base inseguros a MSL generarían beneficios a largo plazo en forma de productos más seguros, lo que sufragaría parte del costo inicial de la transición a los MSL.

La Agencia de Ciberseguridad y Seguridad de Infraestructura de EE. UU. (CISA, por sus siglas en inglés), la Agencia de Seguridad Nacional (NSA, por sus siglas en inglés), la Oficina Federal de Investigaciones (FBI, por sus siglas en inglés) y las autoridades de ciberseguridad de Australia, Canadá, el Reino Unido y Nueva Zelanda* (en lo sucesivo, denominadas "las agencias autoras") desarrollaron en conjunto esta orientación como parte de nuestra campaña colectiva Seguridad desde el Diseño (Secure by Design). Con esta orientación, las agencias autoras instan a los altos ejecutivos de todos los fabricantes de software a reducir el riesgo para los clientes priorizando las prácticas de diseño y desarrollo que implementan MSL. Además, las agencias instan a los fabricantes de software a crear y publicar hojas de ruta seguras para la memoria que detallen cómo eliminarán las vulnerabilidades de seguridad de la memoria en sus productos. Mediante la publicación de hojas de ruta seguras para la memoria, los fabricantes indicarán a los clientes que se están responsabilizando de los resultados de seguridad, adoptando una transparencia radical e implementando un enfoque vertical para desarrollar productos seguros, principios clave de Seguridad desde el Diseño.

> La campaña Seguridad desde el Diseño insta a los proveedores de tecnología a responsabilizarse de los resultados de seguridad de sus clientes incorporando la ciberseguridad en el diseño y el desarrollo. Consulte Seguridad desde el diseño, Asegurar sus productos y Cambio del equilibrio de los riesgos de ciberseguridad: principios y enfoques para la seguridad desde el diseño y por defecto.

Esta orientación proporciona pasos que les permiten a los fabricantes crear hojas de ruta seguras para la memoria e implementar cambios a fin de eliminar las vulnerabilidades de seguridad de la memoria de sus productos. La eliminación de esta clase de vulnerabilidad debería considerarse un imperativo empresarial que probablemente requiera la participación de muchos departamentos. Las agencias autoras instan a los ejecutivos a liderar desde los niveles superiores mediante la identificación pública del personal sénior que impulsará la publicación de su hoja de ruta y ayudará a realinear los recursos según sea necesario.

* El Centro Australiano de Ciberseguridad (ACSC, por sus siglas en inglés) de la Dirección de Señales de Australia (ASD, por sus siglas en inglés), el Centro Canadiense de Ciberseguridad (CCCS, por sus siglas en inglés), el Centro Nacional de Ciberseguridad del Reino Unido (NCSC-UK, por sus siglas en inglés), el Centro Nacional de Ciberseguridad de Nueva Zelanda (NCSC-NZ, por sus siglas en inglés) y el Equipo de Respuesta a Emergencias Informáticas de Nueva Zelanda (CERT NZ, por sus siglas en inglés).

Índice

Introducción

Las vulnerabilidades de seguridad de la memoria [CWE-1399. Categorización integral: seguridad de la memoria] son una clase de vulnerabilidad que afecta la forma en que se puede escribir, asignar o desasignar la memoria, así como acceder a esta, de formas no intencionadas en lenguajes de programación.[4,5,6]

El concepto subyacente a estos errores puede entenderse mediante la metáfora de que el software puede solicitar el elemento número 11 o el número -1 de una lista de solo 10 elementos. A menos que el desarrollador o el lenguaje impidan este tipo de solicitudes, el sistema podría devolver datos de alguna otra lista de elementos.

Según el tipo de vulnerabilidad, un agente malicioso podría acceder de forma ilícita a datos o dañarlos, o ejecutar código malicioso arbitrario. Por ejemplo, un agente malicioso puede enviar una carga útil cuidadosamente diseñada a una aplicación que daña la memoria de la aplicación y, luego, provoca que ejecute malware. O bien, un agente malicioso puede enviar un archivo de imagen con formato incorrecto que incluya malware para crear un shell interactivo en el sistema de la víctima. Si un agente puede ejecutar código arbitrario de esta manera, puede obtener el control de la cuenta que ejecuta el software.

Los informes modernos de la industria indican que los defectos que se identificaron por primera vez hace varias décadas siguen siendo vulnerabilidades comunes que explotan los agentes maliciosos en la actualidad para poner en riesgo las aplicaciones y los sistemas de manera rutinaria.[7] Sin embargo, de acuerdo con los informes modernos de la industria, estas vulnerabilidades siguen siendo comunes, y los agentes maliciosos las explotan rutinariamente para poner en riesgo las aplicaciones y los sistemas:

- Alrededor del 70 % de las vulnerabilidades y exposiciones comunes (CVE, por sus siglas en inglés) de Microsoft son vulnerabilidades de seguridad de la memoria (según las CVE de 2006-2018).[8]
- Alrededor del 70 % de las vulnerabilidades identificadas en el proyecto Chromium de Google son vulnerabilidades de seguridad de la memoria.[9]
- En un análisis de las vulnerabilidades de Mozilla, 32 de 34 errores importantes/graves fueron vulnerabilidades de seguridad de la memoria.[10]
- Según el análisis del equipo Proyecto Cero (Project Zero) de Google, el 67 % de las vulnerabilidades del día cero en 2021 fueron vulnerabilidades de seguridad de la memoria.[11]

Medidas de mitigación

Durante las últimas décadas, los desarrolladores de software han buscado continuamente abordar la prevalencia y el impacto de las vulnerabilidades de seguridad de la memoria en el ciclo de vida de desarrollo de software (SDLC, por sus siglas en inglés) a través de los siguientes métodos de mitigación. A pesar de estos esfuerzos continuos, la seguridad de la memoria sigue siendo una de las principales causas de vulnerabilidades divulgadas en productos de software. Sin embargo, estas medidas de mitigación siguen siendo valiosas, especialmente cuando se utilizan en combinación, para proteger el código que aún no ha hecho la transición al MSL o que no puede hacerlo.

Medidas de mitigación para reducir la prevalencia

Capacitación para desarrolladores

Ciertos lenguajes de programación, como C y C++, son ejemplos de lenguajes de programación que pueden producir un código no seguro para la memoria; sin embargo, todavía se encuentran entre los lenguajes más utilizados en la actualidad. En un intento por mitigar los peligros del código no seguro para la memoria en C y C++, muchos fabricantes de software invierten en programas de capacitación para sus desarrolladores. Muchos de estos programas incluyen tácticas diseñadas para reducir la prevalencia de vulnerabilidades inseguras para la memoria que producen esos lenguajes. Además, existen numerosos programas de capacitación de asociaciones comerciales e industriales. Así mismo, diversas organizaciones y universidades ofrecen capacitaciones y un certificado profesional por demostrar conocimientos de prácticas de codificación segura en C y C++.

Si bien la capacitación puede reducir la cantidad de vulnerabilidades que puede introducir un codificador, debido a lo generalizados que son los defectos de seguridad de la memoria, es casi inevitable que aún se produzcan vulnerabilidades de seguridad de la memoria. Incluso los desarrolladores más experimentados escriben errores que pueden introducir vulnerabilidades importantes. La capacitación debe ser un puente mientras una organización implementa controles técnicos más sólidos, como lenguajes seguros para la memoria.

Cobertura de código

La cobertura de código es el proceso de cubrir la mayor cantidad posible de código base con pruebas unitarias y de integración. Las prácticas industriales recomiendan a los equipos de desarrollo esforzarse por lograr una cobertura del 80 % o más, pero esto no siempre es posible debido a las limitaciones de tiempo y recursos. Los equipos de desarrollo tienen como objetivo cubrir todas las áreas fundamentales y que conlleven riesgos de seguridad de una aplicación con casos de prueba tanto positivos como negativos. Los equipos pueden agregar fácilmente este tipo de pruebas a un script o proceso de automatización para realizar pruebas de repetibilidad y regresión. Los beneficios radican en garantizar que no se agreguen nuevas vulnerabilidades a la funcionalidad que se probó anteriormente, pero que puede haberse modificado de forma involuntaria como parte de una actualización y, por lo tanto, no estaba en un plan de prueba de lanzamiento.

Pautas de codificación segura

Las organizaciones y la industria han desarrollado muchas pautas de codificación segura para los lenguajes de programación más prevalentes. Estas pautas describen las áreas en las que los desarrolladores deben tener más cuidado debido a las interrupciones específicas del lenguaje, especialmente en torno al manejo de la memoria. Las organizaciones han intentado garantizar que los equipos de desarrollo no solo utilicen una guía de codificación segura para el lenguaje de programación que elijan, sino que también actualicen de manera activa la guía a medida que el equipo identifica nuevos problemas o estandariza el enfoque de un problema común.

Pruebas de exploración de vulnerabilidades mediante datos aleatorios (fuzzing)

Las pruebas de exploración de vulnerabilidades mediante datos aleatorios prueban el software enviándole una amplia variedad de datos, incluidos los datos no válidos o aleatorios, y detectan cuando los datos de prueba hacen que la aplicación se bloquee o falle en las aserciones de código.[12] Las pruebas de exploración de vulnerabilidades mediante datos aleatorios son un método común para encontrar ciertos errores, como la saturación del búfer. Las pruebas de exploración de

vulnerabilidades mediante datos aleatorios pueden ayudar a descubrir vulnerabilidades, pero ninguna herramienta puede detectar todas las vulnerabilidades. Dado que las pruebas de exploración de vulnerabilidades mediante datos aleatorios representan una táctica no determinista que se aplica después de cometer los errores iniciales de codificación, habrá límites en cuanto a su efectividad. De manera continua, se crean nuevos métodos de pruebas de exploración de vulnerabilidades mediante datos aleatorios que detectan vulnerabilidades no descubiertas anteriormente. Los fabricantes de software deben asegurarse de que sus estrategias de pruebas de exploración de vulnerabilidades mediante datos aleatorios se actualicen continuamente.

Pruebas de seguridad de aplicaciones estáticas y dinámicas

Los desarrolladores utilizan herramientas de pruebas de seguridad de aplicaciones estáticas (SAST, por sus siglas en inglés) y pruebas de seguridad de aplicaciones dinámicas (DAST, por sus siglas en inglés) para encontrar una variedad de vulnerabilidades de software, incluidos los errores relacionados con la memoria.[13] Las herramientas de SAST analizan recursos estáticos, específicamente los binarios o el código fuente, y las herramientas de DAST examinan un sistema en ejecución (o el conjunto de pruebas unitarias, que puede tener una efectividad similar) para encontrar problemas que serían difíciles de detectar con una herramienta de SAST. Muchas organizaciones utilizan ambos tipos de herramientas. Algunas organizaciones más grandes utilizan más de una herramienta de SAST o DAST de diferentes proveedores para brindar cobertura adicional utilizando una variedad más amplia de enfoques.

Según el código base, las herramientas de SAST y, en menor medida, las herramientas de DAST pueden generar una cantidad significativa de falsos positivos, lo que crea una carga para los desarrolladores de software. Además, ninguna herramienta de SAST o DAST puede detectar todas las vulnerabilidades.

Subconjuntos de lenguajes más seguros

La comunidad de C++ ha estado contemplando[14] el equilibrio entre la compatibilidad con versiones anteriores, los valores predeterminados de seguridad de la memoria y otras prioridades para el lenguaje base.[15] Existen múltiples esfuerzos dirigidos para hacer que C y C++ sean menos vulnerables para los códigos base y los productos existentes. Por ejemplo, Apple ha modificado la cadena de herramientas del compilador de C utilizada en el sistema iBoot [16] para mitigar los problemas de seguridad de tipos y de la memoria. El análisis externo[17] indica que puede haber costos no triviales de desempeño y uso de memoria. Microsoft ha desarrollado “Checked C”, que “agrega una verificación estática y dinámica a C para detectar o evitar los errores comunes de programación, como la saturación del búfer y los accesos a la memoria fuera del límite”.[18] Hay esfuerzos más generales para mejorar la seguridad de la memoria de C++ para el código existente,[19] incluidos ciertos esfuerzos, como Carbon.[20,21]

Medidas de mitigación para reducir el impacto

Memoria no ejecutable

La mayoría de las arquitecturas informáticas modernas no contienen memoria separada para datos y códigos, lo que permite a los agentes maliciosos que explotan los problemas de seguridad de la memoria introducir código como datos que, luego, el procesador podría verse obligado a ejecutar. Uno de los primeros intentos de mitigar los problemas de seguridad de la memoria fue marcar algunos segmentos de la memoria como no ejecutables. En tales casos, una unidad central de procesamiento (CPU, por sus siglas en inglés) no ejecutaría las instrucciones contenidas en dichas páginas, ya que solo estaban destinadas a almacenar datos, no código. Desafortunadamente, han

surgido técnicas más sofisticadas, como la programación orientada al retorno (ROP, por sus siglas en inglés), que permite reutilizar segmentos de código existentes dentro de un programa para ejecutarlos en datos controlados por el adversario a fin de subvertir el control de un programa.[22]

Integridad del flujo de control

La tecnología de integridad del flujo de control (CFI, por sus siglas en inglés) identifica todas las ramas indirectas y agrega una verificación en cada una.[23] Durante el tiempo de ejecución, el programa detectará las ramas no válidas, lo que causará que el sistema operativo finalice el proceso. A pesar de algunos éxitos, se han descubierto numerosas elusiones de la CFI,[24] incluidas aquellas que se han explotado en la naturaleza.[25]

Aleatoriedad en la disposición del espacio de direcciones (ASLR, por sus siglas en inglés)

Tradicionalmente, los agentes cibernéticos maliciosos que detectan una vulnerabilidad de la memoria elaborarán una carga útil para explotar esa vulnerabilidad e intentarán encontrar una forma de ejecutar su código. Encontrar la disposición exacta de la memoria para ejecutar su código puede requerir algo de experimentación. Sin embargo, cuando la encuentren, la vulnerabilidad funcionará en cualquier instancia de la aplicación.

La ASLR es una técnica en la que el sistema en tiempo de ejecución traslada diversos componentes, como la pila y el montón, a diferentes direcciones virtuales cada vez que se ejecuta el programa. La ASLR tiene como objetivo garantizar que los agentes cibernéticos maliciosos no sepan cómo está organizada la memoria, lo que hace que sea mucho más difícil explotar la vulnerabilidad. Sin embargo, las elusiones de la ASLR son comunes porque se puede persuadir a los programas para que filtren direcciones de la memoria,[26,27,28,29] lo que significa que la ASLR no evita por completo la explotación de las vulnerabilidades de seguridad de la memoria.

Otras medidas de mitigación del compilador

Los compiladores modernos incluyen varias medidas de mitigación contra la explotación de problemas de seguridad de la memoria. Ciertas técnicas, como los valores controlados de pila y las pilas no grabables, aprovechan diferentes enfoques para mitigar algunos problemas de seguridad de la memoria. Sin embargo, los agentes también han identificado técnicas en el lado de las vulnerabilidades para eludir estas medidas de mitigación, como identificar filtraciones de datos y ROP.

Espacios aislados

Los equipos de desarrolladores pueden utilizar los espacios aislados para aislar diferentes partes de un sistema y limitar el alcance de cualquier vulnerabilidad potencial. Los desarrolladores dividirán la aplicación en subsistemas y restringirán los recursos que pueden usar, incluida la memoria, el acceso a la red y el control de procesos. Los espacios aislados brindan una capa de protección para muchas clases de vulnerabilidad; incluso, vuelven a la utilidad chroot para evitar recorridos del sistema de archivos.

Un subsistema que maneja datos no confiables, como comunicaciones de red o contenido generado por el usuario, puede ser un buen candidato para aislarlo de otras partes del sistema mediante un espacio aislado. Si los agentes maliciosos detectan una vulnerabilidad relacionada con la memoria en un subsistema, se enfrentan a la tarea adicional de salir del espacio aislado. Obligar a los adversarios a encontrar múltiples defectos nuevos aumenta el costo del ataque.

A pesar del valor que aportan los espacios aislados, los desarrolladores pueden verse limitados para impulsar este modelo. Mientras más espacios aislados utilizan, más complejo se vuelve el código. Además, existen límites prácticos en cuanto a la cantidad de espacios aislados que puede tolerar un sistema, especialmente en dispositivos restringidos, como los teléfonos. Asimismo, suelen descubrirse elusiones del espacio aislado, también conocidas como fugas del espacio aislado, lo que frustra las protecciones de seguridad. La presentación de Google sobre los espacios aislados[30] en el sistema operativo Android demuestra los límites asociados con esta táctica de mitigación.

Refuerzo de los asignadores de memoria

Al igual que en el caso de las medidas de mitigación del compilador y de la ASLR, el refuerzo de los asignadores hace que aprovechar una vulnerabilidad sea más difícil, pero no quita la vulnerabilidad de seguridad de la memoria. Por ejemplo, Apple informó que su asignador, denominado "kalloc_type", "hace que la explotación de la mayoría de las vulnerabilidades de daño de la memoria sea inherentemente poco confiable". También existen asignadores comerciales seguros para la memoria que se dirigen a dominios específicos, como los dispositivos de tecnología operativa (OT, por sus siglas en inglés).

Posibles medidas de mitigación futuras: uso de hardware

Un área prometedora en desarrollo activo implica usar hardware para apoyar las protecciones de la memoria. El proyecto Instrucciones RISC Mejoradas de Hardware de Capacidad (CHERI, por sus siglas en inglés)[31] es un proyecto de investigación conjunto de SRI International y la Universidad de Cambridge (University of Cambridge) que agrega nuevas características a las arquitecturas de chip existentes. El programa Seguridad Digital desde el Diseño (DSBD, por sus siglas en inglés) del Gobierno del Reino Unido recolectó £70,000,000 de financiación gubernamental con £117,000,000 de coinversión de la industria para seguir desarrollando la tecnología.[32] Además de una variedad de actividades de investigación y desarrollo respaldadas por la industria y el ámbito académico, el programa permitió a Arm enviar su prototipo Morello de procesador, sistema en chip (SoC, por sus siglas en inglés) y placa habilitado para CHERI en enero de 2022. Tanto Arm como Microsoft han documentado sus esfuerzos de CHERI y una variedad de otras actividades que respaldó el programa DSBD. Ahora existe una comunidad de desarrolladores que crean herramientas y bibliotecas para permitir la adopción generalizada de la tecnología.

CHERI se puede implementar en arquitecturas distintas de las de Arm; el programa DSBD también anunció recientemente una inversión de £1,200,000 en un proyecto de demostración con la arquitectura RISC-V.[33] El objetivo es mostrar cómo la tecnología puede implementarse de forma beneficiosa en sistemas automotrices, en los que la seguridad es fundamental.

Arm introdujo otra tecnología denominada extensión de etiquetado de memoria (MTE, por sus siglas en inglés) en algunas de sus líneas de productos de CPU para detectar errores de tipo uso después de la liberación y fuera del límite (también conocidos como saturación del búfer).[34] Cuando se asigna memoria, el sistema le asigna una etiqueta. Todo el acceso posterior a esa memoria debe realizarse con esa etiqueta. La CPU generará un error si las etiquetas no coinciden. Arm estima que la sobrecarga de la MTE está entre el 1 y el 2 %. Los dispositivos móviles pueden experimentar las primeras implementaciones generalizadas.[35] Intel también anunció capacidades de etiquetado de memoria en futuros conjuntos de chips. [36]

Otra investigación basada en hardware incluye FineIBT, que cuenta con compatibilidad con la integridad del flujo de control (CFI), además de la tecnología de aplicación del flujo de control (CET, por sus siglas en inglés) basada en hardware de Intel.[37]

Algunas características de hardware, como la MTE, serán necesarias incluso en sistemas escritos en MSL. Por ejemplo, los programadores de Rust pueden marcar cierto código como “inseguro”, beneficiándose de los controles de hardware.

Aunque las protecciones de memoria en el hardware aún no están ampliamente disponibles, algunos observadores de la industria consideran que serán útiles en muchos escenarios de implementación en los que la migración a los MSL llevará un tiempo prolongado. En tales escenarios, los ciclos de actualización del hardware pueden ser lo suficientemente cortos como a la hora de proporcionar protecciones de memoria importantes para los clientes hasta que haya otras protecciones disponibles. Se están realizando experimentos con estas protecciones de hardware, lo que incluye el trabajo para medir el impacto en el desempeño en el mundo real y las características de consumo de memoria de estos nuevos diseños. En algunos casos, es posible que las protecciones de hardware permitan un mayor desempeño si el software las utiliza de manera óptima.

El caso de los lenguajes seguros para la memoria

A pesar de que los fabricantes de software invierten grandes recursos en intentar mitigar las vulnerabilidades de seguridad de la memoria, estas siguen siendo generalizadas. Por lo tanto, los clientes deben gastar importantes recursos en responder a estas vulnerabilidades a través de onerosos programas de administración de correcciones y actividades de respuesta a incidentes.

Como señaló anteriormente la NSA en la Hoja de información sobre ciberseguridad relacionada con la seguridad de la memoria del software y en otras publicaciones,[38] la medida de mitigación más prometedora es que los fabricantes de software utilicen un lenguaje de programación seguro para la memoria, ya que se trata de un lenguaje de codificación que no es susceptible a las vulnerabilidades de seguridad de la memoria. Sin embargo, los lenguajes de programación inseguros para la memoria, como C y C++, se encuentran entre los lenguajes de programación más comunes.[39] Las aplicaciones y los dispositivos de Internet en todo el panorama tecnológico utilizan lenguajes de programación inseguros para la memoria. Estos lenguajes ejecutan sistemas operativos, sistemas con recursos limitados y aplicaciones que requieren un alto desempeño. La generalización de los lenguajes inseguros para la memoria significa que, actualmente, existe un riesgo significativo en las funciones informáticas más fundamentales.

Al mismo tiempo, las agencias autoras reconocen la realidad comercial de que la transición a MSL implicará importantes inversiones y atención ejecutiva. Además, cualquier transición de este tipo requerirá una planificación cuidadosa a lo largo de varios años. Aunque existe un costo inicial en la migración de códigos base a los MSL, estas inversiones mejorarán la calidad y la confiabilidad del producto, y, fundamentalmente, la seguridad del cliente.

Los fabricantes de software se beneficiarán del uso de MSL, y sus clientes se beneficiarán de productos más seguros. Entre los beneficios para clientes y desarrolladores, se pueden incluir los siguientes:

- **Mayor confiabilidad.** Los MSL crean un código más confiable que los lenguajes de programación inseguros para la memoria. Por ejemplo, un sistema operativo que utiliza un lenguaje de programación inseguro para la memoria solo puede bloquear la aplicación si detecta una vulneración de la memoria. Si la aplicación es un proceso entre el cliente y el servidor, podría interrumpir las conexiones de todos los usuarios conectados. Además, el sistema operativo puede bloquearse si el daño de la memoria se produce en el núcleo. Por otro lado, los MSL prohíben que se generen vulneraciones de la memoria.

- **Menos interrupciones para los desarrolladores.** Cuando alguien informa un error de seguridad de la memoria, los desarrolladores adoptan el modo reactivo y deben detener otros trabajos para diagnosticar y mitigar el problema. Menos defectos de seguridad de la memoria pueden implicar menos respuestas imprevistas y, a menudo, urgentes a los descubrimientos de vulnerabilidades. Los equipos de desarrollo han informado que los errores de seguridad de la memoria son algunos de los más difíciles de diagnosticar y abordar correctamente. En consecuencia, a menudo deben retirar a los desarrolladores más experimentados de otros trabajos importantes para detectar y corregir de manera adecuada estos defectos. La transición a los MSL permitiría a los desarrolladores centrarse en las prioridades de trabajo actuales, en lugar de reaccionar ante vulnerabilidades recién descubiertas.
- **Menos emergencias para el personal de apoyo.** Una menor cantidad de vulnerabilidades de seguridad de la memoria puede generar menos versiones de emergencia, lo que ahorra tiempo a ciertos equipos, como los del área de Construcción, Control de Calidad, Administración de Productos y Asistencia.
- **Menos emergencias (y vulneraciones) para los clientes.** Quitar la clase de vulnerabilidad de seguridad de la memoria de un producto mediante la transición a un MSL elimina la necesidad de lanzar versiones de seguridad para problemas de la memoria. Esto reducirá la cantidad de lanzamientos urgentes de productos que un cliente deberá admitir, lo que ahorrará tiempo y evitará vulneraciones.

Además de aportar beneficios para los fabricantes de software y sus clientes, los MSL reducen la superficie de ataque de un producto. Esa reducción en la superficie de ataque aumentará el costo para los agentes maliciosos, que, luego, deberán invertir más recursos para descubrir otras vulnerabilidades explotables. Como ocurre con cualquier otra medida de mitigación, la transición a los MSL no detendrá ni impedirá por sí sola la ciberdelincuencia o el espionaje. La rentabilidad de los agentes cibernéticos maliciosos determinará dónde el agente buscará el próximo vector de intrusión para cumplir su misión. Sin embargo, debido a la generalización actual de las explotaciones y vulnerabilidades de seguridad de la memoria, reducir y, con el tiempo, eliminar esa vía de ataque aumentará significativamente el costo de un ataque.

Planificación de la transición a lenguajes seguros para la memoria

Consideraciones

Los fabricantes de software deben considerar los siguientes factores técnicos y no técnicos al desarrollar su hoja de ruta:

1. **Orientación para la priorización.** Los fabricantes deben considerar cómo priorizar la migración a los MSL mediante la elaboración de hojas de ruta y orientación específica para los equipos técnicos y de desarrollo.
2. **Elección de MSL adecuados para el caso de uso.** Existen numerosos MSL, y cada uno tiene su propio conjunto de compensaciones en lo que respecta a la arquitectura, las herramientas, el desempeño, la popularidad, el costo y otros factores. Ningún MSL es adecuado para todas las necesidades de programación. Los fabricantes deben analizar los casos de uso que utilizan lenguajes inseguros para la memoria y elegir el MSL más adecuado para cada uno. Al seleccionar un MSL, los fabricantes de software deben seguir procesos estándar de administración de riesgos, ya que los MSL no están libres de otras vulnerabilidades potenciales de gravedad crítica. Como parte de su programa de administración de riesgos, los fabricantes deben seguir atentamente la cadena de

suministro y las prácticas seguras del ciclo de vida de desarrollo tal como se definen en el Marco de administración de riesgos y el Marco de desarrollo de software seguro (SSDF, por sus siglas en inglés) del Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés).

3. **Capacidades y recursos del personal.** Los fabricantes deben considerar cómo capacitarán a los desarrolladores en un MSL seleccionado, cómo pueden priorizar la contratación de desarrolladores con las habilidades relevantes y qué recursos pueden necesitar para apoyar el lenguaje seleccionado.

Orientación para la priorización

Si bien no existe una única manera de priorizar una transición a los MSL, la siguiente lista de opciones ayudará a los equipos de desarrollo a elegir proyectos de migración del tamaño adecuado que les brindarán experiencia, un bucle de retroalimentación estrecho y una cantidad de riesgos controlable.

- **Comience con proyectos nuevos y más pequeños.** Reescribir el código existente en un MSL puede ser un desafío importante, especialmente si el código ya funciona bien y la organización aún no tiene experiencia en el MSL elegido. Considere comenzar con proyectos nuevos y más pequeños que conlleven un menor riesgo para darles a los equipos tiempo para experimentar con nuevas herramientas y procesos.
- **Reemplace los componentes inseguros de la memoria.** Considere tomar un componente autónomo escrito en un lenguaje de programación inseguro para la memoria y reescribirlo en el MSL elegido. Considere formas de ejecutar el componente existente de forma simultánea con el componente en MSL recién actualizado y compare los resultados. Una vez que el componente en MSL actualizado produzca constantemente el mismo resultado que el componente anterior, debería ser posible retirar el componente anterior.
- **Priorice el código fundamental para la seguridad.** Partes de su código base pueden ser sensibles desde una perspectiva de seguridad o a lo largo de una vía de ataque fundamental.[40] Entre los ejemplos, se incluye el código que realiza operaciones en contenido generado por el usuario, que es un infame vector de mal uso. Entre otros ejemplos, se incluye el código que maneja claves secretas, abre conexiones de red, realiza autenticaciones y autorizaciones u opera en niveles bajos, como el firmware.[41] Priorice el código que conlleve riesgos para la seguridad cuando el equipo tenga la experiencia necesaria en MSL. Revise el uso de componentes criptográficos y otras “raíces de confianza”, sobre las cuales se construye el sistema durante el proceso de priorización.
- **Utilice instrumentación.** Por ejemplo, la herramienta de asignación GWP-ASan del sistema operativo Android puede detectar errores de memoria dinámica que las pruebas de exploración de vulnerabilidades mediante datos aleatorios no detectan.
- **Evalúe el desempeño y la complejidad.** A menudo, existen motivos para reescribir componentes o sistemas más grandes para incorporar nuevos requisitos no previstos en el diseño original. La implementación existente de un sistema puede haberse vuelto frágil y, por lo tanto, al equipo de desarrollo puede resultarle difícil apoyar la evolución necesaria para cumplir con los nuevos requisitos. Si el equipo reescribirá el sistema por cualquier motivo, debería considerar dividir los requisitos en partes más pequeñas y escribir partes o la totalidad en el MSL que eligió la organización.
- **Determine qué módulos dependen de la CPU.** Algunas aplicaciones dependen de la CPU, lo que significa que la velocidad de la CPU limita el desempeño del sistema en general. Específicamente, en el caso de los MSL que recogen elementos no utilizados, una aplicación

que depende de la CPU puede experimentar más fluctuaciones de desempeño que una que se ve limitada por otras cuestiones, como el tiempo de respuesta humana, la latencia de red o la entrada y salida (I/O, por sus siglas en inglés) del disco.

- **Intensifique los sistemas paralelos.** Si un equipo de desarrollo transfiere una aplicación altamente paralela a un MSL, debe dirigir una pequeña parte del flujo de trabajo al nuevo código base y supervisar los resultados. Una vez que haya confianza en que el sistema está funcionando correctamente, el equipo debe aumentar los incrementos de carga hasta eliminar por completo el sistema anterior.
- **Encapsule las aplicaciones.** Cuando un equipo no puede actualizar una aplicación existente insegura para la memoria a un MSL, debe escribir una aplicación intermediaria para todas las interfaces públicas en el MSL. La aplicación de encapsulamiento deberá garantizar que todas las entradas no puedan exceder los límites de la memoria dentro de la aplicación secundaria.

Cómo elegir un lenguaje seguro para la memoria

Las agencias autoras recomiendan que los fabricantes de software evalúen varios MSL antes de integrarlos en sus programas de trabajo. Consulte el apéndice para obtener una descripción general de algunos lenguajes seguros para la memoria.

Capacidades y recursos del personal

Los fabricantes de software deben considerar lo siguiente:

- **Planificación del tiempo para el aprendizaje.** Asegúrese de que los equipos tengan acceso al material de capacitación y aprendizaje que puedan necesitar. Proporcione a los equipos tiempo de aprendizaje dedicado tanto para el autodidactismo como para aprender de los miembros sénior del equipo. Brinde a los nuevos empleados ejemplos de problemas que se revolvieron previamente; incluya tanto ejercicios de codificación independientes como desafíos de depuración en el MSL.
- **Planificación del tiempo para la integración.** Cree una estrategia para integrar personal nuevo en los equipos existentes y resolver conflictos potenciales, p. ej., entre los nuevos miembros del equipo que están familiarizados con el MSL y los miembros sénior más familiarizados con los lenguajes existentes.
- **Establecimiento de comunidades internas y externas.** Establezca un grupo de expertos internos de los MSL. Además de asignarles la migración y el desarrollo en sus proyectos normales, el liderazgo de ingeniería debe darles tiempo para trabajar con otros equipos y entender sus esfuerzos, desafíos y éxitos. Estos expertos pueden desarrollar una comunidad interna y difundir información entre proyectos organizando sesiones de capacitación, salas de chat y reuniones internas. Además de crear una comunidad interna, pueden ayudar a conectar su organización con expertos externos organizando reuniones tanto virtuales como presenciales.
- **Contratación e incorporación.** No siempre será posible para las organizaciones contratar a desarrolladores que ya sean expertos en MSL. En lugar de esperar para encontrar a un desarrollador que tenga el conjunto perfecto de habilidades, experiencia y conocimientos, algunas organizaciones informan haber tenido éxito en la contratación de desarrolladores que dominan múltiples lenguajes de programación, entre los que a menudo se incluyen C y C++. Para compensar la falta de experiencia del nuevo empleado en el MSL elegido, las organizaciones pueden modificar su proceso establecido de incorporación de desarrolladores a fin de incluir un sistema de mentores o compañeros y una capacitación sobre el MSL.

- **Creación de una cartera de personal.** Las organizaciones deben indicar su demanda de desarrolladores capacitados en seguridad y seguridad de la memoria a las facultades, las universidades y las instituciones educativas. Muchas instituciones basan sus planes de estudios de informática e ingeniería de software en la demanda que reciben de los estudiantes que esperan recibir ofertas de los empleadores. Si no se espera que los estudiantes necesiten una habilidad específica, las instituciones no gastarán recursos para proporcionarla.

Desafíos de implementación

Hay varios desafíos que los fabricantes de software deben tener en cuenta a medida que desarrollan y comienzan a implementar su hoja de ruta segura para la memoria, incluidos los siguientes ejemplos.

Cambio de la seguridad hacia la izquierda

Las agencias autoras recomiendan a los fabricantes de software adelantar las consideraciones de seguridad en el SDLC, lo que puede mejorar la seguridad y la confiabilidad del producto tras la implementación. Según el lenguaje de programación en cuestión, puede haber un aumento en la cantidad de trabajo inicial. Algunos MSL requieren que el equipo de desarrollo acceda a la memoria de maneras particulares, lo que puede requerir más tiempo para que el equipo adquiera competencia en la creación de código correcto desde el punto de vista idiomático. Aunque este esfuerzo adicional puede parecer pesado, mitigar las vulnerabilidades de seguridad de la memoria beneficia al equipo de desarrollo y al cliente.

Vale la pena considerar los beneficios de migrar de un entorno donde el código administra la memoria de manera adecuada la mayor parte del tiempo a otro donde el lenguaje de programación garantiza que el código administrará la memoria de manera correcta todo el tiempo.

Impacto en el desempeño

Algunos MSL usan un recolector de elementos no utilizados como parte de su administración de memoria. El proceso de recolección de elementos no utilizados puede introducir una latencia impredecible que afecta las características generales de desempeño de la aplicación, aunque algunos lenguajes pueden utilizar subprocesos adicionales para limpiar y liberar memoria. El recolector de elementos no utilizados también introducirá una sobrecarga adicional en lo que respecta al CPU y a la memoria. Si bien el impacto de la recolección de elementos no utilizados en los lenguajes de 5.ª generación y el hardware moderno es insignificante, estas características de desempeño aún pueden afectar a los dispositivos restringidos, como los de los sistemas integrados. Los desarrolladores deberán prestar atención a estas características de desempeño, especialmente en entornos que demandan desempeño en tiempo real y alta escalabilidad.

Bibliotecas existentes inseguras para la memoria

Los MSL, ya sea que utilicen un modelo de recolección de elementos no utilizados o no, casi seguramente deberán depender de las bibliotecas escritas en ciertos lenguajes, como C y C++. Si bien se están realizando esfuerzos para reescribir las bibliotecas utilizadas ampliamente en MSL, ningún esfuerzo de este tipo podrá reescribirlas todas pronto.

En un futuro predecible, la mayoría de los desarrolladores deberán trabajar en un modelo híbrido de lenguajes de programación seguros e inseguros. Los desarrolladores que comiencen a escribir en un MSL deberán interactuar con bibliotecas de C y C++ que no son seguras para la memoria. Del mismo modo, habrá situaciones en las que una aplicación insegura para la memoria deberá

interactuar con una biblioteca segura para la memoria. Al interactuar con una aplicación o un componente inseguros para la memoria, la aplicación que solicita la interacción debe conocer explícitamente los límites de la memoria definidos y limitar cualquier entrada que les transmita.

Las garantías de seguridad de la memoria que ofrecen los MSL se verán limitadas cuando los datos fluyan a través de estos límites. Otros posibles desafíos incluyen las diferencias en la clasificación de datos, el manejo de errores, la concurrencia, la depuración y el control de versiones.

Esta clase de desafío es común y está bien estudiada. Se encuentran disponibles herramientas y técnicas para administrar la interacción entre lenguajes, y es posible que sea un área para la innovación y el desarrollo en el futuro.

Excepciones de protección de memoria

También vale la pena señalar que es posible frustrar las garantías de protección de la memoria en algunos MSL. Por ejemplo, el software escrito en Rust puede contener la palabra clave "unsafe" (inseguro).[42] Existen varios motivos importantes para utilizar esta palabra clave, como interactuar directamente con el sistema operativo. Sin embargo, los desarrolladores no deben utilizar esta palabra clave para evitar la introducción de vulnerabilidades de seguridad de la memoria.

Actualización de la educación en informática

> El liderazgo a nivel ejecutivo debería impulsar la transición a lenguajes de programación seguros para la memoria porque la inseguridad de la memoria es, fundamentalmente, un problema de estrategia empresarial. Por lo tanto, el director ejecutivo (CEO, por sus siglas en inglés) u otro ejecutivo empresarial debería firmar la hoja de ruta.

Muchos, quizás la mayoría, de los programas de grado en informática no enseñan con detalle a los estudiantes los peligros de los lenguajes de programación inseguros para la memoria y los daños en el mundo real que se derivan de estos. Los motivos de este hecho son numerosos. Por ejemplo, los gerentes de contratación de muchos fabricantes de software necesitan desarrolladores que puedan trabajar en los entornos existentes, muchos de los cuales incluyen grandes códigos base de C y C++. Es posible que tengan un interés a corto plazo en contratar a estudiantes que dominen los lenguajes que utilizan principalmente. Sin embargo, abordando las necesidades a corto plazo de los fabricantes de software, se ha vuelto difícil generar impulso en torno al código seguro para la memoria.

Otro motivo de la falta de énfasis en los MSL en las universidades puede deberse a las estructuras de incentivos que determinan cómo los profesores emplean su tiempo. El costo de dominar nuevos lenguajes de programación y actualizar los trabajos del curso es considerable, y pocos profesores tienen tiempo libre para hacer esa inversión además de cumplir con sus otras obligaciones. Esta área está lista para realizar investigaciones adicionales y explorar las opciones posibles para proporcionar a los profesores las herramientas adecuadas para comenzar a enseñar MSL.

Vale la pena señalar que ciertos lenguajes, como C y C++, pueden ayudar a que los estudiantes comprendan cómo funciona la computadora en los niveles inferiores, una habilidad útil cuando estos estudiantes necesitan pensar en profundidad sobre el desempeño y la escalabilidad de los sistemas complejos. Un equilibrio reflexivo entre los trabajos del curso sobre la elección del lenguaje y los ejercicios sobre la seguridad de la memoria puede producir un graduado completo.

Hay muchas maneras en que las personas aprenden a escribir software fuera del entorno universitario. Muchas personas aprenden a programar de manera autodidacta leyendo un libro, realizando un curso en línea, leyendo blogs o viendo videos. Algunos comienzan modificando las extensiones de otra persona en su videojuego o navegador favorito y, luego, aprenden a escribir las

suyas propias. Los defensores del MSL en la industria deberían pensar en cómo implementar un sesgo en cuanto a los MSL en estas formas comunes en que la gente comienza a programar.

Sistemas del Internet de las cosas (IoT, por sus siglas en inglés), de OT y de baja potencia

Aunque muchos MSL están ampliamente disponibles para plataformas móviles, de servidor y de escritorio, están disponibles en menor medida en los sistemas restringidos, en los que la memoria, la CPU y las conexiones de la red están muy limitadas. Con frecuencia, los sistemas de OT priorizan la disponibilidad y la confiabilidad sobre muchas otras consideraciones, y carecen del amplio espectro de lenguajes de programación que se encuentran en sistemas más potentes. Cualquier transición a un MSL en sistemas de OT requerirá una demostración de desempeño, confiabilidad y garantías en tiempo real. Además, esa transición requerirá el mismo tipo de herramientas disponibles para plataformas menos restringidas para que los desarrolladores puedan construir, probar y depurar sus sistemas.

Hojas de ruta seguras para la memoria

Las agencias autoras recomiendan encarecidamente a los fabricantes de software que escriban y publiquen hojas de ruta seguras para la memoria. Al hacerlo, los fabricantes indicarán a los clientes que están adoptando los principios clave de seguridad desde el diseño: (1) se están responsabilizando de los resultados de seguridad, (2) están adoptando una transparencia radical y (3) están implementando un enfoque vertical para desarrollar productos seguros.

Los desarrolladores de software y el personal de apoyo deben elaborar la hoja de ruta, que debe detallar cómo el fabricante modificará su SDLC para reducir drásticamente y, con el tiempo, eliminar el código inseguro para la memoria en sus productos. Para garantizar recursos adecuados e indicar un apoyo de nivel ejecutivo a los clientes, las agencias autoras instan encarecidamente a los ejecutivos a identificar públicamente al personal sénior para impulsar la publicación de la hoja de ruta y ayudar a realinear los recursos según sea necesario.

La hoja de ruta debe incluir los siguientes elementos:

1. **Fases definidas con fechas y resultados.** Como ocurre con todos los esfuerzos de desarrollo de software, los equipos pueden dividir el esfuerzo mayor en proyectos más pequeños con resultados claros para medir su progreso. Las fases pueden incluir lo siguiente:
    a. Una evaluación de MSL.
    b. Una prueba piloto de la escritura de un nuevo componente en un MSL o la incorporación de un MSL en un componente existente.
    c. El modelado de amenazas para detectar el código inseguro para la memoria más peligroso.
    d. La refactorización del código inseguro para la memoria.
2. **Fecha para la incorporación del MSL en nuevos sistemas.** Publique la fecha después de la cual la empresa escribirá código nuevo únicamente en un MSL. Las organizaciones pueden poner un límite a la cantidad de posibles vulnerabilidades de seguridad de la memoria escribiendo nuevos proyectos en un MSL. Establecer públicamente una fecha para ese cambio demostrará un compromiso con la seguridad del cliente.

3. **Plan interno de integración y capacitación de desarrolladores.** Ninguna transición a MSL será gratuita, y el fabricante deberá reservar tiempo para que los desarrolladores adquieran competencia en lo siguiente:
    a. Escritura de software en el lenguaje seleccionado.
    b. Depuración.
    c. Uso de herramientas.
    d. Integración del MSL en las compilaciones.
    e. Procesos generales de control de calidad.
4. **Plan de dependencias externas.** La hoja de ruta debe documentar el plan para manejar las dependencias de bibliotecas escritas en C y C++. La mayoría de los productos de software se basan en numerosas bibliotecas de software de código abierto (OSS, por sus siglas en inglés), y muchas de ellas están escritas en C y C++. Una hoja de ruta segura para la memoria no estará completa sin incluir OSS, especialmente porque la mayoría de los productos existentes utilizan OSS.
5. **Plan de transparencia.** Mantener la información anterior actualizada con actualizaciones periódicas, p. ej., quizás trimestrales o semestrales, generará aún más confianza en que la organización está tomando en serio las vulnerabilidades de seguridad de la memoria. Además, publicar un análisis detallado de los logros y desafíos, especialmente las mejoras del SDLC, puede inspirar a otros a comenzar su proceso de seguridad de la memoria.
6. **Plan del programa de apoyo para las CVE.** La industria necesita datos públicos detallados y correctos sobre las clases de vulnerabilidades que generan riesgos para los clientes. Necesita descripciones de vulnerabilidades que brinden suficientes detalles sobre los errores de codificación para distinguir entre los defectos de seguridad de la memoria de C y C++, y otras clases de defectos. Con ese fin, las organizaciones deben comprometerse públicamente a proporcionar enumeraciones de debilidades comunes (CWE, por sus siglas en inglés) para el 100 % de las CVE de manera oportuna, así como cualquier contexto adicional para ayudar a la industria a comprender el defecto. Si bien algunos proveedores lo hacen bien en la actualidad, existen agentes notables que no lo hacen, lo que limita los conocimientos que la industria puede adquirir de los datos de vulnerabilidades.

Por último, las agencias autoras recomiendan encarecidamente a las organizaciones que utilizan un modelo de madurez evolucionar y mejorar su SDLC con el tiempo para integrar sus esfuerzos de seguridad de la memoria a fin de demostrar un mayor nivel de madurez.

Conclusión

El código inseguro para la memoria es un problema importante para los fabricantes de software y sus clientes. Los intentos anteriores de resolver el problema solo han logrado avances parciales y, actualmente, dos tercios de las vulnerabilidades informadas en los lenguajes de programación inseguros para la memoria todavía se relacionan con los problemas de la memoria. La vía más prometedora para eliminar las vulnerabilidades de seguridad de la memoria es que los fabricantes de software encuentren formas de estandarizar los lenguajes de programación seguros para la memoria y migren los componentes del software fundamentales para la seguridad a un lenguaje de programación seguro para la memoria para los códigos base existentes.

Las agencias autoras instan a los ejecutivos de los fabricantes de software a priorizar el uso de MSL en sus productos y a demostrar ese compromiso escribiendo y publicando hojas de ruta seguras para la memoria. Las agencias autoras recomiendan a los fabricantes de software liderar desde los niveles superiores designando públicamente a un ejecutivo empresarial que impulsará

personalmente la eliminación de las vulnerabilidades de seguridad de la memoria de la línea de productos.

Publicando hojas de ruta seguras para la memoria, los fabricantes de software indicarán a los clientes y a la industria que están alineados con los siguientes principios de seguridad desde el diseño:

- **Responsabilizarse de los resultados de seguridad de sus clientes.** Dada la prevalencia de vulnerabilidades de seguridad de la memoria en el mercado de software, eliminar esta clase de vulnerabilidad mejorará las posturas de seguridad de los clientes.
- **Transparencia radical.** La hoja de ruta será un plan público que detallará el enfoque que el fabricante planea adoptar para eliminar esta clase de vulnerabilidad de sus líneas de productos. El equilibrio entre las medidas de mitigación a corto plazo para reducir los peligros de los lenguajes de programación inseguros para la memoria, las transiciones a los MSL y la investigación de hardware variará ampliamente entre los fabricantes. Sin embargo, independientemente del enfoque, el objetivo debería ser el mismo: establecer un cronograma público con hitos claros para demostrar a los clientes que están realizando inversiones urgentes para resolver este problema.
- **Liderar desde los niveles superiores.** Los fabricantes de software que publican su hoja de ruta y designan públicamente a un líder empresarial para que apoye los esfuerzos demuestran que lideran desde los niveles superiores y que estas importantes iniciativas no son ocurrencias tardías que simplemente se delegaron al personal de nivel inferior.

Cuando los fabricantes de software se topan con obstáculos, deben plantear esos desafíos y sugerir posibles soluciones como parte de la comunidad de interés que busca resolver esta clase de errores de codificación. Trabajando juntos en estos desafíos, la industria del software puede identificar y promover soluciones que ninguna organización puede lograr por sí sola. Este es un problema que afecta a toda la industria y resolverlo requerirá una respuesta de toda la industria.

Independientemente del enfoque, las agencias autoras instan a las organizaciones a actuar de inmediato para reducir y, con el tiempo, eliminar las vulnerabilidades de seguridad de la memoria de sus productos.

Recursos

La siguiente es una lista no exhaustiva de recursos disponibles públicamente sobre seguridad de la memoria y lenguajes de programación seguros para la memoria:

- Blog de la CISA: La necesidad urgente de seguridad de la memoria en los productos de software
- Fundación de Seguridad de Código Abierto (Open Source Security Foundation) (Linux Foundation) https://openssf.org/
- Grupo de Investigación de Seguridad de Internet (Internet Security Research Group) (abetterinternet.org) https://www.abetterinternet.org/
- Chris Palmer presenta los esfuerzos de Google para evitar las vulnerabilidades de seguridad de la memoria en el sistema operativo Android. https://www.usenix.org/conference/enigma2021/presentation/palmer
- Alex Gaynor presenta las reacciones comunes a los lenguajes de programación seguros para la memoria https://www.usenix.org/conference/enigma2021/presentation/gaynor
- Introducción a la inseguridad de la memoria para vicepresidentes de ingeniería https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/
- Prossimo, un proyecto del Grupo de Investigación de Seguridad de Internet (ISRG, por sus siglas en inglés): https://www.memorysafety.org/docs/memory-safety/
- Atlantic Council, Reducir el riesgo de compra: seguridad de la memoria: https://www.atlanticcouncil.org/content-series/buying-down-risk/memory-safety/
- "Retención de ciclos y administración de la memoria en Swift", de İsmail GÖK, que incluye una buena explicación del sistema de conteo automático de referencias (ARC, por sus siglas en inglés) de Swift. https://betterprogramming.pub/retain-cycles-and-memory-management-in-swift- fb6226165b17
- Visualización de la administración de la memoria en Rust (parte de una serie), de Deepu K Sasidharan: https://deepu.tech/memory-management-in-rust/
- Ada SPARK es un lenguaje de programación, un conjunto de herramientas de verificación y un método de diseño destinado a entornos en los que la alta confiabilidad es un requisito. https://www.adacore.com/about-spark
- Información sobre lenguajes de programación seguros para la memoria que se diseñaron aproximadamente al mismo tiempo que C: https://noncombatant.org/2023/05/21/protel-sos-dsm-100/
- Adam Zabrocki, Alex Tereshkin: Explotación en la era de la verificación formal https://www.youtube.com/watch?v=TcIaZ9LW1WE Medida de mitigación SPARK en DEF CON 30.
- Desempeño de C en comparación con el de Rust: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html
- Hoja de información de ciberseguridad de la NSA. Seguridad de la memoria del software: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Apéndice: Lenguajes seguros para la memoria

Lenguaje	Descripción
C#	Microsoft presentó C# en el 2000 y lo diseñó para que fuera un lenguaje simple, eficiente y con seguridad de tipos adecuado para una amplia variedad de aplicaciones, incluidas aplicaciones móviles, aplicaciones web y juegos. El código fuente de C# se compila en un lenguaje intermedio, llamado lenguaje intermedio común (CIL, por sus siglas en inglés), antes de que lo ejecute el entorno en tiempo de ejecución .NET. Asimismo, C# se usa ampliamente para crear aplicaciones de servidor y de escritorio de Windows, y también está disponible en Linux y MacOS para arquitecturas x86, x64 y ARM.
Go	Go es un lenguaje de programación compilado multiplataforma desarrollado por Google y lanzado en el 2007. Google lo diseñó para que fuera simple y eficiente de usar, y es muy adecuado para desarrollar aplicaciones concurrentes y en red. Es sintácticamente como C, pero con seguridad de memoria, recolección de elementos no utilizados y tipificación estructural. Varias empresas de alto perfil han migrado algunos sistemas a Go desde otros lenguajes, como Python. Ciertas aplicaciones, como Terraform, Docker y Kubernetes, están escritas en Go.
Java	Java es un MSL de recolección de elementos no utilizados propiedad de Oracle y lanzado a mediados de la década de 1990. Es uno de los lenguajes más populares[43] y se utiliza en aplicaciones web, software empresarial y aplicaciones móviles. El código fuente de Java se compila en códigos de bytes de la máquina virtual Java (JVM, por sus siglas en inglés) que se pueden ejecutar en cualquier máquina JVM y es independiente de la plataforma.
Python	Python se lanzó por primera vez en 1991. Por lo general, es un lenguaje interpretado, aunque se puede compilar en código de bytes. Se tipifica dinámicamente, y se recolectan elementos no utilizados. Se ejecuta en Windows, Mac y Linux, y es popular para escribir aplicaciones web y algunos sistemas integrados, como Raspberry Pi. Con frecuencia, se cita como el lenguaje de programación más popular.[44]
Rust	Mozilla lanzó Rust en 2015. Es un lenguaje compilado y se centra en el desempeño, la seguridad de tipos y la concurrencia. Tiene un modelo de propiedad diseñado para garantizar que solo haya un propietario de un dato. Tiene una característica denominada “verificador de préstamos” diseñada para garantizar la seguridad de la memoria y evitar carreras de datos concurrentes. Si bien no es perfecto, el sistema de verificación de préstamos contribuye en gran medida a abordar los problemas de seguridad de la memoria en el momento de la compilación. Rust aplica la corrección en el momento de la compilación para evitar problemas de seguridad de la memoria y concurrencia en el tiempo de ejecución. Por ejemplo, una carrera de datos es un tipo de error de software muy difícil de localizar. “Con Rust, puede verificar estáticamente que no tiene carreras de datos. Esto significa que, en primer lugar, para evitar errores difíciles de depurar, simplemente no permite que ingresen a su código. El compilador no le permitirá hacerlo”.[45] Rust ha recibido mucha atención por parte de varias tecnologías de alto perfil, incluido el núcleo de Linux, Android y Windows. También se utiliza en aplicaciones, como las de Mozilla, y en otros servicios en línea, como Dropbox, Amazon y Facebook.[46]
Swift	Apple lanzó el lenguaje de programación Swift en 2014 y lo diseñó para que fuera fácil de leer y escribir. Se diseñó para ser el reemplazo de C, C++ y Objective-C. Es posible incorporar código de Swift en código de Objective-C existente para simplificar la migración a Swift. Swift se utiliza principalmente para desarrollar aplicaciones iOS, Watch OS y Mac OS X. Apple afirma que Swift es hasta 2.6 veces más rápido que Objective-C.

Propósito

Las autoridades de ciberseguridad de EE. UU., Australia, Canadá, el Reino Unido y Nueva Zelanda desarrollaron esta orientación para promover sus respectivas misiones de ciberseguridad, incluidas sus responsabilidades de desarrollar y emitir especificaciones y medidas de mitigación de ciberseguridad.

Agradecimientos

Las autoridades de ciberseguridad de EE. UU., Australia, Canadá, el Reino Unido y Nueva Zelanda desean agradecer a Microsoft por sus contribuciones a esta orientación.

Descargo de responsabilidad

La información contenida en este informe se proporciona "tal cual" solo con fines informativos. La CISA, la NSA, la FBI, el ACSC, el CCCS, el NCSC-UK, el NCSC-NZ y el CERT-NZ no respaldan ningún producto o servicio comercial, incluido cualquier tema de análisis. Cualquier referencia a productos, procesos o servicios comerciales específicos mediante marcas de servicio, marcas registradas, fabricantes o de otro modo no constituye ni implica la promoción, la recomendación ni la preferencia.

Información de contacto

Organizaciones de EE. UU.: Informen los incidentes y la actividad anómala al Centro de Operaciones de la CISA, disponible las 24 horas, los 7 días de la semana, enviando un correo electrónico a report@cisa.gov o llamando al (888) 282-0870; o bien, a la FBI a través de su oficina local de la FBI, al centro CyWatch de la FBI, disponible las 24 horas, los 7 días de la semana, llamando al (855) 292-3937 o enviando un correo electrónico a CyWatch@fbi.gov. Si es posible, incluyan la siguiente información sobre el incidente: la fecha, la hora y la ubicación del incidente; el tipo de actividad; la cantidad de personas afectadas; el tipo de equipo utilizado para la actividad; el nombre de la empresa u organización que presenta la denuncia; y un punto de contacto designado. Para realizar comentarios sobre este documento, comuníquense con SecureByDesign@cisa.dhs.gov. **Organizaciones de Australia:** Visiten cyber.gov.au o llamen al 1300 292 371 (1300 CYBER 1) para denunciar incidentes de ciberseguridad y acceder a las alertas y los avisos. **Organizaciones de Canadá:** Denuncien los incidentes enviando un correo electrónico al CCS a contact@cyber.gc.ca. **Organizaciones del Reino Unido :** Denuncie un incidente importante de ciberseguridad en ncsc.gov.uk/report-an-incident (supervisado las 24 horas) o, para recibir asistencia urgente, llamen al 03000 200 973. **Organizaciones de Nueva Zelanda:** Denuncien los incidentes de ciberseguridad a incidents@ncsc.govt.nz o llamen al 04 498 7654.

Referencias

[1] Microsoft. “A Proactive Approach to More Secure Code.” Microsoft Security Response Center (MSRC) Blog. July 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[2] Chromium. “Memory Safety.” Chromium Security. n.d. https://www.chromium.org/Home/chromium-security/memory-safety/.

[3] Hosfelt, Diane. “Implications of Rewriting a Browser Component in Rust.” Mozilla Hacks - the Web Developer Blog. March 5, 2019. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[4] Existen varios tipos de errores de codificación relacionados con la memoria, entre los que se incluyen los siguientes:

1. **Saturación del búfer** [CWE-120: Copia del búfer sin comprobar el tamaño de la entrada (“saturación clásica del búfer”)], donde un programa intenta escribir datos en un búfer, pero supera el límite del búfer y sobrescribe otra memoria en el espacio de direcciones.
2. **Uso después de la liberación** [CWE-416: Uso después de la liberación], donde un programa elimina la referencia de un puntero colgante de un objeto que ya se ha eliminado.
3. **Uso de memoria no inicializada** [CWE-908: Uso de recurso no inicializado], donde la aplicación accede a la memoria que no se ha inicializado.
4. **Doble liberación** [CWE-415: Doble liberación], en la que un programa intenta liberar memoria que ya no necesita dos veces y, posiblemente, daña las estructuras de datos de administración de la memoria.

[5] MITRE. “CWE CATEGORY: Comprehensive Categorization: Memory Safety.” Common Weakness Enumeration. n.d. https://cwe.mitre.org/data/definitions/1399.html#:~:text=CWE%20-%20CWE-1399%3A%20Comprehensive%20Categorization%3A%20Memory%20Safety%20%284.12%29,Community-Developed%20List%20of%20Software%20%26%20Hardware%20Weakness%20Types

[6] Hicks, Michael. “What Is Memory Safety? - The PL Enthusiast.” The Programming Languages Enthusiast. January 22, 2018. http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

[7] One, Aleph. “Smashing The Stack For Fun And Profit.” University of California, Berkeley. 2008. https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.

[8] Microsoft. “A Proactive Approach to More Secure Code.” Microsoft Security Response Center (MSRC) Blog. July 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[9] Chromium. “Memory Safety.” Chromium Security. n.d. https://www.chromium.org/Home/chromium-security/memory-safety/.

[10] Hosfelt, Diane. “Implications of Rewriting a Browser Component in Rust.” February 28, 2019. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[11] Stone, Maddie. “The More You Know, The More You Know You Don’t Know.” April 2022. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[12] Manès VJ, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M. “The Art, Science, and Engineering of Fuzzing: A Survey . IEEE Transactions on Software Engineering.” 2019 Oct 11;47(11):2312-31 [LINK]

[13] Open Web Application Security Project (OWASP) Foundation. “OWASP Top Ten.” OWASP. n.d. https://owasp.org/www-project-top-ten/.

[14] Bastien, JF. “Keynote: Safety and Security: The Future of C++.” CppNow (YouTube Channel). 2023. https://www.youtube.com/watch?v=Gh79wcGJdTg.

[15] Hinnant, H, Orr, B, Stroustrup, B, Vandevoorde, D, Wong, M. “Opinion on Safety for ISO C++” https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2759r0.pdf.

[16] Apple. “Memory Safe iBoot Implementation.” Apple Support. February 18, 2021. https://support.apple.com/en-il/guide/security/sec30d8d9ec1/web.

[17] Amar, Saar. “Introduction to Firebloom (iBoot).” iBoot_Firebloom. n.d. https://saaramar.github.io/iBoot_firebloom/.

[18] Microsoft. “GitHub - Microsoft/Checked” n.d. https://github.com/microsoft/checkedc.

[19] Neumann, Thomas "P2771R0: Towards Memory Safety in C++." Open STD. January 17, 2023. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2771r0.html.

[20] Carbon-Language. "GitHub - Carbon-Language." n.d. https://github.com/carbon-language/carbon-lang#memory-safety.

[21] CppNow23. "YouTube: Carbon Language Successor Strategy: From C++ Interop to Memory Safety - Chandler Carruth - CppNow 23." September 2023. https://youtu.be/1ZTJ9omXOQ0?si=t8KsLWBv1DDFOUXs&t=3455.

[22] Fei, Shufan, Zheng Yan, Wenxiu Ding, and Haomeng Xie. "Security Vulnerabilities of SGX and Countermeasures: A Survey." July 13, 2021. *ACM Computing Surveys* 54 (6): 1–36. https://dl.acm.org/doi/abs/10.1145/3456631.

[23] Microsoft. "Control Flow Guard for Clang/LLVM and Rust." Microsoft Security Response Center. August 2020. https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/.

[24] Yunhai, Zhang. "Bypass Control Flow Guard Comprehensively." BlackHat. July 19, 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf.

[25] iamelli0t. "Exploiting Windows RPC to Bypass CFG Mitigation: Analysis of CVE-2021-26411 In-the-Wild Sample." iamelli0t's Blog (blog). April 10, 2021. https://iamelli0t.github.io/2021/04/10/RPC-Bypass-CFG.html.

[26] Groß, Samuel. "Remote iPhone Exploitation Part 2: Bringing Light into the Darkness -- a Remote ASLR Bypass." Google Project Zero blog. January 2020. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-2.html.

[27] Jurczyk, Mateusz. "MMS Exploit Part 5: Defeating Android ASLR, Getting RCE." Google Project Zero blog. August 2020. https://googleprojectzero.blogspot.com/2020/08/mms-exploit-part-5-defeating-aslr-getting-rce.html.

[28] Microsoft. "MS15-124. Vulnerability in Internet Explorer Could Lead to ASLR Bypass: December 16, 2015." Microsoft Support. n.d. https://support.microsoft.com/en-us/topic/ms15-124-vulnerability-in-internet-explorer-could-lead-to-aslr-bypass-december-16-2015-7e012708-4af6-487c-12e2-4ffa0f9d7b66.

[29] Boneh, Dan. "On the effectiveness of address-space randomization." ACM Digital Library. October 25, 2004. https://dl.acm.org/doi/10.1145/1030083.1030124.

[30] Palmer, Chris (Google Chrome Security). "The Limits of Sandboxing and Next Steps." February 03, 2021. https://www.usenix.org/conference/enigma2021/presentation/palmer.

[31] Watson, Robert N. M. (University of Cambridge), Moore, Simon W. (University of Cambridge), Sewell, Peter (University of Cambridge), Davis, Brooks (SRI International), Neumann, Peter (SRI International). "Capability Hardware Enhanced RISC Instructions (CHERI)." September 2023. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[32] Innovate UK, EPSRC and ESRC. "Digital Security by Design." Digital Catapult. n.d. https://www.dsbd.tech/.

[33] Cowie, Alan. "Digital Security by Design Driving Investment in the Automotive Sector and Embedded Systems - Innovate UK KTN." Innovate UK KTN (blog). September 12, 2023. https://iuk.ktn-uk.org/news/digital-security-by-design-driving-investment-in-the-automotive-sector-and-embedded-systems/.

[34] Mitsunami, Koki. "Enhanced Security through Memory Tagging Extension." June 24, 2021. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced- security-through-mte.

[35] Rahman, Mishaal. "Android 14 May Add an Advanced Memory Protection Feature to Protect Your Device from Memory Safety Bugs." XDA Developers. February 8, 2023. https://www.xda-developers.com/android-14- advanced-memory-protection/.

[36] Koduri, Raja. (Intel). Intel Architecture Day 2020. Intel. https://d1io3yog0oux5.cloudfront.net/_6f1902c731ed10bd8538c1c8c9ca7ca1/intel/db/861/8422/pdf/Intel-Architecture-Day-2020-Presentation-Slides.pdf.

[37] Larabel, Michael. "Intel Working To Combine The Best Of CET + CFI Into 'FineIBT'" phoronix. August 6, 2021. https://www.phoronix.com/news/Intel-FineIBT-Security.

[38] Grauer, Yael, Dhalla, Amira (Consumer Reports). "Report: Future of Memory Safety." CR Advocacy. January 23, 2023. https://advocacy.consumerreports.org/research/report-future-of-memory-safety/.

[39] TIOBE. "TIOBE Index for November 2023, November Headline: Kotlin still on the rise in the TIOBE index." TIOBE Index. November 2023. https://www.tiobe.com/tiobe-index/.

[40] Palmer, Chris (Noncombatant.org). "Prioritizing Memory Safety Migrations." April 9, 2021. https://noncombatant.org/2021/04/09/prioritizing-memory-safety-migrations/.

[41] Walbran, Andrew (Android Rust Team, Google). "Bare-Metal Rust in Android." Google Online Security Blog. October 9, 2023. https://security.googleblog.com/2023/10/bare-metal-rust-in-android.html.

[42] Klabnik, Steve, Nichols, Carol. "Unsafe Rust - The Rust Programming Language." The Rust Programming Language. February 9, 2023. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[43] TIOBE. "The Java Programming Language." TIOBE Index. November 2023. https://www.tiobe.com/tiobe-index/java/.

[44] Cass, Stephen. "The Top Programming Languages 2023." IEEE Spectrum, November 14, 2023. https://spectrum.ieee.org/the-top-programming-languages-2023.

[45] Clark, Lin. "Inside a Super Fast CSS Engine: Quantum CSS (Aka Stylo) - Mozilla Hacks - the Web Developer Blog." Mozilla Hacks - the Web Developer Blog. October 9, 2017. https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/.

[46] Newman, Lily Hay. "The Rise of Rust, the 'Viral' Secure Programming Language That's Taking over Tech." *WIRED*, November 2, 2022. https://www.wired.com/story/rust-secure-programming-language-memory-safe/.