Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

The memory safety problem

How a software program manages memory is core to preventing many vulnerabilities and ensuring a program is robust. Exploiting poor or careless memory management can allow a malicious cyber actor to perform nefarious acts, such as crashing the program at will or changing the instructions of the executing program to do whatever the actor desires. Even un-exploitable issues with memory management can result in incorrect program results, degradation of the program's performance over time, or seemingly random program crashes.

Memory safety is a broad category of issues related to how a program manages memory. One common issue is called a "buffer overflow" where data is accessed outside the bounds of an array. Other common issues relate to memory allocation. Languages can allocate new memory locations as a program is executing and then deallocate the memory, also called releasing or freeing the memory, later when the memory is no longer needed. But if this is not done carefully by the developer, new memory may be allocated again and again as the program executes. Consequently, memory is not always freed when it is no longer needed, resulting in a memory leak that could cause the program to eventually run out of available memory. Due to logic errors, programs can also attempt to use memory that has been freed, or even free memory that has already been freed. Another issue can arise when languages allow the use of a variable that has not been initialized, resulting in the variable using the value that was previously set at that location in memory. Finally, another challenging issue is called a race condition. This issue can occur when a program's results depend on the order of operation of two parts of the program accessing the same data. All of these memory issues are much too common occurrences.

By exploiting these types of memory issues, malicious actors—who are not bound by normal expectations of software use—may find that they can enter unusual inputs into the program, causing memory to be accessed, written, allocated, or deallocated in unexpected ways. In some cases, a malicious actor can exploit these memory management mistakes to access sensitive information, execute unauthorized code, or cause other negative impacts. Since it may take a lot of experimenting with unusual inputs to find one that causes an unexpected response, actors may use a technique called "fuzzing" to either randomly or intelligently craft multitudes of input values to the program until one is found that causes the program to crash. Advances in fuzzing tools

and techniques have made finding problematic inputs easier for malicious actors in recent years. Once an actor discovers they can crash the program with a particular input, they examine the code and work to determine what a specially crafted input could do. In the worst case, such an input could allow the actor to take control of the system on which the program is running.

Memory safe languages

Using a memory safe language can help prevent programmers from introducing certain types of memory-related issues. Memory is managed automatically as part of the computer language; it does not rely on the programmer adding code to implement memory protections. The language institutes automatic protections using a combination of compile time and runtime checks. These inherent language features protect the programmer from introducing memory management mistakes unintentionally. Examples of memory safe language include C#, Go, Java®, Ruby™, Rust®, and Swift®.

Even with a memory safe language, memory management is not entirely memory safe. Most memory safe languages recognize that software sometimes needs to perform an unsafe memory management function to accomplish certain tasks. As a result, classes or functions are available that are recognized as non-memory safe and allow the programmer to perform a potentially unsafe memory management task. Some languages require anything memory unsafe to be explicitly annotated as such to make the programmer and any reviewers of the program aware that it is unsafe. Memory safe languages can also use libraries written in non-memory safe languages and thus can contain unsafe memory functionality. Although these ways of including memory unsafe mechanisms subvert the inherent memory safety, they help to localize where memory problems could exist, allowing for extra scrutiny on those sections of code.

Languages vary in their degree of memory safety instituted through inherent protections and mitigations. Some languages provide only relatively minimal memory safety whereas some languages are very strict and provide considerable protections by controlling how memory is allocated, accessed, and managed. For languages with an extreme level of inherent protection, considerable work may be needed to simply get the program to compile due to the checks and protections.

Memory safety can be costly in performance and flexibility. Most memory safe languages require some sort of garbage collection to reclaim memory that has been

allocated, but is no longer needed by the program. There is also considerable performance overhead associated with checking the bounds on every array access that could potentially be outside of the array.

Alternatively, a similar performance hit can exist in a non-memory safe language due to the checks a programmer adds to the program to perform bounds checking and other memory management protections. Additional costs of using non-memory safe languages include hard-to-diagnose memory corruption and occasional program crashes along with the potential for exploitation of memory access vulnerabilities

It is not trivial to shift a mature software development infrastructure from one computer language to another. Skilled programmers need to be trained in a new language and there is an efficiency hit when using a new language. Programmers must endure a learning curve and work their way through any “newbie” mistakes. While another approach is to hire programmers skilled in a memory safe language, they too will have their own learning curve for understanding the existing code base and the domain in which the software will function.

Application security testing

Several mechanisms can be used to harden non-memory safe languages to make them more memory safe. Analyzing the software using static and dynamic application security testing (SAST and DAST) can identify memory use issues in software.

Static analysis examines the source code to find potential security issues. Using SAST allows all of the code to be examined, but it can generate a considerable number of false positives through identifying potential issues incorrectly. However, SAST can be used throughout the development of the software allowing issues to be identified and fixed early in the software development process. Rigorous tests have shown that even the best-performing SAST tools only identify a portion of memory issues in even the simplest software programs and usually generate many false positives.

In contrast to SAST, dynamic analysis examines the code while it is executing. DAST requires a running application. This means most issues will not be identified until late in the development cycle, making the identified problem more expensive to fix and regressively test. DAST can only identify issues with code that is on the execution path when the tool is run, so code coverage is also an issue. However, DAST has a much

lower percentage of false positives than SAST. Issues such as a memory leak can be identified by DAST, but the underlying cause of the memory issue may be very difficult to identify in the software.

Neither SAST nor DAST can make non-memory safe code totally memory safe. Since all tools have their strengths and weaknesses, it is recommended that multiple SAST and DAST tools be run to increase the chances that memory or other issues are identified. Working through the issues identified by the tools can take considerable work, but will result in more robust and secure code. Vulnerability correlation tools can intake the results from multiple tools and integrate them into a single report to simplify and help prioritize analysis.

Anti-exploitation features

The compilation and execution environment can be used to make it more difficult for cyber actors to exploit memory management issues. Most of these added features focus on limiting where code can be executed in memory and making memory layout unpredictable. As a result, this reduces a malicious actor's opportunities to use the exploitation tradecraft of executing data as code and overwriting a return address to direct program flow to a nefarious location.

Leveraging options, such as Control Flow Guard (CFG), will place restrictions on where code can be executed. Similarly, Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) add unpredictability to where items are located in memory and prevent data from being executed as code. [6] [7] Bypassing ASLR and DEP is not insurmountable to a malicious actor, but it makes developing an exploit much more difficult and lowers the odds of an exploit succeeding. Anti-exploitation features can help mitigate vulnerabilities in both memory safe and non-memory safe languages.

The path forward

Memory issues in software comprise a large portion of the exploitable vulnerabilities in existence. NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®. Memory safe languages provide

differing degrees of memory usage protections, so available code hardening defenses, such as compiler options, tool analysis, and operating system configurations, should be used for their protections as well. By using memory safe languages and available code hardening defenses, many memory vulnerabilities can be prevented, mitigated, or made very difficult for cyber actors to exploit.▪

Works cited

[1] Microsoft® (2019), "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape". https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[2] Google (2021), "An update on Memory Safety in Chrome". https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

[3] Georgia Institute of Technology (2022), "Recommendations from the Workshop on Open-source Software Security Initiative". https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/a/2878/files/2022/10/OSSI-Final-Report.pdf

[4] The Linux® Foundation (2022), "The Linux Foundation and Open Source Software Security Foundation (OpenSSF) Gather Industry and Government Leaders for Open Source Software Security Summit II". https://www.linuxfoundation.org/press/press-release/linux-foundation-openssf-gather-industry-government-leaders-open-source-software-security-summit

[5] The Linux Foundation Open Source Security Foundation (2022), "The Open Source Software Security Mobilization Plan". https://openssf.org/oss-security-mobilization-plan/

[6] National Security Agency (2019), Windows® 10 for Enterprises Security Benefits of Timely Adoption. https://media.defense.gov/2019/Jul/16/2002158052/-1/-1/0/CSI-WINDOWS-10-FOR-ENTERPRISE-SECURITY-BENEFITS-OF-TIMELY-ADOPTION.PDF

[7] National Security Agency (2019), Leverage Modern Hardware Security Features. https://media.defense.gov/2019/Sep/09/2002180345/-1/-1/0/Leverage%20Modern%20Hardware%20Security%20Features%20-%20Copy.pdf

Disclaimer of endorsement

Trademarks

Purpose

This document was developed in furtherance of NSA's cybersecurity missions, including its responsibilities to identify and disseminate threats to National Security Systems, Department of Defense, and Defense Industrial Base information systems, and to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

Contact

General Cybersecurity Inquiries: CybersecurityReports@nsa.gov
Defense Industrial Base Inquiries and Cybersecurity Services: DIB_Defense@cyber.nsa.gov
Media Inquiries / Press Desk: 443-634-0721, MediaRelations@nsa.gov