BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

Improving the Security of Garbage Collectors in the Java Virtual Machine

Bc. Patrik Čerbák*

Abstract
This project focuses on improving the security of the Garbage-First garbage collector in the JVM. The increase in security is achieved through continuous relocation of memory regions used by the garbage collector and by "locking" the original memory in which these regions were located. Thanks to this relocation, the dereference of dangling pointers (or other pointers created, for example, by a buffer overflow), which are still pointing to the original memory, would be detected. This makes it much easier to detect memory vulnerabilities in this garbage collector and to debug them.

*xcerba00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The aim of this project is to extend the Garbage-First (G1 for short) garbage collector in the Java Virtual Machine (JVM for short) to be less vulnerable to memory-related vulnerabilities by creating mechanisms to identify when this vulnerability occurs and making them easier to debug by locating the place in the code, *where* the vulnerability occurred.

The idea for this project came from my colleagues at the OpenJDK team at IBM (formerly Red Hat), specifically Martin Balao Alonso and Francisco Ferrari Bihurriet.

2. Common Memory Vulnerabilities

When discussing vulnerabilities, memory vulnerabilities such as use-after-free and buffer overflow are undoubtedly among the most common [1][2], both coming from memory mismanagement. While a use-after-free vulnerability is caused by a dangling pointer accessing deallocated memory, a buffer overflow occurs when an application reads or writes beyond the bounds of an allocated buffer.

Figure 1 illustrates a classic example of a use-after-free memory vulnerability. In this diagram, we see how a dangling pointer is created. Initially, pointer 1 points to object 1. Once the object is deallocated, pointer 1 becomes a dangling pointer because it still points to the same memory location (it was not destroyed). If object 2 is subsequently allocated in that same memory space (pointed to by pointer 2), the dangling pointer 1 can still access it. This allows an attacker to potentially read or modify sensitive data belonging to the new object if they gain access to pointer 1.

Figure 2 demonstrates the buffer overflow vulnerability. The diagram illustrates a 32-byte allocated buffer containing a string, with the pointer `buffer` correctly pointing to its start. However, an attacker might iterate past the end of the array. In the figure, the pointer `buffer` attempts to access memory outside the permitted bounds, pointing to the value 61. This out-of-bounds access could potentially expose sensitive information stored in adjacent memory structures.

3. The Garbage-First Garbage Collector

The Garbage-First garbage collector (or G1 GC for short) is the de facto default garbage collector in the JVM, and it is ideal for almost all use cases. G1 pre-allocates the entire heap memory as a single huge chunk and divides it into equally sized structures known as heap regions [3].

Every heap region currently in use belongs to a generation. The main generations are *young* (further split into *eden* and *survivor*) and *old*. The young generation regions store new objects that are most likely to be collected, and the old generation stores objects that have already survived multiple collections and are likely to survive the next ones. [4]

G1 manages these heap regions by storing them in a so-called *free region list*, taking them out only when it needs more space for object allocation. Later in the application's lifecycle, when all objects within a region are completely deallocated or moved elsewhere (G1 "evacuates" surviving objects to old regions, so that they are closer together), the empty region is returned back to the free region list for future use.

4. The G1 Extension

To secure the G1 GC against these memory vulnerabilities, we designed an extension called **SanitizeGC**. This extension introduces a continuous memory relocation mechanism that allocates new memory each time a region is taken from the free region list and moves the region there. Also, it removes read and write permissions from the old memory where the region previously resided, effectively "locking" the old memory and making every access there invalid.

Figure 3 visually explains the movement and locking of these heap regions. It shows the starting state where the regions are located contiguously. As the program executes, some regions are moved to new memory addresses, leaving behind their original memory spaces marked with an 'X' to represent that they are now inaccessible and locked. The diagram further shows that these regions can be moved multiple times. By scattering the heap and locking old memory, any dereference of a dangling pointer, or an out-of-bounds pointer from a buffer overflow, will hit locked memory, causing the JVM to catch the illegal access and print a stack trace for easy debugging. This also makes this extension a potentially useful tool during the development or testing of the JVM, as if there are new memory vulnerabilities being created, they would be caught early in the development cycle.

5. Current State of Implementation

The SanitizeGC extension is currently implemented as a proof-of-concept for the OpenJDK version 23 (that was the current version when we started working on this project) on the Linux x86-64 platform. It can be activated at runtime using an optional command-line switch `-XX:+SanitizeGC`. To implement this without breaking the garbage collector's internal logic, an address mapper utilizing hash maps was developed. This mapper is crucial because the G1 GC relies on the original heap address offsets for different calculations (for example, in structures like card tables). The mapper seamlessly translates between the newly moved addresses and the original heap addresses (and vice versa in some cases).

Furthermore, to prevent excessive physical memory consumption when regions are locked, the implementation utilizes Linux anonymous memory, only locking the virtual address space (on x86-64 systems, the virtual address space spans 256 TiB (2^{48} bytes) [5]) while freeing the underlying physical pages.

The SanitizeGC currently successfully remaps addresses across more than 30 required code locations. It does, however, not yet support special "humongous" heap regions (regions spanning the memory of multiple normal regions), and compresses OOPs (a mechanism to make address pointers in the JVM compressed), as those are beyond the proof-of-concept nature of this project/thesis.

Acknowledgements

I would like to thank my colleagues (and former colleague) from IBM, mainly Francisco Ferrari Bihurriet, Martin Balao Alonso, Thomas Fitzsimmons, and my supervisor, Ing. David Kozák, Ph.D., for their help in writing this project.

References

[1] CISA. The urgent need for memory safety in software products. online, 12 2023.

[2] OpenSSF. Announcing the release of "the memory safety continuum". online, 04 2025.

[3] Monica Beckwith. Garbage first garbage collector tuning. online, 08 2013.

[4] Oracle Corporation. Garbage-first (g1) garbage collector. online.

[5] Wikipedia contributors. x86-64. online, 03 2026.