Dissecting Google’s AddressSanitizer (ASAN)

Vishal Juneja, Computer Science (Cybersecurity)

Mentor: Yan Shoshitaishvili, Associate Professor, School of Computing and Augmented Intelligence

Introduction

AddressSanitizer (ASAN) is an automatic error detection tool made by Google. It is used to find bugs such as Use-After-Free, Out-Of-Bound accesses to heap, stack and global objects etc.

Problems

1. ASAN is not very well documented. Current documentation is very concise, not very user friendly and lacks detailed explanation. On top of it, resources for learning ASAN is very scattered. The error messages produced by ASAN are difficult to interpret.
2. ASAN has high memory and time overhead. Average slowdown caused by ASAN is 73% [1].

Background Study


Future Work

Currently, we are conducting tests on various programs, calculating the overall memory and time overhead caused by ASAN. We are in the process of identifying the components that are causing the overhead, and then we plan to enhance the performance of these components by improving their functionality.

Observations

1. We researched on the inner workings of ASAN and identified the important components.
2. The backbone of ASAN involves shadow memory, which stores metadata about the program’s memory usage.
3. Shadow memory indicates whether memory is accessible and its state, such as if it is accessible, partially accessible, or inaccessible.
4. The correspondence between program memory and shadow memory is one shadow memory byte for every 8 bytes of program memory.
5. A shadow memory byte of 0 indicates that all 8 bytes are accessible, while other values from 1 to 7 represent partial accessibility, and -1 means no memory is accessible.
6. The ASAN Shadow Memory address calculation formula allows the computation of shadow memory addresses.
7. Instrumentation involves checking the shadow memory byte before memory access, introducing a time overhead for enhanced security.
8. The runtime library in ASAN manages shadow memory, replacing standard memory allocation functions with specialized implementations.
9. Malloc allocates redzones around memory regions to detect overflows and underflows, with redzone size influencing detection scope.
10. Free function poisons memory regions and places them in quarantine as a FIFO queue.
11. Redzones for global objects are generated at compile time.

Background Study