# Bug Triage Report

## Introduction

Bug triage is the process of reviewing, categorizing, and prioritizing bugs or vulnerabilities found in software to determine their severity and assign resources for their resolution. This Lab2 report employs the AddressSanitizer Memory Detector to triage the crashes discovered in Lab1, which resulted from fuzzing Fuzzgoat with GCC-based, LLVM-based, and QEMU-based instrumentations.

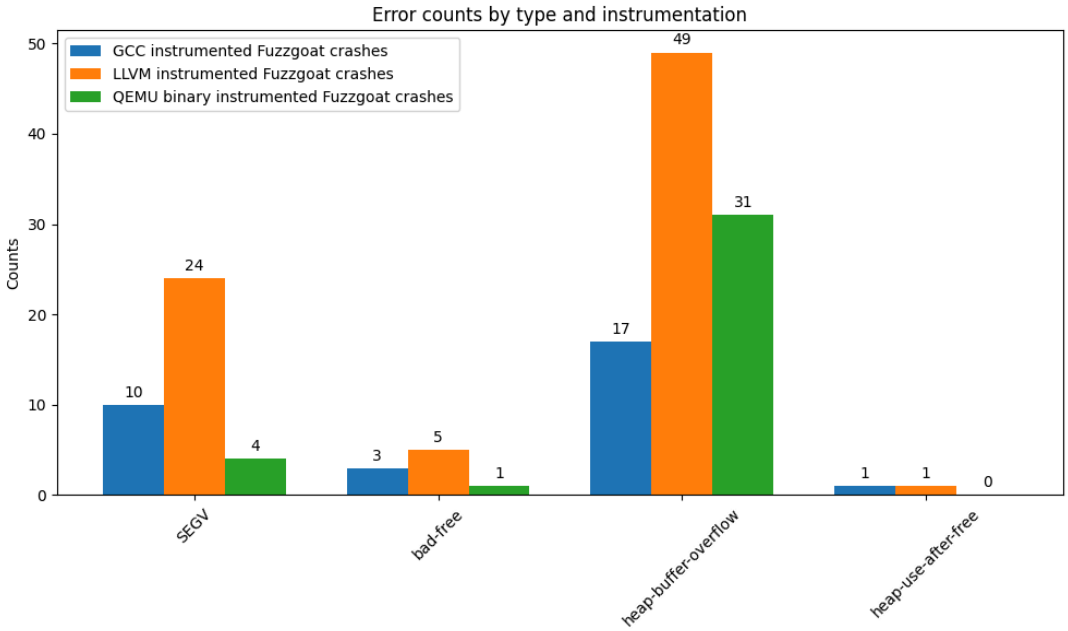
## Methodology

After successfully capturing crash data through the fuzzing of Fuzzgoat, the subsequent phase encompasses a detailed triage of these crashes. AddressSanitizer (ASAN), renowned for its adeptness at uncovering memory-related errors, was the tool of choice for recompiling Fuzzgoat, enabling a more nuanced analysis of the crashes. This recompilation facilitated the use of crash files as inputs to rerun Fuzzgoat under conditions that closely mimic the fuzzing environment, thereby generating detailed error reports and stack traces for each crash.

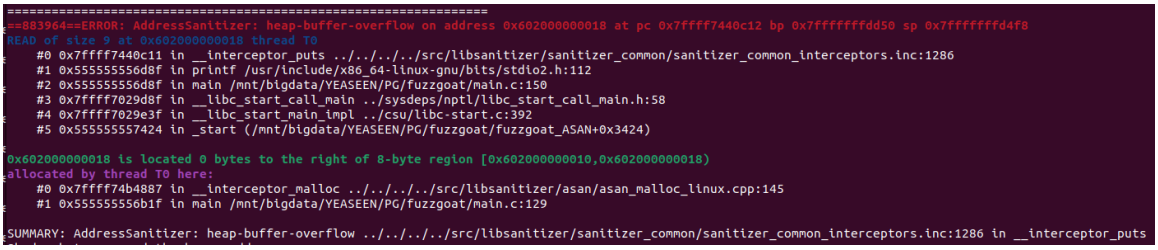
To streamline the triage process, a Python script was specifically crafted. This script's primary function was to sift through the voluminous crash data, identifying each unique crash and quantifying its frequency. This automation was crucial in organizing the crashes into categories based on their uniqueness and the context of their occurrence, which varied across the different fuzzing approaches. The culmination of this process was a meticulous review of the stack traces and error reports associated with each unique crash. This examination was not superficial; it delved into each report's minutiae to unearth the bugs' root causes. By dissecting these reports, insights were gained into the specific conditions under which each crash occurred, including the events leading up to the crash and the precise memory operations implicated. This in-depth analysis was instrumental in identifying and understanding the underlying issues that led to each crash, paving the way for targeted bug fixes and enhancements to Fuzzgoat's stability and security posture.

## Findings and Analysis

The following grouped bar chart shows the overall categorized unique crashes and their occurrences in each mode of fuzzing:




## Analyzing **heap-buffer-overflow** bug:

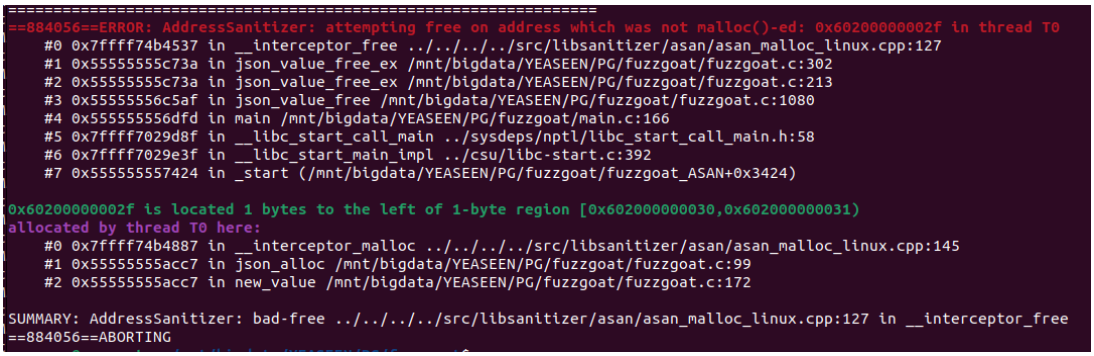



- **Summary:** heap-buffer-overflow
- **Crash Input Description:** a JSON object with a single key-value pair, where both the key and the value are empty strings.
- **Technical Analysis:** The error manifested during a memory access operation at address 0x602000000018, triggered by the __interceptor_puts function during a READ operation involving 9 bytes. This problematic access occurred within the executing code at main.c:150, a call initiated by __libc_start_main_impl. The root cause of this error was a buffer overflow linked to a heap-allocated region of 8 bytes, ranging from 0x602000000010 to 0x602000000018. An attempt was made to access memory exactly at the region's boundary, an action that exceeded the

allocated space. This issue stemmed from the initial memory allocation performed by a call to malloc in main.c:129, setting the stage for the overflow to occur during subsequent operations.

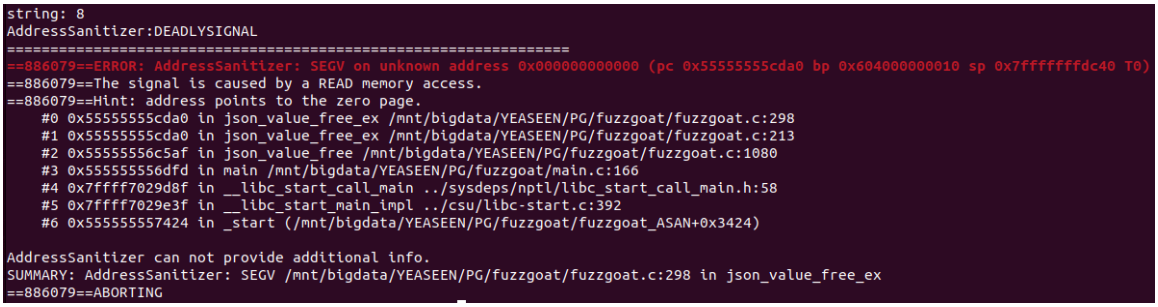
- **Impact Analysis:** This stack trace is telling us that the program attempted to access memory just beyond an allocated heap buffer, leading to a heap-buffer-overflow error. This kind of error is critical because it can lead to undefined behavior, potentially allowing exploitation for arbitrary code execution or causing the program to crash.
- **Reproduction Steps:** A JSON object where the key or value is a null string can cause the error.

## Analyzing **bad-free** bug:



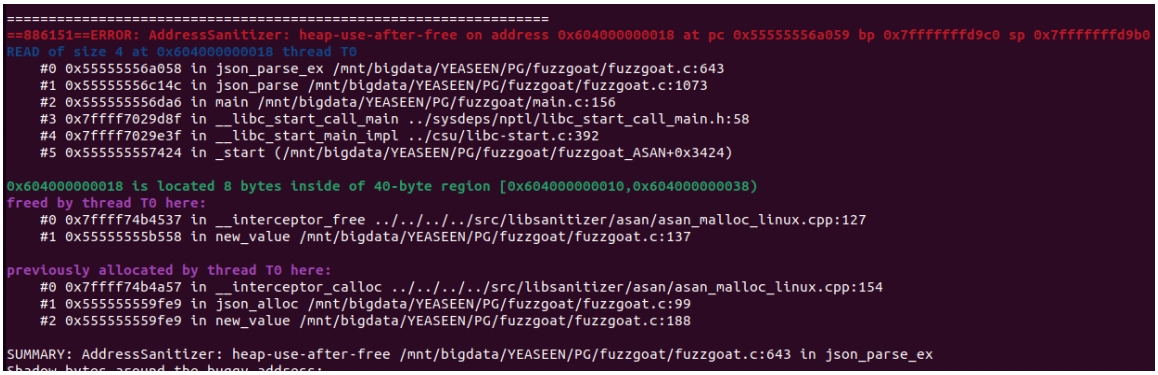

- **Summary:** bad-free
- **Crash Input Description:** A sequence of characters from various Unicode blocks representing symbols and letters from different writing systems or symbolic notations.
- **Technical Analysis:** The free operation causing the error is initiated in json_value_free_ex within fuzzgoat.c at lines 302 and 213, and the attempt to free the unallocated memory is intercepted by ASAN's __interceptor_free.
- **Impact Analysis:** ASAN detected an attempt to free a memory address that does not correspond to a known allocation, indicating a logic error in how memory is managed and freed in the program. Such errors can compromise program stability and security. They necessitate thorough review and correction of the program's memory management routines to prevent undefined behavior or exploitation.
- **Reproduction Steps:** Various characters from different unicode blocks can cause the error.

## Analyzing **SEGV** bug:




- **Summary:** SEGV
- **Crash Input Description:** A string followed by some null characters.
- **Technical Analysis:** Segmentation fault on an unknown address. AddressSanitizer couldn't provide additional information.
- **Impact Analysis:** The output ending with "Segmentation fault" indicates that the program has attempted to access memory that it is not allowed to, leading to a crash. This typically happens due to programming errors such as dereferencing a null or uninitialized pointer, accessing memory out of bounds of allocated structures, or similar issues that violate the access restrictions imposed by the operating system on the program's memory space.
- **Reproduction Steps:** A similar kind of input can cause this error.

## Analyzing **heap-use-after-free** bug:




- **Summary:** heap-use-after-free
- **Crash Input Description:** An empty array followed by some null characters.
- **Technical Analysis:** The error identified as a heap-use-after-free occurred when the program attempted to perform a READ operation of 4 bytes at the address 0x604000000018. This dangerous action can lead to undefined behavior since it involves accessing a heap memory location

that has already been freed. This specific address was accessed by the json_parse_ex function in fuzzgoat.c at line 643, as part of an execution flow that originated from the main function at line 156, which then called json_parse, leading to the problematic access in json_parse_ex. The address in question, 0x604000000018, is situated 8 bytes inside a 40-byte region [0x604000000010,0x604000000038) that was previously allocated but subsequently freed. The initial allocation of this memory region was performed through a call to __interceptor_calloc (an intercepted calloc call by ASAN) within json_alloc in fuzzgoat.c:99 and new_value in fuzzgoat.c:188, and it was later freed by a call to __interceptor_free (intercepted free call by ASAN) in new_value at fuzzgoat.c:137, before being illegally accessed again in json_parse_ex.

- **Impact Analysis:** Use-after-free errors can lead to serious issues, including program instability, data corruption, and potential security exploits. Attackers might exploit such vulnerabilities to execute arbitrary code.
- **Reproduction Steps:** An empty array followed by some characters.

## Conclusion

In conclusion, my analysis of Fuzzgoat application crashes, facilitated by AddressSanitizer (ASAN), revealed significant vulnerabilities due to memory management issues like heap-buffer-overflow, bad-free, use-after-free, and segmentation faults. The integration of ASAN pinpointed error locations and patterns, enabling efficient bug categorization and prioritization through a custom Python script. This report highlights the critical need for rigorous testing, vulnerability assessment, and robust memory management in software development. Addressing these identified vulnerabilities is essential for enhancing application security and stability, underscoring the value of automated tools like ASAN in improving coding standards and testing strategies for more secure software.