Testing ROS with Sanitizers

Hanno Böck

2019-01-07

Contents

1 Introduction

In this project we used compiler sanitizers to test ROS software packages for common C/C++ bugs.

ROS (Robot Operating System) is a framework for robotics. It's mostly written in Python and C++.

The compilers gcc and clang provide sanitizer features that allow detecting common C/C++ bugs. Address Sanitizer [**?**] detects memory safety violations, particularly buffer overflows, buffer overreads and use after free bugs. Undefined Behavior Sanitizer detects code that exposes behavior that is undefined according to the C/C++ standard. Thread Sanitizer detects concurrency issues and race conditions. Memory Sanitizer, which is only available in clang, detects the use of uninitialized memory.

In this project we used these sanitizers to discover bugs in the ROS ecosystem. We focus on ROS version 1, yet the same techniques could be applied to ROS 2.

We discovered 6 errors from undefined behavior sanitizer, 213 errors from thread sanitizer and 18 errors from address sanitizer. We reported some example errors to the ROS project and share our results with the public.

2 Sanitizers

The sanitizers are a direct part of the compilers supporting them and can simply be enabled by passing an appropriate flag to the compiler. The flag is -fsanitize=[sanitizer], where [sanitizer] can be address, undefined, thread or memory.

For the purpose of this project we primarily focused on address sanitizer and undefined behavior sanitizer, because they are the easiest to use. We also ran tests with Tread Sanitizer, but we haven't analyzed them, as the error reports are difficult to interpret without knowledge of the application internals.

Memory sanitizer is much more complex to use in practice. It requires not only the application to be built with it, but also all dependency libraries. Therefore we didn't test with Memory Sanitizer during this project.

3 Test setup 1: Single package compilation with Ubuntu

ROS melodic was the latest version of the ROS system during our testing. We installed ROS on an Ubuntu 18.04 (bionic) system with the ROS apt repository according to the instructions in the ROS wiki [1].

We then built individual packages from their Github repositories with the three sanitizers (address, undefined, thread) and ran their test suites. ROS has its own build system called catkin and the ROS packages are spread over various Github repositories. We checked out all repositories from the ROS group on Github [2] and searched for buildable packages.

To allow others to replicate this we have shared our scripts [3]. Via environment variables the sanitizers allow logging found errors. We also share the log files of the errors we found [4].

During building and testing the packages we discovered 6 errors from undefined behavior sanitizer, 213 errors from thread sanitizer and 18 errors from address sanitizer. Some of the errors are duplicates (i.e. the same code gets called multiple times during a test run). The address sanitizer errors contain 16

[1] `http://wiki.ros.org/melodic/Installation/Ubuntu`
[2] `https://github.com/ros`
[3] `https://github.com/hannob/rosproject-scripts`
[4] `https://github.com/hannob/ros-sanitizer-logs`

memory leaks, which are usually not security sensitive and should be considered low impact.

4 Test setup 2: Gentoo with Address Sanitizer

In a previous project the author of this document created a Gentoo Linux system where all the packages are compiled with Address Sanitizer. [5] This allowed an easy test of the ROS packages available in the Gentoo package manager.

Gentoo is a source-based Linux distribution, meaning that usually all packages are compiled on each system. This makes it suitable for such experiments.

The main difference to the setup above is that compiling all dependencies allows not only detecting bugs in the package currently tested, but also in dependent libraries, which makes the testing more extensive. For example a library may contain code with a bug detectable by Address Sanitizer, but it is not called during its own tests. Yet the code gets indirectly called by a tool using that library.

We found a use after free bug in the ROS package topic_tools [6] that didn't show up in the single package based test in Ubuntu.

While we didn't find many additional bugs a use after free bug is a severe finding that can have security implications.

5 Bug examples

Here we describe two bugs we found with undefined behavior Sanitizer during this project.

5.1 Invalid zero index access on vector

In the xmlrpcpp package we found several places where a C++ vector was converted into a C++ string via the c_str() method and subsequently the address of the index zero was read.

This works fine in all cases where the vector is of non-zero size. But in case of an empty vector this points to invalid memory and reading that address is undefined behavior according to the C++ standard.

Modern variants of C++ allow using the data() method to avoid this issue, but this is not compatible with older versions of the C++ standard.

We provided a patch for this issue [7] that prevents this bug by preventing this pattern with zero-sized vectors.

[5] `https://wiki.gentoo.org/wiki/AddressSanitizer`
[6] `https://github.com/ros/ros_comm/issues/1530`
[7] `https://github.com/ros/ros_comm/pull/1547`

5.2 Calling memcpy with NULL pointer

According to the C standard the memcpy() function can not be called with an invalid pointer, even when the size of the copied space is zero.

We found an instance in the xmlrpcpp code where memcpy sometimes gets called in such a way. We provided a patch that prevents this with a simple check for a zero size. [8]

6 Challenges with the ROS ecosystem

During this project there were noticeable challenges with the ROS ecosystem and its build tools. The complexity and fragility of the ROS ecosystem may be a barrier to the involvement of contributors to ROS.

6.1 Build System

ROS uses CMake as its build system, however it is not intended to be manually run. Instead ROS has its own build system (catkin for ROS 1, ament for ROS 2) built on top of CMake.

This adds complexity and requires new people who want to interact with the ROS ecosystem to learn how to handle a new build system. A standardized build solution would make contributions easier.

6.2 Inconsistent repository structure

The ROS packages are spread over various repositories on Github. However unfortunately the structure is quite confusing.

Some packages have their own repository, some packages are located in a subdirectory of a repository and yet others are grouped in subdirectories.

This inconsistent structure makes it hard to create a list of packages or test them in an automated fashion.

6.3 Fragility of the build system

During the project we also tried to build the complete ROS code manually according to their instructions, [9] but it failed in various ways. Our first compilation attempt failed due to a bug that was already reported, [10] but without a helpful solution.

During our tests we discovered that the ROS build system had problems with unexpected Python versions (it called python expecting it to be python 2, while our test system had python 3 as a default) and with newer CMake versions. [11]

[8] https://github.com/ros/ros_comm/pull/1546
[9] http://wiki.ros.org/melodic/Installation/Source
[10] https://github.com/vcstools/wstool/issues/125
[11] https://bugs.gentoo.org/672396

Overall we got the impression that the ROS build system is very brittle.

6.4 Broken CI

ROS runs a Jenkins-based continuous integration system that automatically runs tests when pull requests are submitted to their Github repositories.

When we submitted a pull request to the ros_comm repository we got a report that their CI checks failed. [12] However these failing checks were not due to our pull request, instead the tests were already failing in the main repository. This indicates that at the time there were several test failures in the main development branch, which shouldn't happen.

While CI systems can be helpful tools to help contributors of a software project, a CI system that is not properly maintained and gives false errors to contributors can have the opposite effect.

6.5 Summary

We found that the complexity and fragility of the build system, failing CI tests and a confusing structure of the ROS software created considerable difficulties in interacting with the ROS ecosystem.

If the ROS project wants to create a community that is welcoming to new contributors it should try to work on these issues. Build system failures should be resolved with high priority. Commits that break the continuous integration tests should never be merged into the master branch of repositories. The complex, custom build system should be replaced with a standardized solution. A consistent structure of the code repositories would be desirable.

[12] `http://build.ros.org/job/Mpr__ros_comm__ubuntu_bionic_amd64/403/`