PROJECT PROPOSAL

FORMULA SAE PROPOSAL

A proposal for a cybersecurity focused Formula SAE project

Master of Science
in
Cybersecurity

Submitted By,

Daniele Giachetto
647820
Department of Information Engineering

Edoardo Caciorgna
578245
Department of Information Engineering

Under the guidance of,
STEFANO CHESSA
PROFESSOR & CHAIR OF THE M.SC. IN CYBERSECURITY,
COMPUTER SCIENCE

DEPARTMENT OF INFORMATION ENGINEERING
Università degli studi di Pisa

Accademic Year: 2023-2024

Contents

THE PROJECT

1.1 PROJECT SCOPE AND GOALS

The current objective of the team is to develop an autonomous vehicle. This project is also called “Driverless” and needs to meet high standards of reliability and security in order for the car to compete and the project to be deemed a success. This directly led to the project scope and goal: allow the team to pursue an increase in the security of the codebase, mainly targeting the development, test and build phases of the software.

1.2 DRIVERLESS CODE STRUCTURE

Before delving in the issues and solutions, we need to look at the code structure to better comprehend the many problems that the team is facing. The main programming languages used are **C, C++** and **Python**. The first two languages have many vulnerabilities related to memory usage and leaks while the latter known for vulnerabilities related to duck typing.

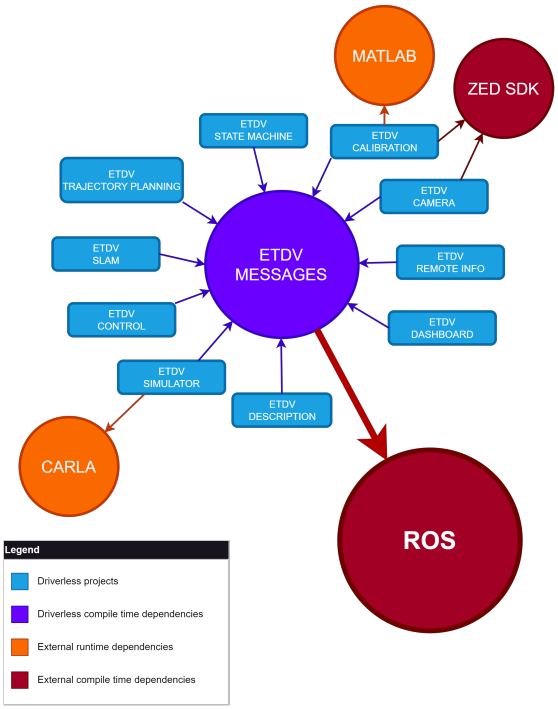

Figure 1: Dependencies graph

1.3 ISSUES FOUND

- **Lack of documentation:** results in fragmented and obscure codebase;
- **Lack of shared code style:** results in high amount of bugs, security issues and bad practices;
- **Different development environments:** every developer has a different environment, without standardization and a clear guide;
- **Lack of tests:** there are no tests, and everything is tested manually;
- **Lack of versioning:** the code is versioned, but the releases are not. This leads to bugs and security hazard when different components interact;

1.4 OBJECTIVES

1.4.1 DEVELOPER GUIDELINES

Documentation is the first step to help a developer ensure high quality code. Those guidelines will contain installation guides and visualization of the "Driverless" project structure.

1.4.2 BUILD ENVIRONMENT

The "Driverless" project builds are complicated to say the least. Every developer needs to install in its own machine all of the dependencies that the current project has as shown in Figure 1. Right now this leads to builds error such as clashing libraries versions and in the production environment there could be runtime dependencies missing.
To solve this problem we need to create an enviroment containing all of the libraries required, so that it can be easily installed, replicated and in which development and builds can be done without issues. Further research on the subject is required before deciding the technology to be used.

1.4.3 TESTS

Currently, all the tests are done either by running the code in the production environment and manually checking the car behaviour or by using the simulator module. This process is dangerous, inefficient, and behaviours cannot be reproduced consistently. It could lead to security issues regarding the production enviroment, human errors and more commonly edge cases not tested. To reduce the vulnerabilities that can be found in the code we aim to introduce three different test methodologies:

- **Unit testing:** testing the smallest component that can be isolated, usually only the body of a function while mocking the external function calls found;
- **Integration testing:** testing a functionality, from the input to the output mocking only calls to external software (like build dependencies);
- **End to end testing:** testing the complete flow of the program, checking if all the components can work together.

1.4.4 CI/CD

Planning a well-tought out continous integration and delivery pipeline will be beneficial to solve all of the previously discussed issues. Currently those are the phases that will be executed:

- **Linting:** used to enforce rules on the codebase. It will be used to statically analyze the code, avoid bugs and possibly introduce rigid static type checkers to scripting languages and memory leak detection for C;
- **Code formatting:** used to automatically format the code following pre-established rules (ex: line length);
- **Test:** unit tests, integration tests and end to end tests will be automatically executed to ensure code quality;
- **Build:** using the standardized build environment, we can automate the building process. Doing so will also allow us to introduce versioning to the releases.