Lab 01 Report

Bruno Luiz D. A. de Castro

January 2024

1 Introduction

During this lab, we were tasked with using different tools available for Linux to try to detect and fix some common venerability issues with given codes. These snippets of code were extracted and adapted from notorious software and were involved in famous exploits found on these programs.

2 The tools

In total, 8 tools were used, divided in two main groups: **Static Analysis** and **Dynamic Analysis**.

2.1 Static Analysis

During static analysis, your code examined either at its source state, or at compilation time. These tools can only report vulnerabilities detectable before running the program.

2.1.1 GCC Warnings

GCC is likely the most famous and used compiler for C and C++ in Unix based system. This compiler can also give the programmer some useful insights on some possible vulnerabilities on compilation time.

To use it, just add the necessary flags to the compiler call, and the warnings will be given, in case any exists.

It's a robust tool, but being static, can only do a limited amount of things, and vulnerabilities are only detected during compilation time.

2.1.2 Clang Warnings

Clang is the main competitor of GCC in the compiler war. In fact, they work in a similar manner, and as GCC, offers a similar tool for warnings during compilation time.

In order to use it, just add the necessary flags to your clang compiler call, and it will output the warnings found during compilation time, if any were.

Just like **GCC Warnings**, it is a fairly robust tool, but can only do so much being static, and limited to compilation time.

2.1.3 cppcheck

This program will scan your source file(s) and output some possible vulnerabilities it found.

Works in a similar way to both **GCC** and **Clang Warnings**, but this one is not a compiler, or associated with one.

It is also available as an extension for multiple IDEs.

2.1.4 GCC Analyzer

Alongside **GCC Warnings**, GCC also offers a Analysis tool, called **GCC Analyzer**. The main difference between both, is that the analyser will only work in your source code, and will not perform any actual compilation.

To use it, use the flag *-fanalyzer* with the regular GCC installation.

2.1.5 Clang Analyzer

It is the Clang alternative to **GCC Analyzer**. It performs static analysis of your source code, and perform no actual compilation.

To run it, install the *clang-tools* package, and call the use the *scan-build* program.

2.2 Dynamic Analysis

These tools are capable of looking for vulnerabilities happening at run time. They are powerful, but adds a significant amount of overhead to your program, and makes it run painfully slower.

2.2.1 GCC Sanitizers

This is a tool included with GCC installations. It will generate a special executable of your program, that can detect certain vulnerabilities at run time.

To use it, add the flag *-fsanitize*, along with the vulnerabilities you wish to look for (such as *leak, address, undefined, ...*), and upon running this program, it will report if any vulnerability is triggered.

2.2.2 Clan Sanitizers

Such as **Clang Warnings** and **Analyzer**, Clang Sanitizers is the Clang alternative to **GCC Sanitizers**.

They work in the exact same way, and to use it you just need to add the vulnerabilities you wish to scan for during run time as a parameter using the *-fsanitize* flag, and the compiler will output a special executable, that you can run.

2.2.3 Valgrind

Valgrind is a Dynamic Analysis tool, capable of detecting many memory management and threading bugs.

In order to use it, install it in your machine, and pass an executable binary as input, the analysis is performed on to of it.

3 Bash Script

In order to automate the process of running all the tools, a bash script was created. It will run all the tools, and output the results to a file.

4 Exercises

4.1 Exercise 1

This exercise includes the code of a simple HTTP Server writen in C++, using the Pistache library.

Due to the inavailability of the library, most of the tools were unable to run.

4.1.1 GCC Warnings

GCC Warnings was not able to run, due to the inavailability of the Pistache library.

Figure 1: GCC Warnings output

4.1.2 Clang Warnings

Similar to GCC Warnings, Clang Warnings was not able to run, due to the inavailability of the Pistache library.

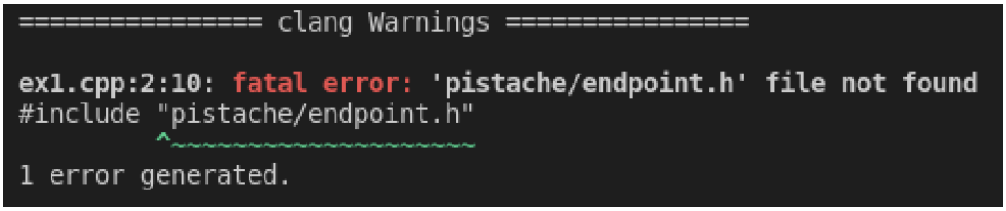

Figure 2: Clang Warnings output

4.1.3 cppcheck

cppcheck was able to find one problem in the code: the function *onRequest* is not used anywhere in the code. This is not a security vulnerability, so nothing needs to be done.

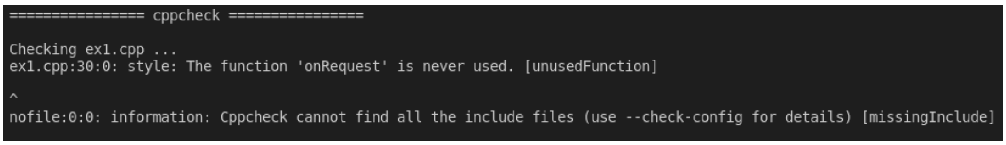

Figure 3: cppcheck output

4.1.4 GCC Analyzer

GCC Analyzer was not able to run, due to the inavailability of the Pistache library. This is a bit surprising, since this tool does not perform any compilation, and should be able to run without the library. Either way, nothing can be done about it.

Figure 4: GCC Analyzer output

4.1.5 Clang Analyzer

Clang Analyzer was also not able to run as well, due to the inavailability of the Pistache library. Similar to GCC Analyzer, this is a bit surprising, since this tool does not perform any compilation, and should be able to run without the library.

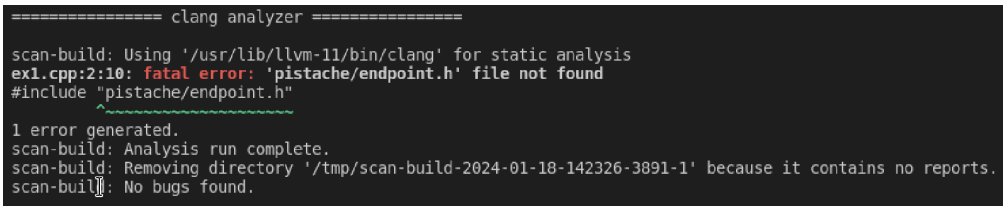

Figure 5: Clang Analyzer output

4.1.6 GCC Sanitizers

GCC Sanitizers was not able to run, due to the inavailability of the Pistache library. As this tool needs to generate a special executable, it is not surprising at all.

Figure 6: GCC Sanitizers output

4.1.7 Clang Sanitizers

Clang Sanitizers was also not able to run, due to the inavailability of the Pistache library. For the same reason as GCC Sanitizers, this is not surprising at all, as it also needs to generate a special executable, and without the library, it is not possible.

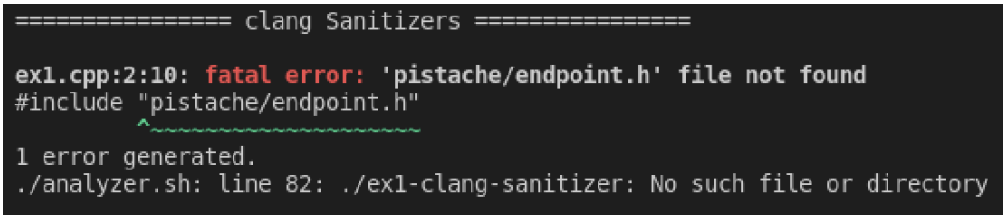

Figure 7: Clang Sanitizers output

4.1.8 Valgrind

Valgrind needs an executable to run, and as the Pistache library is not available, an executable cannot be generated, and thus, Valgrind cannot run.

4.2 Exercise 2

This exercise includes the code simple program, that receives an input, fills a buffer with it, and then prints it.

4.2.1 GCC Warnings

GCC Warnings gave the following warnings as output:

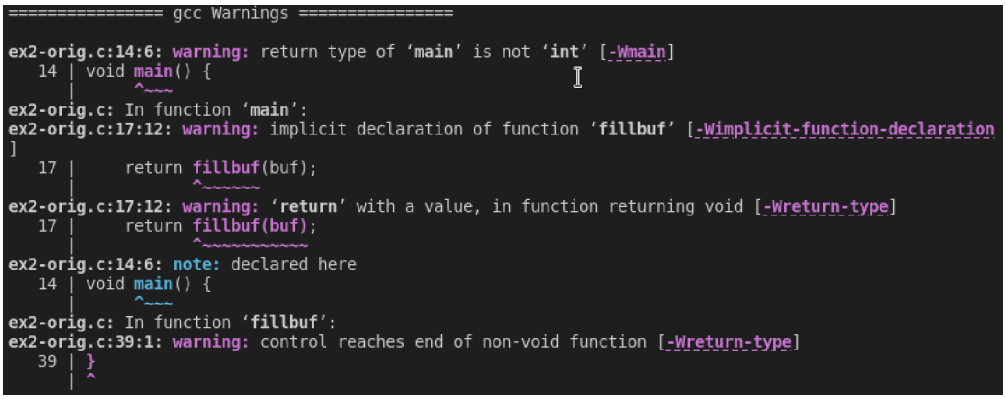

Figure 8: Clang Sanitizers output

- Changed the return type of main to int.
- Changed the order of declaration of the functions.
- Added a return statement to the function *fillbuf*.