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.

1.1 GitHub

All the code used during this lab can be found on the following GitHub Repository.

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.

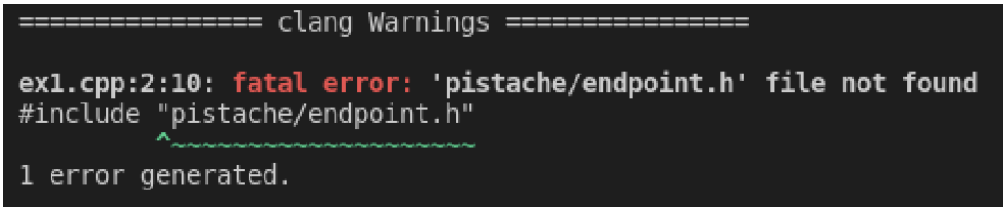

Figure 2: Clang Warnings output

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.

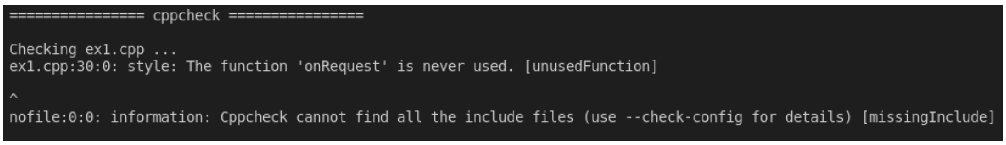

Figure 3: cppcheck output

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.

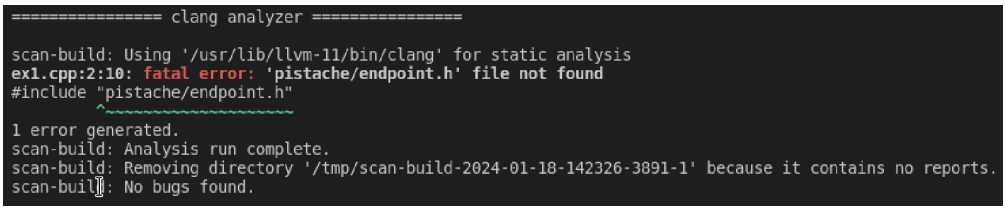

Figure 5: Clang Analyzer output

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.

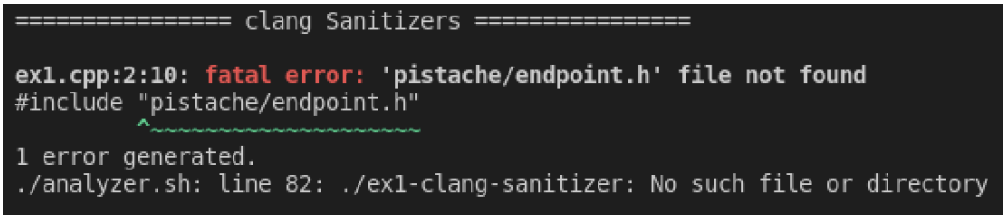

Figure 7: Clang Sanitizers output

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:

```
================ gcc Warnings ================

ex2-orig.c:14:6: warning: return type of ‘main’ is not ‘int’ [-Wmain]
   14 | void main() {
      |      ^~~~
ex2-orig.c: In function ‘main’:
ex2-orig.c:17:12: warning: implicit declaration of function ‘fillbuf’ [-Wimplicit-function-declaration
]
   17 |     return fillbuf(buf);
      |            ^~~~~~
ex2-orig.c:17:12: warning: ‘return’ with a value, in function returning void [-Wreturn-type]
   17 |     return fillbuf(buf);
      |            ^~~~~~~~~~~
ex2-orig.c:14:6: note: declared here
   14 | void main() {
      |      ^~~~
ex2-orig.c: In function ‘fillbuf’:
ex2-orig.c:39:1: warning: control reaches end of non-void function [-Wreturn-type]
   39 | }
      | ^
```

Figure 8: Clang Sanitizers output

The fixes were:

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

After the fixes, GCC Warnings gave no output.

4.2.2 Clang Warnings

Clang Warnings gave the same warnings as GCC Warnings, so no fixes were needed.

4.2.3 cppcheck

cppcheck gave the following output:

```
================ cppcheck ================

Checking ex2-orig.c ...
ex2-orig.c:31:24: warning: Either the condition 'target!=NULL' is redundant or there is possible null pointer derefer
ence: target. [nullPointerRedundantCheck]
        printf("%s\n", target);
                       ^
ex2-orig.c:28:20: note: Assuming that condition 'target!=NULL' is not redundant
       if (target != NULL)
                  ^
ex2-orig.c:31:24: note: Null pointer dereference
        printf("%s\n", target);
                       ^
nofile:0:0: information: Cppcheck cannot find all the include files (use --check-config for details) [missingIncludeS
ystem]
```

Figure 9: cppcheck output

The fixes were:

- Moved all instructions in while to the the *if(taget != NULL)* block.

After the fixes, cppcheck gave no output.

4.2.4 GCC Analyzer

GCC Analyzer gave no output at this point.

4.2.5 Clang Analyzer

Clang Analyzer gave the following output:

```
============== clang analyzer ==============

scan-build: Using '/usr/lib/llvm-11/bin/clang' for static analysis
ex2-orig.c:19:23: warning: Array access (via field 'arg') results in a null pointer dereference [core.NullDereference
]
    (*al)->arg[index] = inputbuf;
           ~~~        ^
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /tmp/scan-build-2024-01-19-221215-4068-1' to examine bug reports.
```

Figure 10: Clang Analyzer output

The fixes were:

- Moved all instructions in while to the the *if(taget != NULL)* block.

After the fixes, Clang Analyzer gave no output.

4.2.6 GCC Sanitizers

GCC Sanitizers gave no output.

4.2.7 Clang Sanitizers

Clang Sanitizers gave no output.

4.2.8 Valgrind

Valgrind gave no output.

4.3 Exercise 3

This exercise includes the code of a simple program, that creates an object, checks for a flag, and delete it.

4.3.1 GCC Warnings

GCC Warnings did not give any output.

4.3.2 Clang Warnings

Clang Warnings found the following warning:

```
================ clang Warnings ================

ex3.cpp:16:10: warning: format string is not a string literal (potentially insecure) [-Wformat-security]
  printf(argv[1]);
         ^~~~~~
ex3.cpp:16:10: note: treat the string as an argument to avoid this
  printf(argv[1]);
         ^
         "%s",
1 warning generated.
```

Figure 11: Clang Warnings output

The fixes were:

- Added a "%s" to the *printf()* call, to avoid a possible attack with "%" in the string.

After the fixes, Clang Warnings gave no output.

4.3.3 cppcheck

cppcheck gave no output.

4.3.4 GCC Analyzer

GCC Analyzer gave no output.

4.3.5 Clang Analyzer

Clang Analyzer gave no output.

4.3.6 GCC Sanitizers

GCC Sanitizers gave no output.

4.3.7 Clang Sanitizers

Clang Sanitizers gave no output.

4.3.8 Valgrind

Valgrind found no possible vulnerabilities.

4.4 Exercise 4

This exercise includes the code of a simple websocket server, that receives a message, and sends it back to the client.

4.4.1 GCC Warnings

GCC Warnings gave the following warnings:

```
================ gcc Warnings ================

ex4.cpp: In function ‘int main(int, char**)’:
ex4.cpp:62:19: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   62 |       char *buf = "hello\n";
      |                   ^~~~~~~~
ex4.cpp:8:14: warning: unused parameter ‘args’ [-Wunused-parameter]
    8 | int main(int args, char** argv) {
      |          ~~~~^~~~
ex4.cpp:8:27: warning: unused parameter ‘argv’ [-Wunused-parameter]
    8 | int main(int args, char** argv) {
      |                    ~~~~~~~^~~~
```

Figure 12: GCC Warnings output

The fixes were:

- Removed the unused args from the *main()* function.
- Changed the *buf* variable to a const.

After the fixes, GCC Warnings gave no output.

4.4.2 Clang Warnings

Clang Warnings gave no warnings.

4.4.3 cppcheck

cppcheck gave the following output:

```
================ cppcheck ================

Checking ex4.cpp ...
ex4.cpp:44:7: style: The scope of the variable 'cli' can be reduced. [variableScope]
  int cli;
      ^
```

Figure 13: cppcheck output

The fixes were:

- Changed the scope of the *int cli* variable.

After the fixes, cppcheck gave no output.

4.4.4 GCC Analyzer

GCC Analyzer gave the following warning:

```
================ gcc analyzer ================

ex4.cpp: In function 'int main()':
ex4.cpp:54:13: warning: use of possibly-NULL 'recvBuf' where non-null expected [CWE-690] [-Wanalyzer-possible-null-ar
gument]
   54 |       memset(recvBuf, 0x00, 1024);
      |       ~~~~~~^~~~~~~~~~~~~~~~~~~~~
```

Figure 14: GCC Analyzer output

The fixes were:

- Added a guard clause to the do while loop.

After the fixes, GCC Analyzer gave no output.

4.4.5 Clang Analyzer

Clang Analyzer gave no warnings.

4.4.6 GCC Sanitizers

GCC Sanitizers gave no output.

4.4.7 Clang Sanitizers

Clang Sanitizers gave no output.

4.4.8 Valgrind

Valgrind found no vulnerabilities.

4.5 Exercise 5

This exercise includes the code of a program that pretends to copy a fake file.

4.5.1 GCC Warnings

GCC Warnings gave the following warnings:

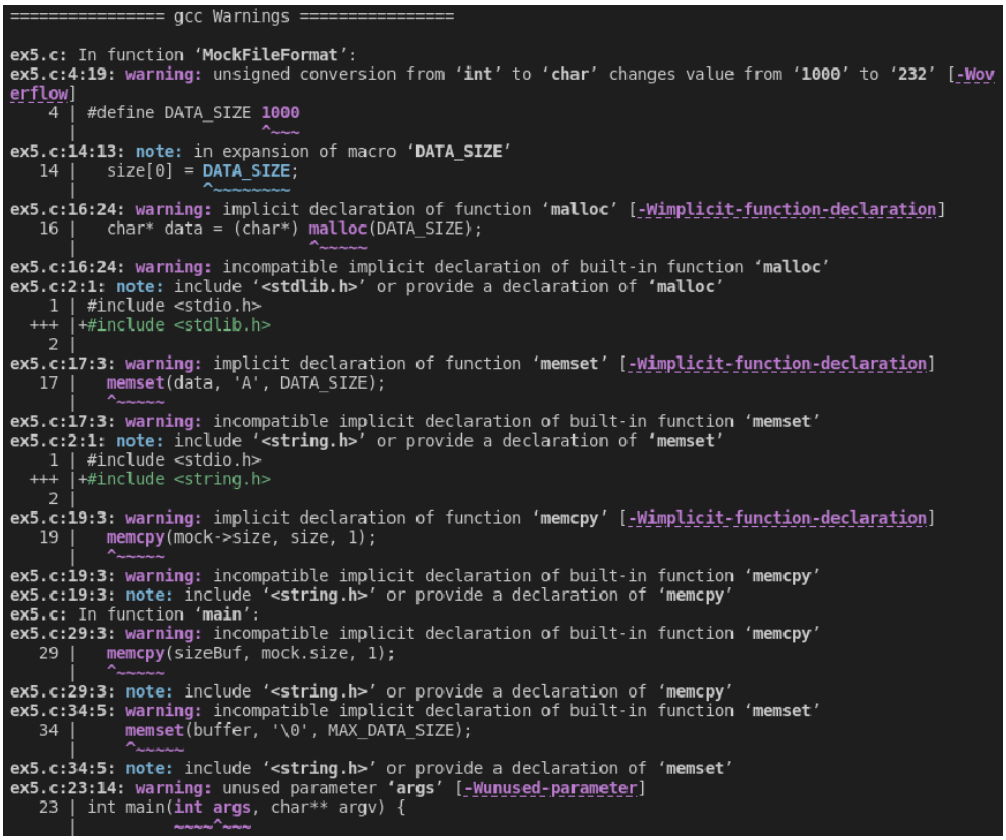

Figure 15: GCC Warnings output

The fixes were:

- Changed the *size* array type to int.
- Added an include to the *stdlib.h* library.
- Added an include to the *string.h* library.
- Removed the unused *argc* and *argv* args from the *main()* function.

After the fixes, GCC Warnings gave no output.

4.5.2 Clang Warnings

Clang Warnings gave the same warnings as GCC Warnings, so no fixes were needed.

4.5.3 cppcheck

cppcheck gave no warnings.

4.5.4 GCC Analyzer

GCC Analyzer gave the following warning:
The fixes were:

```
================ gcc analyzer ================

ex5.c: In function ‘MockFileFormat’:
ex5.c:19:3: warning: use of possibly-NULL ‘data’ where non-null expected [CWE-690] [-Wanalyzer-possibl
e-null-argument]
   19 |   memset(data, 'A', DATA_SIZE);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 16: GCC Analyzer output

- Added a guard clause to the *data* variable.

After the fixes, GCC Analyzer gave no output.

4.5.5 Clang Analyzer

Clang Analyzer gave the following warning:

```
================ clang analyzer ================

scan-build: Using '/usr/lib/llvm-11/bin/clang' for static analysis
ex5.c:40:5: warning: 2nd function call argument is an uninitialized value [core.CallAndMessage]
    memcpy(buffer, mock.data, size);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /tmp/scan-build-2024-01-19-231229-9711-1' to examine bug reports.
```

Figure 17: Clang Analyzer output

The fixes were:

- Added a chack for the *data* variable in *mock* object.

After the fixes, Clang Analyzer gave no output.

4.5.6 GCC Sanitizers

GCC Sanitizers detected a memory leak:

```
================ gcc Sanitizers ================

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA

=================================================================
==9903==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1000 byte(s) in 1 object(s) allocated from:
    #0 0xffffa906ae7c in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:
145
    #1 0xaaaac582142c in MockFileFormat /home/bruno/Documents/ESIEE-secure-c-and-cpp/TP1/code/ex5.c:18
    #2 0xaaaac58210f8 in main /home/bruno/Documents/ESIEE-secure-c-and-cpp/TP1/code/ex5.c:33
    #3 0xffffa84c5e14 in __libc_start_main ../csu/libc-start.c:308
    #4 0xaaaac5821298  (/tmp/ex5-gcc-sanitizer+0x1298)

SUMMARY: AddressSanitizer: 1000 byte(s) leaked in 1 allocation(s).
```

Figure 18: GCC Sanitizers output

The fixes were:

- Freed the *mock.data* variable in the end of the program.

After the fixes, GCC Sanitizers gave no output.

4.5.7 Clang Sanitizers

Clang Sanitizers detected the same memory leak as GCC Sanitizers, so no fixes were needed.

4.5.8 Valgrind

Valgrind detected the same memory leak as GCC Sanitizers, so no fixes were needed.

4.6 Exercise 6

This exercise includes the code of a program with a simple Object class with a constructor and a destructor.

4.6.1 GCC Warnings

GCC Warnings gave the following warnings:

```
================ gcc Warnings ================

ex6.cpp: In function 'void test(Object)':
ex6.cpp:21:18: warning: unused parameter 'obj' [-Wunused-parameter]
   21 | void test(Object obj) {
      |           ~~~~~~~^~~
ex6.cpp: In function 'int main(int, char**)':
ex6.cpp:25:14: warning: unused parameter 'args' [-Wunused-parameter]
   25 | int main(int args, char** argv) {
      |          ~~~~^~~~
ex6.cpp:25:27: warning: unused parameter 'argv' [-Wunused-parameter]
   25 | int main(int args, char** argv) {
      |                    ~~~~~~~^~~~
```

Figure 19: GCC Warnings output

The fixes were:

- Removed the unused *argc* and *argv* args from the *main()* function.
- Removed the *obj* argument from the *test()* function.

After the fixes, GCC Warnings gave no output.

4.6.2 Clang Warnings

Clang Warnings gave the same warnings as GCC Warnings, so no fixes were needed.

4.6.3 cppcheck

cppcheck gave the following output:

```
=============== cppcheck ===============

Checking ex6.cpp ...
ex6.cpp:10:5: warning: Class 'Object' does not have a copy constructor which is recommended since it h
as dynamic memory/resource allocation(s). [noCopyConstructor]
    buf = malloc(1024);
    ^
ex6.cpp:10:5: warning: Class 'Object' does not have a operator= which is recommended since it has dyna
mic memory/resource allocation(s). [noOperatorEq]
    buf = malloc(1024);
    ^
```

Figure 20: cppcheck output

The fixes were:

- Added a copy constructor to the *Object* class.
- Added an operator= to the *Object* class.

After the fixes, cppcheck gave no output.

4.6.4 GCC Analyzer

GCC Analyzer found no warnings.

4.6.5 Clang Analyzer

Clang Analyzer found no warnings.

4.6.6 GCC Sanitizers

GCC Sanitizers found no vulnerabilities.

4.6.7 Clang Sanitizers

Clang Sanitizers found no vulnerabilities.

4.6.8 Valgrind

Valgrind found no vulnerabilities.

4.7 Exercises 7 and 8

These exercises were impossible to be done in the platform I was working on. Exercise 7 had some compiler issues, and Exercise 8 lacks the *Windows.h* library, which is not available on Linux.

5 Conclusion

During this lab, we were able to use some tools to detect some commos vulnerabilities in code. Some vulnerabilities are easy to detect, and can be done in compile time. Others require a more direct approch. It is extremely important to know how and when to use each tools, and how to fix the vulnerabilities they find.