Static code analysis report

Francesco Paolo Di Lorenzo
student ID: 1712990
e-mail: dilorenzo.1712990@studenti.uniroma1.it

Introduction

In this report, a static analysis of a C code fragment is performed using tools such as Splint and Flawfinder.
In the first section there is a high-level description of the tools,which indicates their main strengths and weaknesses.
The second section, shows the output of the respective tools (mainly vulnerabilities and problems) and the resolution of the latter.
The last section presents the correct version of the program obtained by solving the problems reported with the analysis.

1 Static Analysis tools

This section describes main strenghts and weaknesses of Flawfinder and Splint.

1.1 Flawfinder

Flawfinder is a tool for statically scanning C/C++ source code for `possible security weaknesses`. These security weaknesses are called *flaws* or *hits* and are ordered by risk level.
The risk level is shown in square brackets and can take value ranging from 0 (very little risk) to 5(high risk)[1].
Furthermore it is compatible with CWE (Common Weakness Enumeration)[4][3] and may detect many of the most widespread and critical errors drafted in the 2011 CWE/SANS Top 25 list.
Flawfinder is a simple and easy to use tool. This involves some pros and cons.[1]

Unlike programs such as Splint or gcc's warning flags, Flawfinder has no access to the program control flow, data flow and data type when looking for vulnerabilities. This leads the program to produce false positives or fail to report some vulnerabilities. In his favor, instead, we have that he can also analyze programs that cannot be compiled,in a fast and efficient way.

1.2 Splint

Splint is a tool for statically checking C programs for `possible security vulnerabilities and coding mistakes`.[5]
It is very useful for checking type,checking of variable and function assignments, efficiency, unused variables and function identifiers,unreachable code and possible memory leaks.
Splint is a very light static analysis tool, it helps to improve the quality of the code, even if it does not help to eliminate all the security flaws and produces many warnings that can lead to confusion[6].

2 Output description

This section describes the outputs of the respective tools and shows how starting from these outputs, it is possible to improve the code and free it from vulnerabilities and errors that can lead to serious problems.

As mentioned in the previous section, one of the peculiarities of Flawfinder is that it can perform analysis even on fragments of C / C ++ code that cannot be compiled. The fragment available for this analysis is a fragment of text in which there is a C code. Splint does not have the same peculiarity as Flawfinder and causes problems with files that do not have a C extension and are not written so that they can be compiled.
For simplicity, the fragment has been modified into a C code fragment with a *.c* extension and has been compiled after small modifications, so it was possible to analyze the *.c* fragment without problems even with Splint.

2.1 Flawfinder output

Running the program with this command:

```
$ flawfinder fragment.c
```

You get the following result:

```
Flawfinder version 2.0.10, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examining fragment.c

FINAL RESULTS:

fragment.c:55:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination [MS-↩
      banned]
  (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: ↩
      strncpy
  easily misused).
fragment.c:9:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to ↩
      potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, ↩
      use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
fragment.c:16:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to ↩
      potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, ↩
      use
```

```
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
fragment.c:18:  [2] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination
  [MS-banned] (CWE-120). Consider using strcat_s, strncat, strlcat, or
  snprintf (warning: strncat is easily misused). Risk is low because the
  source is a constant string.
fragment.c:17:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for ↩
      invalid
  pointers [MS-banned] (CWE-120).
fragment.c:27:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:29:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:39:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:46:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).

ANALYSIS SUMMARY:

Hits = 9
Lines analyzed = 61 in approximately 0.04 seconds (1686 lines/second)
Physical Source Lines of Code (SLOC) = 47
Hits@level = [0]   1 [1]   5 [2]   3 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]  10 [1+]   9 [2+]   4 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 212.766 [1+] 191.489 [2+] 85.1064 [3+] 21.2766 ↩
    [4+] 21.2766 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
```

The output of Flawfinder is basically divided into 2 parts. In the first part called *Final Results* shows and describes all the *hits* found at the end of the static analysis.
In the second part instead shows the number of these hits, the number of hits that belong to a certain level of risk and information on the time taken to analyze the fragment lines.
Furthermore, Flawfinder reminds that not all of these hits must necessarily represent vulnerabilities, stating that some of them may be false positives: find out what, programmer's job is.
So the fragment has nine hits,let's describe them one by one and try to understand if they actually represent a vulnerability or a false positive.
In the first case, it is shown how to solve the vulnerability to obtain a more secure code.

Hit No.1 (Risk level 4)

```
fragment.c:55:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination [MS-↩
      banned]
  (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: ↩
      strncpy
  easily misused).
```

On line 55 of the fragment, the `strcpy` function does not check for buffer overflows when copying to the destination.This vulnerability is assigned the *CWE-120 weakness ID*, where *CWE* stands for *Common Weakness Enumaration*[3].
Looking at the documentation related to the strcpy function[8], it is possible to see how this function is not safe:

```
char * strcpy ( char * destination, const char * source );
```
Copies the C string pointed by source into the array pointed by destination, including the terminating null character (and stopping at that point).

strcpy does not specify the size of the destination array and this is very dangerous because if the destination array is not large enough to accommodate the source string, this will cause a *buffer overflow*.
So Flawfinder suggests using other more secure functions like snprintf, strcpy_s, strlcpy, strncpy. All these functions allow you to enter the size of the destination array, unlike strcpy.
strncpy is poorly performing and less secure than the proposed functions.
The problem with strncpy is that it does not add the terminator character and strings without the terminator character can cause *segmentation fault* [7].
snprintf, on the other hand, always adds the NULL terminator character,but in some older systems, its implementation is subject to the *buffer overflow*[9].
So the choice fell on the **strlcpy** function: safer and more performant [9] and always NULL-terminated,but not a standard C function.

How to use strlcpy function:
1) install the library: `sudo apt-get install libbsd-dev`
2) add the header: `#include <bsd/string.h>`
3) compile with `-lbsd` flag

```
int  main()
{
  char *foo = "↩
     fooooooooooooooooooooooooooooooooooooooooooooooooooooooooo↩
     ";
  char *buffer = (char *)malloc(10 * sizeof(char));
  strcpy(buffer, foo);
  func1();
  func3(sizeof(*foo));
}
```

```
51 int  main()
52 {
53   char *foo = "↩
        fooooooooooooooooooooooooooooooooooooooooooooooooooooooooo↩
        ";
54   char *buffer = (char *)malloc(10 * sizeof(char));
55   strcpy(buffer, foo);
56   func1();
57   func3(sizeof(*foo));
58 }
```

2.2 Splint output

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent elementum aliquam massa, eget vestibulum arcu porta lacinia. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et turpis dignissim, viverra arcu id, eleifend risus. Nulla facilisi. Sed venenatis placerat nibh, vel vehicula mi molestie a. Donec porttitor pharetra tortor id sodales. In ultricies, ex quis aliquet laoreet, sapien est pulvinar elit, vitae gravida metus lorem et mi. Nulla lacus ante, feugiat a diam at, sodales rutrum sapien. Nam sit amet odio nisl. Sed in auctor metus. Fusce viverra aliquet metus a condimentum.

3 Corrected version of the fragment

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent elementum aliquam massa, eget vestibulum arcu porta lacinia. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et turpis dignissim, viverra arcu id, eleifend risus. Nulla facilisi. Sed venenatis placerat nibh, vel vehicula mi molestie a. Donec porttitor pharetra tortor id sodales. In ultricies, ex quis aliquet laoreet, sapien est pulvinar elit, vitae gravida metus lorem et mi. Nulla lacus ante, feugiat a diam at, sodales rutrum sapien. Nam sit amet odio nisl. Sed in auctor metus. Fusce viverra aliquet metus a condimentum.

4 Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent elementum aliquam massa, eget vestibulum arcu porta lacinia. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et turpis dignissim, viverra arcu id, eleifend risus. Nulla facilisi. Sed venenatis placerat nibh, vel vehicula mi molestie a. Donec porttitor pharetra tortor id sodales. In ultricies, ex quis aliquet laoreet, sapien est pulvinar elit, vitae gravida metus lorem et mi. Nulla lacus ante, feugiat a diam at, sodales rutrum sapien. Nam sit amet odio nisl. Sed in auctor metus. Fusce viverra aliquet metus a condimentum.

References

[1] Flawfinder official documentation,
`https://dwheeler.com/flawfinder/flawfinder.pdf`

[2] The MITRE Corporation. *Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code*. CWE(Common Weakness Enumeration)[3].

[3] CWE (Common Weakness Enumeration),
`https://cwe.mitre.org/index.html`

[4] Rahma Mahmood, Qusay H. Mahmoud. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. Department of Electrical, Computer & Software Engineering,University of Ontario Institute of Technology,Oshawa, ON, Canada.

[5] Splint official page,
`https://splint.org/`

[6] Pedro pereira, Ulisses Costa. *Splint the C code static checker*. Formal Methods in Software Engineering, May 28, 2009

[7] Why strcpy and strncpy are not safe to use,
`https://www.geeksforgeeks.org/why-strcpy-and-strncpy-are-not-safe-to-use/`

[8] strcpy in cpluspluc reference,
`http://www.cplusplus.com/reference/cstring/strcpy/`

[9] Secure Programming for Linux and Unix HOWTO,Chapter 6. Avoid Buffer Overflow
`https://dwheeler.com/secure-programs/3.50/Secure-Programs-HOWTO/dangers-c.html`