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)[5][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`.[6]
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[7].

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+]
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 destination.This vulnerability is assigned the *CWE-120 weakness ID*, where *CWE* stands for *Common Weakness Enumaration*[3].

Portion of the fragment code, affected by the possible vulnerability:

```
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 }
```

Looking at the documentation related to the strcpy function[9], 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 if there is no null character among the first n character of the source, the string placed in destination will not be null-terminated.So strncpy() does not guarantee that the destination string

will be NULL terminated.
Strings without the terminator character can cause *segmentation fault* [8].
snprintf, on the other hand, always adds the NULL terminator character,but in some older systems, its implementation is subject to the *buffer overflow*[10].
So the choice fell on the **strlcpy** function: safer and more performant [10] 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

Based on the previous information, this is the portion of the fragment where the vulnerability has been removed:

```
51 int  main()
52 {
53   char *foo = "↩
       fooooooooooooooooooooooooooooooooooooooooooooooooooooooooo↩
       ";
54   char *buffer = (char *)malloc(10 * sizeof(char));
55   strlcpy(buffer,foo,sizeof(buffer)); //hit no.1
56   func1();
57   func3(sizeof(*foo));
58 }
```

Hits No.2,No.3,No.4 (Risk level 2), **No.5** (Risk level 1)

```
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.
```

These two hits describe the same problem:There are in the code,two statically-sized arrays that can be improperly restricted, leading to potential overflows or other issues.This vulnerability is assigned the *CWE-119!/CWE-20 weakness ID*[3]
Flawfinder suggests performing a bounds checking and other actions to avoid possible overflows.

```
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.
```

This hit is very similar to the hit number 1 related to the strcpy function.
On line 18 of the fragment,the `strcat` function does not check for buffer overflows when concatenating to destination.This vulnerability is assigned the *CWE-120 weakness ID*, where *CWE* stands for *Common Weakness Enumaration*[3].

```
fragment.c:17:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
```

Flawfinder reminds you that using the strcpy function can be dangerous. Here the same considerations made for the hit No.1 are valid: strncpy() does not guarantee that the destination string will be NULL terminated [8]

Portion of the fragment code, affected by the possible vulnerabilities:

```
7  void func1()
8  {
9    char buffer[1024]; // hit no.2
10   printf("Please enter your user id :");
11   fgets(buffer, 1024, stdin);
12
13   if (!isalpha(buffer[0]))
14   {
15
16       char errormsg[1044]; //hit no.3
17       strncpy(errormsg, buffer,1024); //hit.no.4
18       strcat(errormsg, " is not  a valid ID"); //hit no.5
19   }
20 }
```

The hit no.2 is a **false positive**: the fgets function reads at most 1024 from the standard input and stores them into `buffer`,`buffer` has been statically allocated with 1024 bytes which are enough to not cause any overflow.

The hits no.3, no.4, no.5 are also a **false positive**: `errormsg` is statically allocated with enough bytes to contain the contents of the buffer and the constant string *"is not a valid ID"*.
Furthermore strncpy guarantees the NULL terminator character, since fgets always adds the NULL terminator character.

Hits No.6,No.7 (Risk level 1)

```
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).
```

These hits concern the use of the read function. This vulnerability is assigned the *CWE-120,CWE-20 weakness IDs*.
In particular the weakness id 20 concerns an improper validation of the input.[3]

CWE-20: Improper Input Validation
Extended description: when software does not validate input properly, an

attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution [4].

Portion of the fragment code, affected by the possibile vulnerabilities:

```
23 void func2(int f2d)
24 {
25   char *buf2;
26   size_t len;
27   read(f2d, &len, sizeof(len)); //hit no.6
28   buf2 = malloc(len+1);
29   read(f2d, buf2, len); //hit no.7
30   buf2[len] = '\0';
31
32 }
```

The hit no.6 is a **false positive**.
The read function correctly reads sizeof(len) bytes from `f2d` and stores them in `len` which is large enough to accept them.
The main problem is caused by the fact that there is no validation of the input: an attacker can supply a large value of `len` which overflows to zero (**integer overflow**) on line 28,which will cause a **buffer overflow** in the next line, since the read function uses the original value of len.
How is it possible?
The variable `len` is of type `size_t`: size_t is an unsigned integral data type,means that it can only take non-negative values.
An attacker can supply the **maximum possible value** for size_t,causing an interger overflow on line 28, indeed assuming that MAX_VALUE represents the maximum value that size_t can assume,we have that MAX_VALUE +1 = 0 and that's exactly what happens on line 28 in this case.

Based on the previous information, this is the portion of the fragment where the vulnerabilities has been removed:

```
23 void func2(int f2d)
24 {
25   char *buf2;
26   size_t len;
27   size_t limit = 1024;
28   read(f2d, &len, sizeof(len)); //hit no.6
29   if(len > limit) return;
30   buf2 = malloc(len+1);
31   read(f2d, buf2, len); // hit.no.7
32   buf2[len] = '\0';
33
34 }
```

To avoid the interger overflow and the consequent and possible buffer overflow and to resolve the vulnerability shown by the hit no.7, a variable called `limit` is used to limit the size of `len`.
This way the `read` function can never cause buffer overflows, since the destination buffer is sufficiently large (len +1).
Finally it is not even necessary to check that the variable `len` can assume negative values, since the type of data `size_t` can only assume **non-negative values**.

Hits No.8,No.9 (Risk level 1)

```
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).
```

Like the previous hits, the latter also concern the use of the read function,so the considerations concerning the CWE-20 weakness ID also apply in this case.

Portion of the fragment code, affected by the possibile vulnerabilities:

```
35 void func3(int f3d)
36 {
37   char *buf3;
38   int len;
39   read(f3d, &len, sizeof(len)); // hit no.8
40   if (len > 8000)
41   {
42     perror("too large length");
43     return;
44   }
45   buf3 = malloc(len);
46   read(f3d, buf3,len);  //hit no.9
47 }
```

The hit no.8 is a **false positive**.
The `read` function correctly reads sizeof(len) bytes from `f2d` and stores them in `len` which is large enough to accept them.
Also here the main problem is caused by an improper validation of the input. This function learns from the previous errors and tries to validate the input by checking that `len` value is not greater than a set value(avoid integer overflow).
In this case the variable `len` is not of type `size_t`, but of type `int` that represents *signed* integers.
An attacker can supply a negative value for `len`,that will be converted to a large positive value when it gets cast to an unsigned integer on line 46 (hit no.9).

Based on the previous information, this is the portion of the fragment where the vulnerabilities has been removed:

```
35 void func3(int f3d)
36 {
37   char *buf3;
38   int len;
39   read(f3d, &len, sizeof(len));
40   if (len < 0 || len > 8000)
41   {
42     perror("too large length");
43     return;
44   }
45   buf3 = malloc(len+1);
46   read(f3d, buf3,len);
47   buf3[len] = '\0';
48 }
```

2.2 Splint output

3 Corrected version of the fragment

4 Conclusion

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] CWE (Common Weakness Enumeration),CWE-20,
`https://cwe.mitre.org/data/definitions/20.html`

[5] Rahma Mahmood, Qusay H. Mahmoud. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. Department of Electrical, Computer & Soft-

ware Engineering,University of Ontario Institute of Technology,Oshawa, ON, Canada.

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

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

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

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

[10] 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