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 void  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*[**?**].
So the choice fell on the **strlcpy** function: safer and more performant [**?**] 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 void main()
52 {
53   char *foo = "↩
       fooooooooooooooooooooooooooooooooooooooooooooooooooooo↩
       ";
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 **these** 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),causing a buffer overflow.

Based on the previous information, this is the portion of the fragment where **these** 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 }
```

A further check on the non-negativity of `len` is necessary to solve the vulnerability shown by hit no.9

2.2 Splint output

The study of splint output focuses mainly on those warnings that can be considered weakness or vulnerability and cause serious security problems.

To run Splint on the fragment you need to run this command:

```
$ splint fragment.c
```

Since Splint produces many warnings,some of them are just false positives, others are real vulnerabilities to be solved.
The explanation of these will take place by analyzing the warnings of each single method of the program (`main()`, `func1()`, `func2()`,`func3()`) respectively.

main()

```
fragment.c:54:17: Storage buffer may become null
```

```
fragment.c:55:9: Possibly null storage buffer passed as non-null param:
                   strcpy (buffer, ...)
```

On line 54 of the fragment, the code does not check if the buffer is NULL.
The same buffer is passed in the next line to the `strcpy` function.
This is a vulnerability with a *CWE-476 weakness ID*: NULL Pointer Dereference [10].

```
fragment.c:58:2: Fresh storage buffer not released before return
```

The memory allocated to a buffer is not realeased before return.
This is a vulnerability with a *CWE-772 weakness ID*: NULL Pointer Dereference [10].

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

```
51 void  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 }
```

Regarding vulnerability CWE-476,a NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit. Generally this type of vulnerability is related to the reliability of the software, but in some cases if an opponent is able to make this type of vulnerability happen,he might be able to use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks [13].
Regarding vulnerability CWE-772,when a resource is not released after use,causes a memory leak.
A memory leak is an unintentional form of memory consumption whereby the developer fails to free an allocated block of memory when no longer needed [14]. it can allow attackers to cause a denial of service by causing the allocation of resources without triggering their release[12].

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

```
86  int  main()
87  {
88    char *foo = "↩
          fooooooooooooooooooooooooooooooooooooooooooooooooooooo↩
          ";
89
90    char *buffer = (char *)malloc(10 * sizeof(char));
91    if(buffer==NULL)
92    {
93        /*error handling*/
94            return -1;
95    }
96    strcpy(buffer, foo);
97    func1();
98
99    func3(sizeof(*foo));
100
101   free(buffer);
102   return 0;
103 }
```

To solve the NULL pointer deference, always check the return value of the malloc function and always release a buffer with the `free()` function to avoid memory leaks.

func1()

```
fragment.c:11:3: Return value (type char *) ignored: fgets(buffer, 10...
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (Use -retvalother to inhibit warning
```

The return value of the fgets function is ignored. Ignoring the return value of functions is a potential vulnerability. The *CWE-252 weakness ID* [11] describes in detail what could happen to a program in which the return values of the functions are not checked.
If an attacker can force the function to fail or otherwise return a value that is not expected, then the subsequent program logic could lead to a vulnerability, because the software is not in a state that the programmer assumes.
So it's definitely a good practice to always do these kind of checks.

Portion of the fragment code, affected by the weakness:

```
7  void func1()
8  {
9    char buffer[1024];
10   printf("Please enter your user id :");
11   fgets(buffer, 1024, stdin);
12
13   if (!isalpha(buffer[0]))
14   {
15
16      char errormsg[1044];
17      strncpy(errormsg, buffer,1024);
18      strcat(errormsg, " is not  a valid ID");
19   }
20 }
```

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

```
7  void func1()
8  {
9    char buffer[1024];
10   printf("Please enter your user id :");
11  if(fgets(buffer, 1024, stdin)!=NULL){
12
13   if (!isalpha(buffer[0]))
14   {
15
16      char errormsg[1044];
17      strncpy(errormsg, buffer,1024);
18      strcat(errormsg, " is not  a valid ID");
19   }
20  }
21  else
22  {
23   /* error handling */
24  }
25 }
```

Now the return value of the fgets function is checked and in case of errors it is possible to handle the error.

func2()

```
fragment.c:27:3: Return value (type ssize_t) ignored: read(f2d, &len, ...
```

```
fragment.c:29:3: Return value (type ssize_t) ignored: read(f2d, buf2, len)
```

These warnings are the same warning already analyzed and solved for the func1 function, the one with CWE-252 wakness id. [11].

```
 fragment.c:28:10: Storage buf2 may become null
```

```
fragment.c:32:2: Fresh storage buf2 not released before return
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
```

These two are the same warnings already analyzed and solved for the `main()` function, those with *CWE-476* [10], *CWE-772* [12] *waekness ids*, respectively. 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));
28   buf2 = malloc(len+1);
29   read(f2d, buf2, len);
30   buf2[len] = '\0';
31
32 }
```

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

```
23 void func2(int f2d)
24 {
25   char *buf2 = NULL;;
26   size_t len = 0;
27   ssize_t ret = 0;
28   if((ret = read(f2d, &len, sizeof(len)))> 0)
29   {
30        buf2 = calloc(len+1,sizeof(char));
31        if(buf2 == NULL)
32        {
33          /*error handling */
34          return;
35        }
36        if((ret = read(f2d,buf2, len))>0)
37        {
38          buf2[ret] = '\0';
39
40        }
41        else
42        {
43           free(buf2);
44            /*error handling */
45           return;
46        }
47   }
48   else
49   {
50     /* error handling */
51   }
52   free(buf2);
53
54 }
```

func3()

```
fragment.c:39:3: Return value (type ssize_t) ignored: read(f3d, &len, ...

fragment.c:46:3: Return value (type ssize_t) ignored: read(f3d, buf3, len)

fragment.c:47:2: Fresh storage buf3 not released before return
```

These warnings represent weaknesses / vulnerabilities already encountered with analyzing the previous functions Portion of the fragment code, affected by the possibile vulnerabilities:

```
51 void func3(int f3d)
52 {
53   char *buf3;
54   int len;
55   read(f3d, &len, sizeof(len));
56   if (len > 8000)
57   {
58     perror("too large length");
59     return;
60   }
61   buf3 = malloc(len);
62   read(f3d, buf3,len);
63 }
```

Based on the previous information, this is the portion of the fragment where **these** vulnerabilities and weaknesses have been removed:

```
51 static void func3(int f3d)
52 {
53
54
55 char *buf3;
56   int len = 0;
57   ssize_t ret = 0;
58   if((ret=read(f3d, &len, sizeof(len)))<0){
59         /*error_handling*/
60   return;
61   }
62   if (len > 8000)
63   {
64     perror("too large length");
65     return;
66   }
67   buf3 = calloc((size_t)len,sizeof(char));
68   if(buf3==NULL)
69   {
70      /*error_handling*/
71      return;
72   }
73   if((ret=read(f3d, buf3,(size_t)len))<0)
74   {
75      /*error handling */
76      free(buf3);
77      return;
78   }
79   free(buf3);
80 }
```

`note`: Not all warnings have been taken into consideration in this splint output analysis.
Many of Splint's warnings do not represent a real vulnerability, but suggestions for obtaining a higher quality and more reliable code.
Despite this, most of the warnings have been analyzed and solved in order to obtain a more reliable and error-free code.

3 Corrected version of the fragment

This section shows the fragment of code considered "safe", obtained at the end of the analysis phase using tools such as Flawfinder and Splint.

```
1  #include <stdio.h>
2  #include <ctype.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include <bsd/string.h>
7  #include <errno.h>
8
9  ssize_t func1()
10 {
11   char buffer[1024];
12     if(fgets(buffer, 1024, stdin)!=NULL)
13     {
14       if (!isalpha(buffer[0]))
15       {
16         char errormsg[1044];
17         strlcpy(errormsg, buffer,1024);
18         strlcat(errormsg, " is not a valid ID",sizeof(errormsg));
19         fprintf(stderr,"%s\n",errormsg);
20         return -1;
21       }
22     }
23     if(ferror(stdin))
24     {
25       fprintf(stderr,"%s\n",strerror(errno));
26       return -1;
27     }
28
29     return 0;
30 }
31
32 ssize_t func2(int f2d)
33 {
34   char *buf2 = NULL;
35   size_t len = 0;
36   size_t limit = 1024;
37   ssize_t ret = 0;
38   ret = read(f2d, &len, sizeof(len));
39   if(ret < 0)
40   {
41       fprintf(stderr,"%s.\n",strerror(errno));
42       return -1;
43   }
44   if(len > limit)
45   {
46     fprintf(stderr,"too large length\n");
47     return -1;
48   }
49   buf2 = calloc(len+1,sizeof(char));
50
51
52
```

```c
  if(buf2==NULL)
  {

    fprintf(stderr,"%s.\n",strerror(errno));
    return -1;
  }
  ret = read(f2d, buf2,len);
  if(ret < 0)
  {
    fprintf(stderr,"%s.\n",strerror(errno));
    free(buf2);
    return -1;
  }
  buf2[ret] = '\0';
  free(buf2);
  return 0;
}


ssize_t func3(int f3d)
{
  char *buf3 = NULL;
  size_t len = 0;
  ssize_t ret = 0;
  ret = read(f3d, &len, sizeof(len));
  if(ret < 0)
  {
    fprintf(stderr,"%s.\n",strerror(errno));
    return -1;
  }
  if (len > 8000)
  {
    fprintf(stderr,"too large length\n");
    return -1;
  }
  buf3 = calloc(len+1,sizeof(char));
  if(buf3==NULL)
  {
    fprintf(stderr,"%s.\n",strerror(errno));
    return -1;
  }
  ret = read(f3d, buf3,len);
  if(ret < 0)
  {
    fprintf(stderr,"%s.\n",strerror(errno));
    free(buf3);
    return -1;
  }
  buf3[ret] = '\0';
  free(buf3);
  return 0;
}



int  main()
{
  char *foo = "fooooooooooooooooooooooooooooooooooooooooooooooooooo";
  char *buffer = (char *)malloc(10 * sizeof(char));
  ssize_t ret = 0;

  if(buffer==NULL)
```

```
115   {
116     fprintf(stderr,"%s.\n",strerror(errno));
117     return -1;
118   }
119   strlcpy(buffer,foo,sizeof(buffer));
120   ret = func1();
121   if(ret < 0)
122   {
123     free(buffer);
124     fprintf(stderr,"%s.\n",strerror(errno));
125     return -1;
126   }
127   ret = func3((int)sizeof(*foo));
128   if(ret <0)
129   {
130     free(buffer);
131     fprintf(stderr,"%s.\n",strerror(errno));
132     return -1;
133   }
134   free(buffer);
135   return 0;
136 }
```

The above code was obtained by merging the changes made to the fragment using Flawfinder and those made using Splint.
A summary of the main changes made:

- even if in the `func1()` function, the `strcpy` and `strcat` functions were used safely,it was decided to replace them anyway with the *"l-version"*, as it is considered more efficient.
- All the variables have been initialized.
- In the `func3()` function the vulnerability that caused the sign error and the subsequent buffer overflow was solved by checking that the variable `len` did not assume negative values.This could happen because len was int.
  In the end it was decided to make `len` a variable of type `size_t`, just like in in the `func2()` function,so you don't have to worry about it taking negative values.
- All errors are handled and printed on the standard error. All functions return 0 in case of success and -1 in case of failure.
  This increases the reliability of the program.
- All allocated memory blocks are correctly released before each return.
- In the absence of particular performance requirements, the `malloc` function has been replaced by the `calloc` function, which, although considered less efficient, is safer because in addition to allocating the required memory, it also takes care of its initialization.

4 Conclusion

Flawfinder and Splint are powerful and very useful tools to look for vulnerabilities in your code, correct errors and improve the overall reliability of the program. However, they are not infallible.
As we have seen, it can happen that some of the hits are simply false positives. It is therefore unthinkable to rely completely on these tools.
To be able to make the best use of them, a basic knowledge of software security is required to make the right choices and distinguish real vulnerabilities from false positives or find vulnerabilities that these programs can't find.

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 & Software 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] CWE (Common Weakness Enumeration),CWE-476,
https://cwe.mitre.org/data/definitions/476.html

[11] CWE (Common Weakness Enumeration),CWE-252,
https://cwe.mitre.org/data/definitions/252.html

[12] CWE (Common Weakness Enumeration),CWE-772,
https://cwe.mitre.org/data/definitions/772.html

[13] Null Deference,
https://www.owasp.org/index.php/Null_Dereference#Description

[14] Memory Leak,
https://www.owasp.org/index.php/Memory_leak