First Project

Edoardo Di Paolo

November 2020

Contents

1 Tools

1.1 Splint

1.1.1 Strengths & Weakness

Splint provides some information about the detected warnings (some example below). When we use Splint with a default configuration, we have a lot of noise, infact we can have some false positive. However, it is very good for checking types, variables, function assignments, efficiency, unused variables/functions and possible memory leaks. It is a bit obsolete and no longer maintained.

1.1.2 Command execution

First run of "*splint fragment.c*":

```
fragment.c: (in function func1)
fragment.c:10:1: 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)
fragment.c: (in function func2)
fragment.c:24:1: Unrecognized identifier: read
  Identifier used in code has not been declared. (Use -unrecog to inhibit
  warning)
fragment.c:25:1: Unrecognized identifier: buf
fragment.c:25:14: Variable len used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (Use -usedef to inhibit warning)
fragment.c:26:1: Variable buf2 used before definition
fragment.c: (in function func3)
fragment.c:34:5: Variable len used before definition
fragment.c:35:1: Unrecognized identifier: error
fragment.c:39:15: Function malloc expects arg 1 to be size_t gets int: len
  To allow arbitrary integral types to match any integral type, use
  +matchanyintegral.
fragment.c:41:2: Fresh storage buf3 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)
   fragment.c:39:1: Fresh storage buf3 created
fragment.c:32:5: Variable i declared but not used
  A variable is declared but never used. Use /*@unused@*/ in front of
  declaration to suppress message. (Use -varuse to inhibit warning)
fragment.c:44:6: Function main declared to return void, should return int
  The function main does not match the expected type. (Use -maintype to inhibit
  warning)
fragment.c: (in function main)
fragment.c:48:8: Possibly null storage buffer passed as non-null param:
                   strcpy (buffer, ...)
  A possibly null pointer is passed as a parameter corresponding to a formal
  parameter with no /*@null@*/ annotation.  If NULL may be used for this
  parameter, add a /*@null@*/ annotation to the function parameter declaration.
  (Use -nullpass to inhibit warning)
   fragment.c:47:16: Storage buffer may become null
fragment.c:50:7: Unrecognized identifier: len
fragment.c:52:9: Possibly null storage aFile passed as non-null param:
                   fprintf (aFile, ...)
   fragment.c:51:15: Storage aFile may become null
fragment.c:53:7: Parse Error. (For help on parse errors, see splint -help
                   parseerrors.)
```

We can observe that Splint found some vulnerabilities, analyze them.

```
fragment.c:10:1: 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).
```

An attacker could use this vulnerability to fail or maybe to return an unexpected value. This is the code of the vulnerability:

```
1 void func1() {
2     char buffer[1024];
3     printf("Please enter your user id: ");
4     fgets(buffer, 1024, stdin);
5     ...
6 }
```

The line 4 is the error, and we can fix it just adding a condition:

```
1 static void func1() {
2     char buffer[1024];
3     printf("Please enter your user id: ");
4     if(fgets(buffer, 1024, stdin) != NULL) {
5         ...
6     } else {
7         // generate an error
8     }
9 }
```

First vulnerability is gone.
No we have func2() that is never used (so we can just comment it), and its code:

```
1 void func2() {
2     char *buf2;
3     size_t len; // refers to fragment.c:25:14
4     read(f2d, &len, sizeof(len));
5     buf = malloc(len+1); // refers to fragment.c:25:1
6     read(f2d, buf2, len);
7     buf2[len] = '\0'; // refers to fragment.c:26:1
8 }
```

First I included "**unistd.h**" (at start) in the fragment code. Then I edited the func2:

```
1  static void func2() {
2      size_t len = 0; // declare & initialize
3      ssize_t retvar;
4      retvar = read(f2d, &len, sizeof(len)); // store in retvar and
       then check retvar
5      if(retvar < 0) {
6          // handle errors
7      } else {
8          char *buf2 = calloc(len + 1, sizeof(char)); // give memory
       to buf2, with calloc function
9          if(buf2 == NULL) {
10             return; // return to avoid error
11         }
12         retvar = read(f2d, buf2, len);
13
14         if(retvar < 0 || retvar == NULL) {
15             free(buf2); // release buf2 memory
16             return;
17         } else {
18             buf2[retvar] = '\0';
19             free(buf2);
20             return;
21         }
22     }
23 }
```

With this code we avoid memory leaks, and possible buffer overflows. Running again the Splint there are no warnings about func2.

Then we have the func3(). We have several warnings and this was the original function:

```
void func3() {
    char *buf3;
    int i, len; // refers to fragment.c:32:5
    read(f3d, &len, sizeof(len)); // refers to frament.c:34:5
    if (len > 8000) {
        error("too long"); // refers to fragment.c:35:5
        return;
    }

    buf3 = malloc(len); // refers to fragment.c:39:15, fragment.c
    :41:2, fragmment.c:39:1
    read(f3d, buf3, len);
}
```

This is the edited function:

```
static void func3() {
    char *buf3 = NULL; // initialize
    size_t len = 0; // initialize (removed also i)
    ssize_t returnvar = 0; // for the read
    returnvar = read(f3d, &len, sizeof(len));
    if(returnvar < 0 || returnvar == NULL) {
        // show some errror and return
        return;
    } else {
        if len( > 8000) {
            perror("too long"); // use perror, not error
            return;
        }
        buf3 = calloc((size_t)len, sizeof(char)); // give memory to
     buf3

        if(buf3 == NULL) { // check memory for buf3
            return;
        }

        returnvar = read(f3d, buf3, len);
        if(returnvar < 0 || returnvar == NULL) {
            free(buf3); // free the memory to avoid memory leak
            return;
        }
        free(buf3); // avoid memory leak.
    }
    return;
}
```

If we run splint on the fragment we have nothing about func3.

At the end we have the main function, in the original fragment it was:

```
void main() {
    char *boo = "booooooooooooooooooooooooooooooooooooooooooooooooo";
    char *buffer = (char*) malloc(10 * siizeof(char));
    strcpy(buffer, boo); // refers to fragment.c:48:8, fragment.c
    :47:16
```

```
    func1();
    func3(len(*boo)); // refers to fragment.c:50:7
    FILE *aFile = fopen("/tmp/tmpfile", "w"); // refers to fragment
    .c:52:9
    fprintf(aFile, "%s", "hello world") // refers to fragment.c
    :53:7
    fclose(aFile);
}
```

About the null pointer, we can say that it occurs when the application dereferences a pointer that it expects to be valid. This can cause an application crash, but maybe an attacker could use the resulting exception to bypass the software logic or he could be able to reveal some debugging information. We have also a memory leak aboutt the buffer that is allocated but then there is no free. I fixed the vulnerabilities changing the main function into:

```
int main() {
    char *boo = "boooooooooooooooooooooooooooooooooooooooooooooooooo";
    char *buffer = (char*) malloc(10 * siizeof(char));

    if(buffer == NULL) {
        // buffer is not allocated, so error
        return -1;
    }

    strcpy(buffer, boo);
    func1();

    FILE *aFile = fopen("/tmp/tmpfile", "w");

    if(aFile == NULL) {
        free(buffer); // avoid memory leak
        return -1;
    }

    int retfprint = 0;
    retfprint = fprintf(aFile, "%s", "hello world")

    if(retfprint < 0) {
        free(buffer);
        return -1;
    }

    (void) fclose(aFile);

    free(buffer); // release memory
    return 0;
}
```

Executing with this code the splint command I've not any vulnerability.

1.2 Flawfinder

1.2.1 Strengths & Weakness

Flawfinder is a tool to find possible security weakness. It categorizes the risks by a level that goes from 0 (very low) to 5 (very high). Flawfinder, not as Splint, has not access to the data flow and neither to the program control flow therefore we can have many false positives.

1.2.2 Command execution

Command "*flawfinder fragment.c*":
Final results

```
fragment.c:38:  [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:4:  [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:8:  [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:10:  [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:41:  [2] (misc) fopen:
  Check when opening files - can an attacker redirect it (via symlinks),
  force the opening of special file type (e.g., device files), move things
  around to create a race condition, control its ancestors, or change its
  contents? (CWE-362).
fragment.c:9:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
fragment.c:18:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:20:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:26:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
fragment.c:32:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
```

reference	code
fragment.c:48	strcpy(buffer, boo)
fragment.c:8	char buffer[1024]
fragment.c:13	char errormsg[1044]
fragment.c:15	strcat(errormsg, " is not a valid ID ")
fragment.c:51	FILE *aFile = fopen("tmp/tmpfile", "w")
fragment.c:14	strncpy(errormsg, buffer, 1024)
fragment.c:24	read(f2d, &len, sizeof(len))
fragment.c:26	read(f2d, buf2, len)
fragment.c:33	read(f3d, &len, sizeof(len))
fragment.c:40	read(f3d, buf3, len)

```
ANALYSIS SUMMARY:

Hits = 10
Lines analyzed = 43 in approximately 0.03 seconds (1506 lines/second)
Physical Source Lines of Code (SLOC) = 42
Hits@level = [0]   2 [1]   5 [2]   4 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]  12 [1+]  10 [2+]   5 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 285.714 [1+] 238.095 [2+] 119.048 [3+] 23.8095 [4+] 23.8095 [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'
```

As we can see, in this first run flawfinder catches 10 possible vulnerabilities. The possible vulnerabilities are sorted by their risk level, so we have one vulnerability of level 4, then 4 of level 2 and then 5 of level 1.

This is the table with the possible vulnerabilities in the fragment.c code. If we run the flawfinder with -F option (so we hide the falsepositive vulnerabilities) we can remove from this table the **second row** and the **third row**.
The two false positives are the declarations of the two buffers. We have to check correctly the data that we are putting in the buffers, otherwise we can have a buffer overflow vulnerability.

About the first warning, flawfinder tells us to use "strlcpy" (not only) function. So I fixed editing the main function in this way:

```
1  void main() {
2      char *boo = "boooooooooooooooooooooooooooooooooooooooooooooooooo";
3      char *buffer = (char*) malloc(10 * siizeof(char));
4      strlcpy(buffer, boo, sizeof(buffer));
5      func1();
6      func3(len(*boo));
7      FILE *aFile = fopen("/tmp/tmpfile", "w");
8      fprintf(aFile, "%s", "hello world");
9      fclose(aFile);
10 }
```

Why is strcpy vulnerable? Because it doesn't specify the size of the destination array, so we can fall in a buffer overflow vulnerability if the buffer is not big enough. A few words about other suggested functions:

1. *strncpy*: is not too much performing and it is less secure than other proposed functions because we can get a string that is null-terminated (so we

can have a *segmentation fault*);

2. *snprintf*: this function always adds \0 at the end of a string, but we can have some buffer overflow.

Then we have warnings about some statistically-sized arrays. This is a **false positive** warning: we have only a *fgets* function that is actually using *buffer* and it reads at most 1024 chars from the stdin, so we can't have any overflow.

```
void func1() {
    char buffer[1024];
    printf("Please enter your user id: ");
    fgets(buffer, 1024, stdin);
    if(!isalpha(buffer[0])) {
        char errormsg[1044];
        strncpy(errormsg, buffer, 1024);
        strcat(errormsg, " is not a valid ID");
    }
}
```

About the *fragment.c:13, fragment.c:15, fragment.c:9*, they are also a **false positive** warning: the *errormsg* contains enough bytes to manage the contents of the buffer plus the "is not a valid ID". We are also safe with the *strncpy* because we have always the null-terminator character due to fgets function. However we can use strlcpy and strlcat functions.

Then we have the func2:

```
void func2() {
    char *buf2;
    size_t len;
    read(f2d, &len, sizeof(len));
    buf2 = malloc(len + 1);
    read(f2d, buf2, len);
    buf2[len] = '\0';
}
```

We have two warnings about read function. The first read is a **false positive**. There is no input validation: an attacker can do an integer overflow in the malloc call, with the len value, and this integer overflow will cause a buffer overflow in the next line. The len variable is a *size_t type* this means that it is an unsinged integer. An attacker could insert the max value for the len and then add 1, and so we have the integer overflow.
To fix this we can limit the len in this way:

```
void func2() {
    char *buf2;
    size_t len;
    size_t limit = 1024;
    read(f2d, &len, sizeof(len));
    if (len > limit) {
        // handle some error;
        return;;
    }
```

```
    buf2 = malloc(len + 1);
    read(f2d, buf2, len);
    buf2[len] = '\0';
}
```

Then we have func3:

```
void func2() {
    char *buf3;
    int i, len;
    read(f3d, &len, sizeof(len));
    if (len > 8000) {
        error("too long");
        return;
    }

    buf3 = malloc(len);
    read(f3d, buf3, len);
}
```

The read warning is also another **false positive**, but the main problem is alwasy the input validation. See that the len variable is a *integer*, so it can be also a negative number and then have a buffer overflow with the last read. We need also to add 1 to the len in the malloc call.

```
void func2() {
    char *buf3;
    int i, len;
    read(f3d, &len, sizeof(len));
    if (len < 0 || len > 8000) {
        error("too long");
        return;
    }

    buf3 = malloc(len + 1);
    read(f3d, buf3, len);
    buf3[len] = '\0'
}
```

To fix the vulnerability about the fopen function, I create a new tmpfile with the *mkostemp* function, passing the flag O_EXCL. Then I use the fdopen function to open this new file and then I write into the file.

2 Fixed fragment

This is the fixed fragment without warnings in Splint and in Flawfinder.

```
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <bsd/string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
10
11 static ssize_t func1() {
12     char buffer[1024];
13     if(fgets(buffer, 1024, stdin) != NULL) {
14         if(!isalpha(buffer[0])) {
15             char errormsg[1044];
16             strlcpy(errormsg, buffer, 1024);
17             strlcat(errormsg, " is not a valid ID", sizeof(errormsg
    ));
18             return -1;
19         }
20     }
21
22     return 0;
23 }
24
25 /*@unused@*/
26 static ssize_t func2(int f2d) {
27     char *buf2 = NULL;
28     size_t len = 0;
29     size_t limit = 1024;
30     ssize_t returnvar = 0;
31
32     returnvar = read(f2d, &len, sizeof(len));
33
34     if(returnvar < 0)
35         return -1;
36
37     if(len > limit)
38         return -1;
39
40     buf2 = calloc(len + 1, sizeof(char));
41
42     if(buf2 == NULL)
43         return -1;
44
45     returnvar = read(f2d, buf2, len);
46     if(returnvar < 0) {
47         free(buf2);
48         return -1;
49     }
50
51     buf2[returnvar] = '\0';
52     free(buf2);
53     return 0;
54 }
55
56 static ssize_t func3(int f3d) {
57     char *buf3 = NULL;
58     size_t len = 0;
59     ssize_t returnvar = 0;
60     returnvar = read(f3d, &len, sizeof(len));
61
62     if(returnvar < 0)
63         return -1;
64
65     if(len > 8000)
```

```
66          return -1;
67
68      buf3 = calloc(len + 1, sizeof(char));
69
70      if(buf3 == NULL)
71          return -1;
72
73      returnvar = read(f3d, buf3, len);
74
75      if(returnvar < 0) {
76          free(buf3);
77          return -1;
78      }
79
80      buf3[returnvar] = '\0';
81
82      free(buf3);
83      return 0;
84  }
85
86  int main() {
87      char *boo = "booooooooooooooooooooooooooooooooooooooooooooooooooo";
88      char *buffer = (char *)malloc(10 * sizeof(char));
89      ssize_t returnvar = 0;
90
91      if(buffer == NULL) {
92          // buffer is not allocated, so error
93          return -1;
94      }
95
96      strlcpy(buffer, boo, sizeof(buffer));
97      returnvar = func1();
98
99      if(returnvar < 0) {
100         free(buffer);
101         return -1;
102     }
103
104     if(returnvar < 0) {
105         free(buffer);
106         return -1;
107     }
108
109     char safeFile[20]; // false positive
110     int fd = -1;
111
112     strlcpy(safeFile, "/tmp/tmpfile", sizeof(safeFile));
113
114     fd = mkostemp(safeFile, O_EXCL);
115
116     if(fd < 0) {
117         free(buffer);
118         free(safeFile);
119         return -1;
120     }
121
122     aFile = fdopen(fd, "w");
```

```
123
124     if(aFile == NULL) {
125         // error
126         free(buffer); // avoid memory leaks
127         free(safeFile);
128         return -1;
129     }
130     int retfprint = 0;
131     retfprint = fprintf(aFile, "%s", "hello world");
132     if(retfprint < 0) {
133         free(buffer);
134         free(fileName);
135         return -1;
136     }
137     (void) fclose(aFile);
138     free(fileName);
139     free(buffer); // release memory
140     return 0;
141 }
```

All the warnings by Splint or Flawfinder are false positive.

3 Conclusion

Flawfinder and Splint are good tools to check our code, but they can fail in many false positive warnings. We need some knowledge about security software to find out which are false positive and which are really vulnerabilities. We have also to imagine other vulnerabilities that can occur starting from another one (like the read function).