Security in Software Applications
Project 1, 2020-21

Davide Quaranta
1715742

Contents

Original fragment

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

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");
     }
}


/* f2d and f3d are file descriptors obtained after opening files*/
void func2(int f2d) {
char *buf2;
size_t len;
read(f2d, &len, sizeof(len));
buf = malloc(len+1);
read(f2d, buf2, len);
buf2[len] = '\0';
}

void func3(int f3d){
char *buf3;
int i, len;
read(f3d, &len, sizeof(len));
if (len > 8000) {
error("too long");
return;
}

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


void main()
{
char *boo = "booooooooooooooooooooooooooooooooooooooooooooooooooo";
char *buffer = (char *)malloc(10 * sizeof(char));
strcpy(buffer, boo);
func1();
FILE *aFile = fopen("/tmp/tmpfile", "w");
fprintf(aFile, "%s", "hello world")
fclose(aFile);
}
```

1. Splint

Splint is a linter that gives very **precise** and **contextualized** indications aimed at improving **code quality** and eliminating vulnerabilities. The main downside is the **steep learning curve** and the fact that it needs to be configured in a non intuitive way, for example by building custom rulesets even for very common header includes (see Appendix). Furthermore some description messages are **unclear**.

1.1 Output

```
Splint 3.1.2 --- 20 Feb 2018
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:51:9: Possibly null storage aFile passed as non-null param:
                   fprintf (aFile, ...)
   fragment.c:50:15: Storage aFile may become null
fragment.c:52:7: Parse Error. (For help on parse errors, see splint -help
                   parseerrors.)
*** Cannot continue.
```

1.2 Comments

Splint gave various suggestions, mainly of this type:

- Unrecognized identifiers, caused by wrong function names or missing imports.
- Variables used before being defined.
- Unused variables.
- Ignored return values.
- Improper use of functions (ex. wrong argument types).
- Memory leaks, caused by not freeing allocated storage.
- Absence of NULL-checking.

Further in the analysis (and fixing) process:

- Wrong `main()` return type.
- Absence of the `static` qualifier for functions that are not used outside the fragment.

2. Flawfinder

Flawfinder only uses a **set of rules** to scan the code and detect possible known flaws related to the use of **dangerous/risky functions**, and that's the reason why it is advertised as capable of working with un-buildable code. It is very **easy** to use, the messages are easy to understand and are directly linked to the corresponding CWEs; it also supports HTML output for a better reading. The downside is that it gives a lot of **false positives**.

2.1 Output

```
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:48:  [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).
strcpy(buffer, boo);
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.
char buffer[1024];
fragment.c:13:  [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.
char errormsg[1044];
fragment.c:15:  [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.
strcat(errormsg, " is not  a valid ID");
fragment.c:50:  [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).
FILE *aFile = fopen("/tmp/tmpfile", "w");
fragment.c:14:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
strncpy(errormsg, buffer,1024);
fragment.c:24:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
read(f2d, &len, sizeof(len));
fragment.c:26:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
```

```
  (CWE-120, CWE-20).
read(f2d, buf2, len);
fragment.c:33:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
read(f3d, &len, sizeof(len));
fragment.c:40:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
read(f3d, buf3,len);

ANALYSIS SUMMARY:

Hits = 10
Lines analyzed = 53 in approximately 0.00 seconds (11756 lines/second)
Physical Source Lines of Code (SLOC) = 45
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+] 266.667 [1+] 222.222 [2+] 111.111 [3+] 22.2222 [4+] 22.2222 [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'
(https://dwheeler.com/secure-programs) for more information.
```

2.2 Comments

The hits by Flawfinder are about:

- Possible buffer overflows due to dangerous functions (ex. `strcpy`).
- Possible buffer overflows due to absence of bounds checking.
- Possible buffer overflows due to absence of NULL-termination.
- Possible race conditions or malicious file redirections.

3. Correction

To carry out the assignment, this strategy was followed:

1. Adjust indentation;
2. Fix syntax errors;
3. Fix wrong function names and add missing imports;
4. Run the tools on the fragment and fix the flaws.

3.1 Corrected fragment

```c
#define _GNU_SOURCE

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

static void func1() {
     char *buffer = calloc(1024, sizeof *buffer);
     if (buffer == NULL) {
          return;
     }
     printf("Please enter your user id :");
     if (fgets(buffer, 1024, stdin) == NULL) {
          exit(EXIT_FAILURE);
     }
     if (!isalpha(buffer[0])) {
          char *errormsg = calloc(1044, sizeof *errormsg);
          if (errormsg != NULL) {
               (void)strlcpy(errormsg, buffer, 1024);
               (void)strlcat(errormsg, " is not  a valid ID", 19);
               free(errormsg);
          }
     }
     free(buffer);
}

/* f2d and f3d are file descriptors obtained after opening files*/
void func2(int f2d) {
     char *buf2;
     size_t len;
     (void)read(f2d, &len, sizeof(len));
     buf2 = malloc(len+1);
     if (buf2 != NULL) {
          (void)read(f2d, buf2, len);
          buf2[len] = '\0';
          free(buf2);
     }
}

```

```c
void func3(int f3d) {
     size_t len;
     (void)read(f3d, &len, sizeof(len));
     if (len > 8000) {
          fprintf(stderr, "too long");
          return;
     }
     char *buf3 = malloc(len);
     if (buf3 == NULL) {
          return;
     }
     (void)read(f3d, buf3, len);
     free(buf3);
}

int main() {
     char *boo = "booooooooooooooooooooooooooooooooooooooooooooooooooo";
     char *buffer = calloc(10, sizeof *buffer);
     if (buffer != NULL) {
          (void)strlcpy(buffer, boo, sizeof(buffer));
          free(buffer);
     }
     func1();

     int fd = mkostemp("/tmp/XXXXXX", O_WRONLY);
     if (fd == -1) {
          exit(EXIT_FAILURE);
     }

     FILE *aFile = fdopen(fd, "w");
     if (aFile == NULL) {
          exit(EXIT_FAILURE);
     }

     fprintf(aFile, "%s", "hello world");
     (void)fclose(aFile);
     return 0;
}
```

About the insecure temporary file (CWE-377)

In function `main()` there was a possible race condition on `/tmp/tmpfile`; moreover, since `/tmp` is world-writable, an attacker could compromise an arbitrary file by creating a symlink called `/tmp/tmpfile`. This flaw can be addressed in various ways, for example by using functions like `tmpnam()`, `tempnam()`, `mktemp()`, `tmpfile()` to generate a random temporary file, but said functions lack sufficient randomization[1]. The function `mkstemp()` can be used to successfully fix the issue, but to protect older systems it is needed to explicitly restrict the permissions with `umask`, and use the `O_EXCL` flag when opening the file. Finally, a simpler and effective way consists of:

- Using `mkostemp()` to create a random temporary file, specifying a template.
- Using `fdopen()` on the file descriptor returned by `mkostemp()` in write mode.

It is worth to note that `mkostemp()` implicitly creates the file with:

- `0600` permissions: read and write for the owner only.
- the flag `O_EXCL`: it fails if the file already exists.
- the flag `O_CREAT`: it creates the regular file if doesn't exist.

[1] https://cwe.mitre.org/data/definitions/377.html

3.2 Splint output

Executed with `splint -load lib fragment.c`:

```
Splint 3.1.2 --- 20 Feb 2018

Finished checking --- no warnings
```

`-load` loaded the ruleset `lib.lcd` (see Appendix).

3.3 Flawfinder output

Executed with `flawfinder --context fragment.c`:

```
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:36:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
     (void)read(f2d, &len, sizeof(len));
fragment.c:39:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
          (void)read(f2d, buf2, len);
fragment.c:47:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
     (void)read(f3d, &len, sizeof(len));
fragment.c:56:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
     (void)read(f3d, buf3, len);

ANALYSIS SUMMARY:

Hits = 4
Lines analyzed = 81 in approximately 0.00 seconds (16457 lines/second)
Physical Source Lines of Code (SLOC) = 73
Hits@level = [0]   3 [1]   4 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   7 [1+]   4 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 95.8904 [1+] 54.7945 [2+]   0 [3+]   0 [4+]   0 [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'
(https://dwheeler.com/secure-programs) for more information.
```

This output, howerver, is only composed by **false positives**, since `read` is not used in a loop or recursive function. So it was possible to instruct Flawfinder to ignore them by increasing the minimum risk level with `flawfinder --context -m 3 fragment.c` and obtain a **clean output**:

```
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:


ANALYSIS SUMMARY:

No hits found.
Lines analyzed = 78 in approximately 0.01 seconds (15188 lines/second)
Physical Source Lines of Code (SLOC) = 71
Hits@level = [0]   3 [1]   5 [2]   1 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   9 [1+]   6 [2+]   1 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 126.761 [1+] 84.507 [2+] 14.0845 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 3
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
(https://dwheeler.com/secure-programs) for more information.
```

3.4 Final screenshot

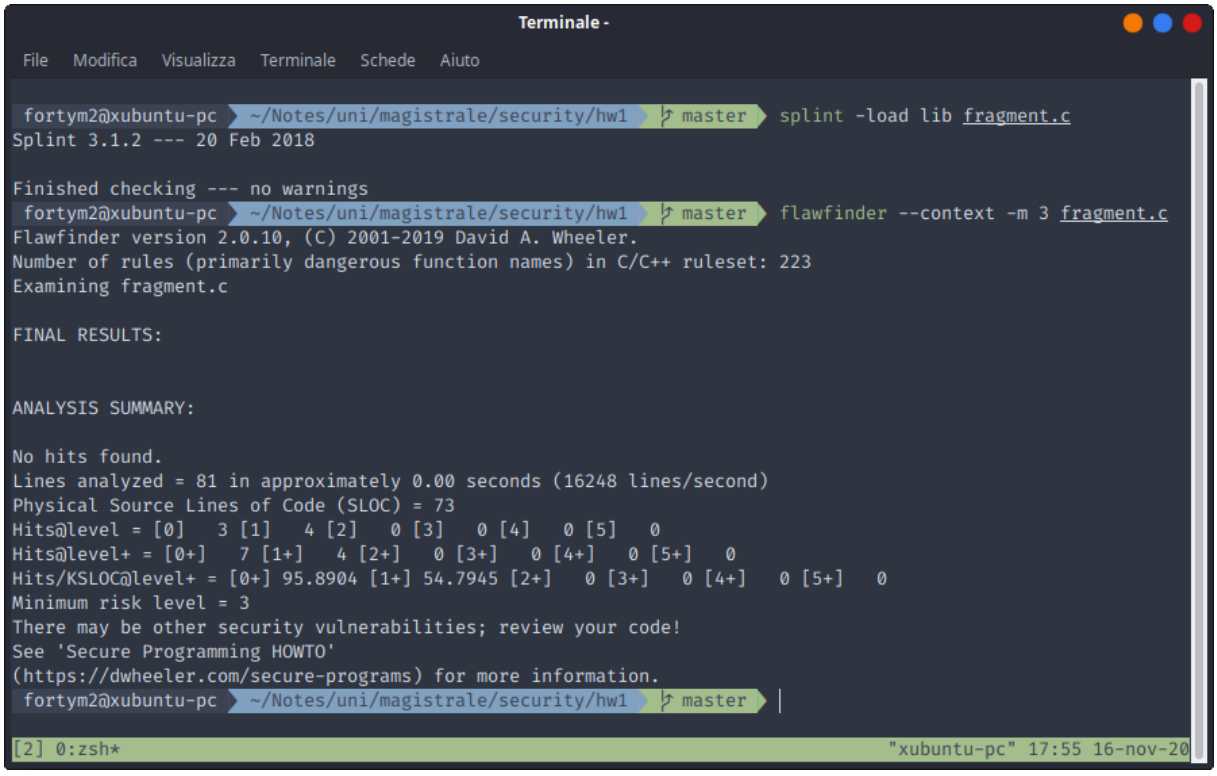

Figure 3.1: Final execution of Splint and Flawfinder

4. Appendix

The analysis was made on a `Linux 5.4.0-52-generic x86\_64 GNU/Linux` machine with the Xubuntu distribution.

To use safe libraries like `strlcat` it was necessary to install the package `libbsd-dev` ("utility functions from BSD systems") and instruct Splint to load the rules for `bsd/string.h`; it was also necessary to run Splint with the `+unixlib` flag, to make it recognize `unistd.h`.

Since the internal `stdlib.h` used by Splint (in `/usr/share/splint/lib`) doesn't have `mkostemp`, it was necessary to tell Splint to specifically use the system one.

To address said issues it was needed to create a custom **specification file** for Splint with:

```
splint /usr/include/bsd/string.h /usr/include/stdio.h /usr/include/stdlib.h +unixlib
-D__gnuc_va_list=va_list -D_GNU_SOURCE=1 either -dump lib
```

That generated a `lib.lcd` file in the current directory. The directive `__gnuc_va_list=va_list` resolved a parse error in `stdio.h` and `_GNU_SOURCE=1` is a required **feature test macro** that allowed to select `mkostemp`.

Then Splint was executed with `splint -load lib fragment.c`.