Security in Software Applications: Homework 1

Professor: Francesco Parisi-Presicce

Student: Riccardo Chiaretti

Contents

1 Splint

Splint, which stands for **Secure Programming Lint**, is a tool used for statically analyzing the source code of C programs searching for security vulnerabilities and programming mistakes. It should be used by programmers during the development process encouraging security-oriented development.
Splint is a sophisticated tool (at least with respect to the tool analyzed in the following chapter namely, Flawfinder) which is able to do many of the traditional checks like type inconsistency, unused declarations, variables used before definition, ignored return values, null dereferences: for example, given the output of some function which may return `NULL`, Splint warns the programmer to run into problems if the output is later used in the program and not checked to be `NULL`.
Further, if the programmer adds *annotations*, Splint makes possible to perform a deeper analysis by doing more powerful checks. In detail, the annotations are particular comments (represented as `/*@@*/`) that the programmer uses in the code to express his intent. At this point, Splint detects if there are inconsistencies between annotations and code: in this case, either the programmer was wrong in writing the annotations or the code he wrote does not really represent his intentions (which may bring to possible bugs).
The main disadvantage of this tool is that it is both *unsound* and *incomplete*: this means that while analyzing some C program with Splint, either it does not warn about something which is actually a possible vulnerability (unsound) or it warns the programmer where there is no actual problem (incomplete).

Below, we discuss which is the output of the analysis performed on the source files `A.c, B.c` and `C.c` by using Splint and Flawfinder. Then, we provide a new version of these programs which fixes all the potential vulnerabilities found by the tool.
We start from analyzing them with Splint.

1.1 Static Analysis of `A.c` with Splint

At first impact, by launching Splint on this file, it warns us about 9 potential vulnerabilities. The first of these warnings is about the statement in the `while` loop in the `stringcopy()` function:

```
A.c:4:8: Test expression for while not boolean, type char: *str2
  Test expression type is not boolean. (Use -predboolothers to inhibit warning)
```

This is not actually a real problem because the expression in the loop is, character by character, the string passed as first argument by the user. Indeed, since that the string `argv[1]` terminates with the `\0` character, when we reach the end of the string, the `while` condition will be false, exiting from the loop without incurring in any infinite loop. The warning can be avoided by put an explicit boolean condition like `*str2 != '\0'`.
The second warning is saying that we are trying to return a temporary storage associated to an argument of the function:

```
A.c:7:8: Implicitly temp storage str2 returned as implicitly only: str2
  Temp storage (associated with a formal parameter) is transferred to a
  non-temporary reference. The storage may be released or new aliases created.
  (Use -temptrans to inhibit warning)
```

By using the `returned` qualifier in `stringcopy(..., /*@returned@*/ *str2)`, Splint understands that, what the function is returning, is the same storage passed as parameter without incurring in the risk of new aliases to be created or storages to be released. Since the return value of the `stringcopy()` function is ignored by the `main()`, apart from the possibility of using the `returned` qualifier, it is possible to cahnge the function to a `void` one (instead of `char *`) so, deleting the `return` instruction.
The third warning is quite important:

```
A.c:16:12: Possibly null storage buffer passed as non-null param:
              stringcopy (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)
   A.c:14:16: Storage buffer may become null
```

This is caused by the `malloc()` function which, in case of failure, returns `NULL`. Since we are going to dereference the output buffer of `malloc()` in `stringcopy()`, if it is `NULL` than our process will be killed with a segmentation fault. What Splint suggests is not a real solution but, just in the case the developer **is aware** that `buffer` may be `NULL`, it is suggesting to put the annotation `/*@null@*/` in the definition of the function which uses `buffer`. At least for this program, it is not a solution; indeed, the right way of handling this scenario, is to safely exiting when `buffer == NULL`.
The fourth one is related to the fact that we are passing a possibly undefined storage to the `stringcopy()` function.

```
A.c:16:12: Passed storage buffer not completely defined (*buffer is undefined):
              stringcopy (buffer, ...)
  Storage derivable from a parameter, return value or global is not defined.
  Use /*@out@*/ to denote passed or returned storage which need not be defined.
  (Use -compdef to inhibit warning)
   A.c:14:51: Storage *buffer allocated
```

Since **we know** that `buffer` is effectively populated in `stringcopy()`, we can follow the suggestion given by Splint and use the `/*@out@*/` annotation in the definition of `stringcopy(/*@out@*/ char *str1, ...)`.
The fifth one is pointing out to the programmer that the return value of the `stringcopy()` function is ignored:

```
A.c:16:1: Return value (type char *) ignored: stringcopy(buffe...
  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)
```

Therefore, in the case in which this value is meaningless for later execution, Splint suggests to cast it to `void` (like `(void) stringcopy(...)`). Since that for our purposes it is useless, we decided just to change the return value of the function to `void` and remove the `return` statement.
The sixth warning highlights the ability of Splint to do the analysis of the *control flow* of a program:

```
A.c:19:2: Path with no return in function declared to return int
  There is a path through a function declared to return a value on which there
  is no return statement. This means the execution may fall through without
  returning a meaningful result to the caller. (Use -noret to inhibit warning)
```

Indeed, the warning is about the return value of `main()`. In C language, if no return type is specified in front of a function, this is implicitly `int`. For this reason, Splint rightly points out that there is no integer returned by `main()`. To avoid this minor problem, in our solution, we put a `return 1` statement.
The seventh one derives from the ability of splint in detecting possible memory leaks:

```
A.c:19:2: Fresh storage buffer 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)
   A.c:14:51: Fresh storage buffer created
```

The locally allocated `buffer` is not freed before the end of the function, when we loose any reference to it. For this reason, before the end of the `main()`, it is necessary to put the instruction `free(buffer)` to release the previously allocated storage.
The second to last warning is not a real vulnerability but it should be taken into consideration when writing a elegant software:

```
A.c:10:10: Parameter argc not used
  A function parameter is not used in the body of the function. If the argument
  is needed for type compatibility or future plans, use /*@unused@*/ in the
  argument declaration. (Use -paramuse to inhibit warning)
```

Since one of the features of Splint is to spot *unused variables*, it indicates that `argc` is not used, suggesting to put the `/*@unused@*/` annotation in front of the variable in the definition of the function (`main(/*@unused@*/int argc, ...)`). On the other hand, regarding the proposed solution, the usage of `argc` is fundamental to avoid a possible segmentation fault. Indeed, if the user does not input any parameter to the program, trying to dereference `argv[1]` the program will be killed because accessing to an invalid memory location.
At the end, the last warning is enforcing the concept of *information hiding*:

```
A.c:2:7: Function exported but not used outside A: stringcopy
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
   A.c:8:1: Definition of stringcopy
```

Indeed, since that no other module is using the `stringcopy()` function but it is still exported, Splint suggests to make it invisible outside the current module by using the `static` qualifier.

Another interesting aspect of Splint is that, if we launch it on a file using the `+bounds` flag, it begins to notify the fact that there may be possible violations of the limits of the buffers used in the program. In order to "solve" this type of warnings, we need to use annotations: as said above, we specify which are the assumptions on buffer sizes and Splint checks whether the annotations we gave are consistent or not. In particular, it gives us with 3 warnings which we will discuss below.
The first to be mentioned should be the following:

```
A.c:16:20: Possible out-of-bounds read: argv[1]
    Unable to resolve constraint:
    requires maxRead(argv @ A.c:16:20) >= 1
     needed to satisfy precondition:
    requires maxRead(argv @ A.c:16:20) >= 1
```

Indeed, since that in `main()` we are calling `stringcopy()` passing `argv[1]` as second parameter, Splint warns that, if `argv` does not have at least two elements, we are reading from a not-allocated storage. Assuming **to be aware** of the possibility that the user does not give any input to the program (indeed in the solution we safely exit in case no input is given), by using the annotation `/*@requires maxRead(argv) == 1 @*/`[1], we are telling to Splint that the maximum index of `argv` which can be safely read is 1 so, `argv[1]` can be legally read.
The second warning is about the buffer `str2` which is used inside the `while` statement, in `stringcopy()`:

```
A.c:4:8: Possible out-of-bounds read: *str2
    Unable to resolve constraint:
    requires maxRead(str2 @ A.c:4:9) >= 0
     needed to satisfy precondition:
    requires maxRead(str2 @ A.c:4:9) >= 0
  A memory read references memory beyond the allocated storage. (Use
  -boundsread to inhibit warning)
```

In particular, since we are de-referencing it, it is possible that we are reading memory beyond the maximum allocated storage. To make Splint understand that `str2` can be safely de-referenced because it contains at least one character, we need to put the following annotation `/*@requires maxRead(str2) >= 0 @*/`. What it is saying, is that there is at least one element in the `str2` buffer which can be read.
However, Splint is not able to resolve the precondition on `str2` in `stringcopy()`, because we did not give any constraint on `argv[1]` in `main()`:

```
A.c:22:1: Possible out-of-bounds read: stringcopy(buffer, argv[1])
    Unable to resolve constraint:
    requires maxRead(argv[1] @ A.c:22:20) >= 0
     needed to satisfy precondition:
    requires maxRead(argv[1] @ A.c:22:20) >= 0
     derived from stringcopy precondition: requires maxRead(<parameter 2>) >= 0
  A memory read references memory beyond the allocated storage. (Use
  -boundsread to inhibit warning)
```

Therefore, we need to write the same annotation as before but referred to the buffer `argv[1]`: `/*@requires maxRead(argv[1]) >= 0 @*/`.
At this point, the last warning is about the buffer `str1`:

```
A.c:5:1: Possible out-of-bounds store: *str1++
    Unable to resolve constraint:
    requires maxSet(str1 @ A.c:5:2) >= 0
     needed to satisfy precondition:
    requires maxSet(str1 @ A.c:5:2) >= 0
  A memory write may write to an address beyond the allocated buffer. (Use
  -boundswrite to inhibit warning)
```

[1]The `requires` clause specifies a predicate which must be true when the function (where the `requires` clause is used) is called. Therefore, Splint will assume that all constraints in the `requires` clause are true.

Indeed, since in `stringcopy()` we are writing in that buffer, Splint warns that if `str1` is not correctly allocated, we may write on memory beyond the allocated one. Basically, this is a possible **buffer overflow** with which an attacker may get to overwrite sensitive data on the heap (in fact, `str1` is allocated on the heap by `main()` so, it is more correct to talk about **heap overflow**). The original program is vulnerable to the heap overflow attack because it does not do any check about the size of the user input to be at most the size of `buf`. For this reason, in the solution below, we did this check in order to be sure to write at most the size of the buffer allocated in `main()` without incurring in the risk of overflowing it.

1.2 Static Analysis of `B.c` with Splint

By launching Splint on the file `B.c` without any flag, we get 4 warnings. The first of these, is the following:

```
B.c:7:1: Unrecognized identifier: read
  Identifier used in code has not been declared. (Use -unrecog to inhibit
  warning)
```

Basically, since to use the `read()` function we need the POSIX library, Splint must be launched with the flag `+posixlib`, selecting the POSIX library. In this way, the tool understands that `read()` is being used, solving also this other warning:

```
B.c:9:5: 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)
```

Indeed, before specifying the use of the library above, Splint was warning about the fact that the `len` variable was used without being defined first. Now, it understands that it will be initialized by the `read()` function.
Another warning is related to the issue shown in the figure below:

```
B.c:7:1: Return value (type ssize_t) ignored: read(fd, &len, s...
  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)
```

Basically, both times the `read()` function is used, the return value is ignored. `read()` returns the number of bytes read (0 if the end of file is reached) but, more important, it returns -1 in case of failure. We should focus on this scenario because, in case of error, if we do not handle it correctly, both `len` and `buffer` may be in an "unknown" state (together with the file associated to the descriptor `fd`) which may bring to possible crash of the program. Indeed, since `len` is later used as size in `malloc()`, what can happen is:

1. `read()` returns an error and `len` is not correctly set;
2. USUALLY, declared but not defined integer variables are set to 0 by default;
3. `len` is used as size of the `buffer` storage to be be allocated by `malloc()`;
4. if the size specified in `malloc()` is 0 two things can happen:
   - either it returns `NULL`;
   - or it returns a pointer to a buffer which theoretically has size 0, causing a buffer overflow when writing on it.

Therefore, not handling in a safe way the return value of the first call to `read()`, may cause either a crash of the program for segmentation fault or a buffer overflow.
The following warning:

```
B.c:13:14: Possibly null storage buf passed as non-null param:
                read (..., buf, ...)
  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)
   B.c:12:7: Storage buf may become null
```

Is related to possible NULL dereferences of the buffer storage in the program. Indeed, since the malloc() function is known to return NULL in case of error, but not only (like in the possible execution path above), buffer may be dereferenced while it should not. As said also in the previous analysis, Splint suggests to use the /*@null@*/ annotation in the case **we are aware** that NULL may be used as parameter in read(). This is NOT our case otherwise the program will terminate with a segmentation fault: we must avoid this scenario by safely exiting when malloc() returns NULL.
The last warning is always referred to the buffer storage:

```
B.c:15:2: Fresh storage buf 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)
   B.c:12:1: Fresh storage buf created
```

It concerns the fact that the locally created storage, is not freed before the end of the function. Since that buffer does not appear in the formal parameters of func() or in the return value, (given that func() is void), we may think that buffer is a temporary storage and the programmer forgot to release it. We are facing a memory leak which is avoided by using free() on the buffer before the end of the function.

Regarding the warnings about the buffers boundaries used within the program, in the solution we did some modification. In particular, since that annotations should be put on formal parameters of a function and in this case neither buffer nor len are arguments of func, there was the "necessity" to declare another function (my_read()) in which these two parameters appear as arguments.
At this point, the warning triggered is the following:

```
Bcorr.c: (in function my_read)
Bcorr.c:11:21: Possible out-of-bounds store:
    read(fd, buffer, size - bytes_read)
    Unable to resolve constraint:
    requires maxSet(buffer @ Bcorr.c:11:30) >= size @ Bcorr.c:11:38 + -1
     needed to satisfy precondition:
    requires maxSet(buffer @ Bcorr.c:11:30) >= size @ Bcorr.c:11:38 -
    bytes_read @ Bcorr.c:11:45 + -1
     derived from read precondition: requires maxSet(<parameter 2>) >=
    <parameter 3> + -1
  A memory write may write to an address beyond the allocated buffer. (Use
  -boundswrite to inhibit warning)
```

It is correctly saying that there is a possible out-of-bounds storage regarding the buffer variable passed as second parameter of my_read(); this is caused by the precondition requires maxSet(<parameter2>) >= <parameter 3> - 1 of read(). Therefore, we need to state that when my_read() is called, at least size bytes can be safely written in buffer, which, translated for being understood by Splint becomes that the maximum index of buffer which can be safely written is size - 1: /*@requires maxSet(buffer) >= (size - 1)@*/.
At this point, another warning is triggered by Splint:

```
Bcorr.c: (in function func)
Bcorr.c:33:17: Possible out-of-bounds store:
    my_read(fd, (size_t *)&len, sizeof((len)))
    Unable to resolve constraint:
    requires maxSet(&len @ Bcorr.c:33:40) >= sizeof((len)) @ Bcorr.c:33:52 + -1
     needed to satisfy precondition:
    requires maxSet(&len @ Bcorr.c:33:40) >= sizeof((len)) @ Bcorr.c:33:52 + -1
     derived from my_read precondition: requires maxSet(<parameter 2>) >=
    <parameter 3> - 1
  A memory write may write to an address beyond the allocated buffer. (Use
  -boundswrite to inhibit warning)
```

Basically, it is triggered due to the fact that the precondition of my_read() assumes the buffer to be at least of size bytes. However, we are sure that reading sizeof(size_t) bytes (8 bytes) in a size_t we are never going out of boundaries. This can be considered a *false positive* indeed, even if we put another annotation like /*@requires maxSet(&len) >= (sizeof(len) - 1)@*/[2] it is redundant with respect to the one written before.

[2]One note about this annotation: Splint triggers a parse error when trying to use the & symbol to denote the address of the variable len. Indeed, even if in the solution we tried to put the annotation, we cannot check whether it works due to the parse error which makes impossible to continue the analysis.

1.3 Static Analysis of `C.c` with Splint

At first impact this program is quite similar to the one analyzed in the previous section. Indeed, launching Splint on the file using the `+posixlib` flag (since we are using the `read()` function) the warnings that are triggered are the same of the previous analysis:

```
C.c: (in function func)
C.c:7:1: Return value (type ssize_t) ignored: read(fd, &len, s...
  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)
C.c:9:10: Possibly null storage buf passed as non-null param:
            read (..., buf, ...)
  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)
   C.c:8:7: Storage buf may become null
C.c:9:1: Return value (type ssize_t) ignored: read(fd, buf, len)
C.c:11:2: Fresh storage buf 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)
   C.c:8:1: Fresh storage buf created
```

Therefore, all the discussion made before about these warnings holds also in this scenario.
However, there is one important difference with respect to the previous program, and it involves the variable `len`. Indeed, since it is read from a file, we do not know which kind of value is going to be passed as size in `malloc()`. In particular, the problem may arise in the following cases:

1. the read value is equal to the maximum number which can be represented using 8 bytes, that is `len` $= 2^{64} - 1$;

2. the value written in the file, whose descriptor is `fd`, is a negative number.

In the first case this is a problem due to the increment of `len` in `malloc()`. Indeed, by incrementing `len`, we are trying to create a number that goes outside the range which can be represented given the number of bits of `size_t` variable (*integer overflow*). This causes a "wrap around" to 0 which, if passed as size to the `malloc()` function either returns `NULL` or a "unique pointer value" (citing the manual page of `malloc()`). While we can check whether there was an error by the fact that it returns `NULL`, it is impossible to distinguish the case when a pointer is returned but the size passed was 0. Indeed, in this latter case, we are in front of a buffer overflow (in particular, it is an heap overflow) because we may try to write on a storage (whose size is theoretically 0) when we should not.
The case 2. arises a problem related to the memory which the OS lets allocate to the user. Indeed, since a `size_t` variable is actually a `long unsigned int`, it cannot be negative. What happens when trying to assign to an `unsigned` variable a negative number, is that the variable will come out to be a (quite) large positive number (almost near the maximum number which can be represented with a `long unsigned int`). Passing this big number as size to `malloc()`, it will return `NULL` most probably because the OS does not allow to allocate such huge amount of memory. To handle this scenario, it should be checked that `malloc()` may return `NULL` and, if it the case, we should safely exit the program.

2 Flawfinder

Flawfinder just like the previously described Splint, is a tool which helps in statically analyzing C programs. It does not need the program under analysis to be executable so, it can be used even if the code is not complete. In addition, it often works also with programs which cannot be even compiled.
Flawfinder does not do any kind of control flow or data flow analysis and it does not even know about the type of function parameters; for this reason, as said before, it is quite naive and simpler with respect to Splint.
Flawfinder is based on a ready-to-use (internal) database called *ruleset* which contains well known risky functions which, if misused, can bring to possible flaws: *buffer overflow* risks (e.g., `strcpy()`, `strcat()`, the `scanf()` family and more of them), *format string* problems (like `printf()`, `snprintf()`), *race conditions*, potential shell metacharacter danger. What it basically does is to take the source code under

analysis and match it against those function names in the DB. Clearly, this is the source of false negatives in using Flawfinder: even if the function is safely used but it matches the name of a function in the internal DB, the tool will continue "complaining" until the `/*flawfinder: ignore*/` comment is written near to the (possibly) unsafe function.
However, it is not so naive because it ignores matches inside comments (so, in some sense it is smarter than a simple `grep` on the file) which eliminates plenty of false positives and also analyses function parameters to give a better estimation of the risk (for example, variable input strings usually are more risky then hardcoded strings).
After this matching phase, the Flawfinder gives a list of "hits" sorted according to the computed risk level. This is an interesting difference with respect to Splint because it gives the possibility of focusing at first on more risky (potential) flaws then going to less risky ones. Clearly, this is not intended to be as excuse to skip "hits" whit a low risk level, but can make the difference when having limited time.
The common point that Flawfinder has with Splint is that both are unsound and incomplete so, not every "hit" can be a vulnerability and not all vulnerabilities are found at all.
Concluding, one of the advantages that Flawfinder can boast, is that it can analyse only changes made on the program (lines that are added or modified). The "patch" file needs to be in a unified diff format but, the real drawback of this approach stays in the fact that it does not notice if lines that strengthened security have been removed.

2.1 Static Analysis of `A.c` with Flawfinder

As default, Flawfinder shows only hits which have a risk level at least of 1. Therefore, launching it on `A.c`, without any further option we are given no warning. However, by setting the minimum risk level to 0 (adding the option `-m 0`), we can see the following warning:

```
A.c:25:  [0] (format) printf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification. Constant format
  string, so not considered risky.
```

As described above, Flawfinder found that there is a call to the `printf()` function known to lead to possible **format string attacks**: this kind of attack can be dangerous when no format argument is given as parameter to `printf()`-like functions. However, since we are telling to format everything as a string, in our case, it is a false positive indeed, we are sure that even if an attacker injects input like a sequence of `%s` trying to crash the program, this won't happen.
The real problem here is that since (in the original file) there is a possible buffer overflow, and since that the same buffer is passed as argument to be printed by the `printf()`, this latter function can be exploited to show if the attack has been successfully completed.

2.2 Static Analysis of `B.c` with Flawfinder

In this case Flawfinder does not give any hit with risk level 0, but two hits both with risk level of 1:

```
B.c:8:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
B.c:13:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
```

In addition, they warn about the same risk: if we are going to use the `read()` function in a loop, we should check that we are not going to write beyond the maximum allocated storage passed as second parameter to the function.
However, given the incompleteness of the tool, this hit can be considered a *false positive* indeed, we are not using the `read()` function in any loop (we are still talking about the original file, not about the solution). On the other hand, the same warning has sense when Flawfinder is launched on the solution proposed below. Indeed, since it is possible that the `read()` returns without having read all the bytes specified as third parameter, we should implement a "mechanism" such that it stops when all bytes are read (apart in case of error). In particular, the proposed solution, implements a `while` loop which, according to the tool may lead to the following problems:

1. when passing parameters in input to the `read()` function, we must be sure that they have been correctly validated. Regarding `buf` we checked that, before being used in this function, it is not `NULL` but correctly allocated. For the third parameter, in one case `sizeof(size_t) = 8` is the number of bytes used to represent a `size_t` variable. For the other case, given all the sanity checks we have done on the variable `len`, when calling `read()` we are sure that `buf` has at least `size` bytes;
2. we must be sure not to write beyond the memory allocated for the `buf` storage. This condition holds because each time we read some bytes, we subtract them from the total: the `while` loop implemented ends only when all specified bytes are read without going any further.

2.3 Static Analysis of `C.c` with Flawfinder

All the considerations we can do on the hits which Flawfinder gives if launched on this file:

```
C.c:9:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
C.c:11:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
```

are the same we did during the analysis of the program in the file `B.c`.

3 Solutions

In this section we present the programs we have statically analyzed above, where all vulnerabilities have been fixed.

3.1 Solution of `A.c` program

Below, we have the correct version of the program `A.c` where all warnings triggered by Splint and Flawfinder and all vulnerabilities have been removed:

```
#include <stdlib.h>
#include <stdio.h>

static void stringcopy(/*@out@*/ char *str1, char *str2)
/*@requires maxRead(str2) >= 0 /\ maxSet(str1) <= 16 @*/;

int main(int argc, char **argv)
/*@requires maxRead(argv) >= 1 /\ maxRead(argv[1]) >= 0 @*/;

static void stringcopy(/*@out@*/ char *str1, char *str2){
    short length = 0;

    while(*str2 != '\0' && length != 16){
        *str1++ = *str2++;
        length++;
    }
    *str1 = '\0';
}

int main(int argc, char **argv) {
    char *buffer;

    if (argc != 2)
        return 0;
    buffer = (char *) malloc(17 * sizeof(char));
    if (buffer == NULL)
        return 0;
    stringcopy(buffer, argv[1]);
    printf("%s\n", buffer); /*flawfinder: ignore*/
    free(buffer);
    return 1;
}
```

3.2 Solution of `B.c` program

The version of the program in the file `B.c` where all vulnerabilities have been fixed, is the following:

```
#include <stdlib.h>
#include <unistd.h>

static int my_read(int fd, /*@partial@*/void *buffer, size_t size)
/*@requires maxSet(buffer) >= (size-1) @*/
{
    size_t bytes_read = 0;
    ssize_t bytes_buf;

    while (size > bytes_read){
        bytes_buf = read(fd, buffer, size - bytes_read);
        if (bytes_buf == -1)
            return 0;
        if (bytes_buf == 0)
            return 1;
        bytes_read += bytes_buf;
        buffer += bytes_buf;
    }
    return 0;
}

void func (int fd) {
    char *buf;
    size_t len = 0;
    int bytes_len, bytes_buf;

    bytes_len = my_read(fd, (size_t *) &len, sizeof(len));
    if (bytes_len == 0 || len > 1024)
        return;
    buf = (char *) malloc(len + 1);
    if (buf == NULL)
        return;
    bytes_buf = my_read(fd, (char *) buf, len);
    if (bytes_buf == 0){
        free(buf);
        return;
    }
    buf[len] = '\0';
    free(buf);
}
```

3.3 Solution of `C.c` program

Last, we have the version of the `C.c` program where all vulnerabilities found by Splint and Flawfinder have been removed:

```
#include <stdlib.h>
#include <unistd.h>

static int my_read(int fd, /*@partial@*/void *buffer, size_t size)
/*@requires maxSet(buffer) >= (size-1) @*/
{
    size_t bytes_read = 0;
    ssize_t bytes_buf;

    while (size > bytes_read){
        bytes_buf = read(fd, buffer, size - bytes_read);
        if (bytes_buf == -1)
            return 0;
        if (bytes_buf == 0)
            return 1;
        bytes_read += bytes_buf;
        buffer += bytes_buf;
```

```
    }
    return 0;
}

void func(int fd) {
    char *buf;
    size_t len = 0;
    size_t max_len = (size_t) (1<<64) - 1;

    int res_len = my_read(fd, (size_t *) &len, sizeof(len));
    if (res_len == 0 || len == max_len)
        return;
    buf = (char *) malloc(len + 1);
    if (buf == NULL)
        return;
    int res_buf = my_read(fd, (char *) buf, len);
    if (res_buf == 0){
        free(buf);
        return;
    }
    buf[len] = '\0';
    free(buf);
}
```