Security in Software Applications

Project 1 - Static Analysis with Splint and Flawfinder

Matteo Mariani

1815188

Contents

1. Splint

Splint is a tool for statically checking C programs for security vulnerabilities and common programming mistakes.
In particular, Splint checks the presence of unused declarations, type inconsistencies, unreachable code, ignored return value, execution path with no return, memory management problems, dereferencing null pointers and so on.
In Splint we have the possibility to put stronger constraints in the code thanks to the use of **annotations**. These are comment-like clauses that can be associated to declarations of variables, to formal parameters of a function or even to its signature.
Combination of both standard checks and annotations can improve dramatically the possibility to recognize some bugs that will stay hidden to standard compilers checks.

Notice that Splint is an *unsound* and *incomplete* tool, meaning that by *default* (i) it may mark as bugs instances which are not actually real problems (it tends to sacrifice maximization of finding bugs to minimize the number of false positives to a reasonable level) and (ii) it doesn't spot all defects actually present in the code.
So Splint is intentionally imprecise, still obtaining false positives and false negatives.
We can tune the degrees of these properties thanks to the use of the annotations, *however* we have to be careful: as the number of annotations rises up, the number of false positive increments as well.

In order to strengthen/weakening/relaxing some checks, Splint permits the use of a multitude of **flags**. Flags are preceded by + or − . When a flag is preceded by + it is *on*; when it is preceded by − it is *off*. The precise meaning of on and off depends on the type of flag.[1]
One of the most important flag is `+bounds`. This enables Splint to check for any instance that may involve the access to a buffer in an imprecise way - reading/writing out of its bounds. This is very useful to avoid some possible *buffer overflows* errors.

2. Flawfinder

Flawfinder is a simple tool for statically searching through C/C++ source code looking for potential security flaws.

Flawfinder uses an internal database called the "**ruleset**" that identifies functions that are common to cause security flaws. The `-listrules` option reports the list of current rules and their default risk level.
This tool works by doing simple lexical tokenization (skipping comments and correctly tokenizing strings), looking for token that *matches* to the database. Then it examines the text of the function parameters to estimate risk.

Flawfinder can find vulnerabilities in programs that cannot be built or cannot be linked. It can often work with programs that cannot even be compiled. It also doesn't get confused by macro definitions.

Once it is applied to some C/C++ source code, Flawfinder produces a list of "*hits*" (potential security flaws), sorted by risk: `5` indicates a high risk function, `0` an harmless one. Hits descriptions have also the relevant *Common Weakness Enumeration* (CWE) identifiers in parentheses.[2]

[1] Splint uses a default flag setting that will be mostly used in the following analysis.
[2] Common Weakness Enumeration (CWE) is a formal list or dictionary of common software weaknesses that can occur in software's architecture, design, code or implementation that can lead to exploitable security vulnerabilities.

Like every unsound and incomplete tool, not every hit is actually a security vulnerability, and not every security vulnerability is necessarily found. In fact it produces many false positives for vulnerabilities and fails to report many others.

Unlike tools such as *Splint*, Flawfinder doesn't use or has access to information about control flow, data flow or data types when searching for potential vulnerabilities or estimating risk level.

3. Code

3.1 A.c

Original code

```
char *stringcopy(char *str1, char *str2){
        while (*str2)
        *str1++ = *str2++;
        return str2;
}

main(int argc, char **argv){
        char *buffer = (char *) malloc(16 * sizeof(char));
        stringcopy(buffer, argv[1]);
        printf("%s\n", buffer);
}
```

3.2.1 Splint

Applying Splint to `A.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ splint A.c
Splint 3.1.2 --- 09 Feb 2018

A.c: (in function stringcopy)
A.c:4:8: Test expression for while not boolean, type char: *str2
  Test expression type is not boolean. (Use -predboolothers to inhibit warning)
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)
A.c: (in function main)
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
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:14:50: Storage *buffer allocated
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)
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)
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:14:50: Fresh storage buffer created
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)
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

Finished checking --- 9 code warnings
```

that have been handled as follows:

- `Test expression for while not boolean, type char: *str2`. Splint warns about the fact that it's used something that is not a boolean expression in the `while` test expression. In this case, we want to permit the loop until the null character (`'\0'`) in the buffer `str2` is found. Even if this check is implicit in the original code, we can explicitly write it in order to clear the warning (`*str2!='\0'`).

- `Implicitly temp storage str2 returned as implicitly only: str2.`. This warning comes from the fact that we're possibly returning a temporary storage. I think that Splint doesn't understand that the return value of `stringcopy` is the same pointer used as argument of the function, so we can clear the warning marking the formal parameter `*str2` with `@returned@` annotation. *However*, once the call ends, `*str2` pointer is useless since it moved at the end of the input string. This strange behavior was handled changing the return value of the function `stringcopy`, as explained below.

- `Possibly null storage buffer passed as non-null param: stringcopy (buffer, ...)` . The warning tells that the variable `buffer` may be null so it should be handled carefully. `buffer` may be null if `malloc` function fails to allocate the necessary memory. Besides, if that happens we can't copy in it any character. In order to solve this, I put a `buffer!=NULL` condition before calling `stringcopy` and I marked the variable with the `@null@` annotation, explicitly saying to Splint that there is the possibility that `*buffer` may be null at some point in the code. As a consequence, the code must check that `buffer` is not null on all paths that may lead to a dereference of the pointer.

- `Passed storage buffer not completely defined (*buffer is undefined): stringcopy (buffer, ...)`. This warning comes from the fact that Splint recognizes as an error when a pointer to allocated but undefined storage is passed as a parameter. However, since this behavior is correct in our code[3] we can just mark as `@out@` the formal parameter `*str1` in order to let know to Splint that the first parameter may be not completely defined.

- `Return value (type char *) ignored: stringcopy(buffer, ...` . Splint warns that the return value of `stringcopy` function is ignored. Considering how `stringcopy` is implemented we can follow two strategies: (i) change the return value of the function to `void`, since returning the modified pointer to source buffer is useless for any purpose[4] or (ii) maintain the original semantic as it is and catch the return value in `main` function. Even if the project is about securing C code, I decided to follow the first strategy since a bad semantic can always lead to sneaky and serious problems that can be hard to spot in more complex projects. For this reason, I changed the return value of the function `stringcopy` to `void`, removing the warning.

- `Path with no return in function declared to return int`. This is an easy one: Splint complains about the fact that there isn't a `return` in `main` function so we can just simply add it.

- `Fresh storage buffer not released before return.`
  `A memory leak has been detected`. The `main` function allocated some memory for `*buffer` but never released it, generating a memory leak. In order to solve the problem I just freed the memory before `main` ends and I marked as `@owned@` the `*buffer` variable. This annotation specifies that only the function that declares that variable is allowed to release the memory associated with.

[3] the buffer will have a defined storage after the call to `stringcopy`.

[4] in fact, the real implementation of the `strcpy` returns the pointer to the beginning of destination string, making an explicit copy of the pointer to destination string before starting the copy.

- `Parameter argc not used`. This is a common warning. For Splint, `main` is a general function so it complains about the fact that `argc` is not used in the code. Since `argc` contains the number of parameters passed in input from the user - name of the program included -, we can use it in order to check whether the user passes a string to copy or not. However, in my solution I choosed a different strategy so I simply marked as `@unused@` the `argc` parameter, cleaning the warning.

- `Function exported but not used outside A: stringcopy`. As Splint suggests we should mark as `static` functions that are not used by other modules. Since this is a dummy C code that has no dependencies with other files, I followed the advice of the tool.

```
[mmariani@dhcppc8 code]$ splint A.c +bounds
Splint 3.1.2 --- 09 Feb 2018

A.c: (in function stringcopy)
A.c:15:8: Possible out-of-bounds read: *str2
    Unable to resolve constraint:
    requires maxRead(str2 @ A.c:15:9) >= 0
     needed to satisfy precondition:
    requires maxRead(str2 @ A.c:15:9) >= 0
  A memory read references memory beyond the allocated storage. (Use
  -boundsread to inhibit warning)
A.c:19:1: Possible out-of-bounds store: str1[max_size - 1]
    Unable to resolve constraint:
    requires maxSet(str1 @ A.c:19:1) >= 15
     needed to satisfy precondition:
    requires maxSet(str1 @ A.c:19:1) >= max_size @ A.c:19:6 - 1
  A memory write may write to an address beyond the allocated buffer. (Use
  -boundswrite to inhibit warning)
A.c: (in function main)
A.c:31:4: Possible out-of-bounds read: argv[1]
    Unable to resolve constraint:
    requires maxRead(argv @ A.c:31:4) >= 1
     needed to satisfy precondition:
    requires maxRead(argv @ A.c:31:4) >= 1

Finished checking --- 3 code warnings
```

Now, let's consider the warnings related to buffers size:

- `Possible out-of-bounds read: argv[1]`. Regardless `stringcopy` is called only if `argv[1]!=NULL` is satisfied, Splint complains about the fact that we're possibly reading something that is out-of-bounds. To explicitly take care of the warning we've to use the `@requires@` annotation on `main` function. This annotation lists all the precoditions that has to be satisfied during the call of the marked function. In order to solve this warning the following precondition is inserted: `@requires maxRead(argv) >= 1@`. This guarantees that the user inserted at least one input, I can read it and consequently copy it.

- `Possible out-of-bounds read: *str2`. Splint warns about the fact that `*str2` may be read out-of-bounds in the `while` test expression. In order to solve this, we can use again the `requires` annotation in the `stringcopy` function (`@requires maxRead(str2)>=0@`).

- `Possible out-of-bounds read: stringcopy(buffer, argv[1])`. This warning is inherited from the `requires` annotation used in the previous step.[5] Since `str2` must have at least one character to read, also the caller must ensure the same property for the passed parameter - in our case `argv[1]`. We should insert an other `requires` annotation in the `main`: `@requires maxRead(argv[1])>=0@`.

In order to solve correctly also the last warning, we have to notice first that a **buffer overflow** may happen: there isn't any check on the amount of characters copied from `argv[1]` to `buffer` so can be that more than `16` characters are put in there. If this happen, once the `printf` occurs all the content of `buffer` is printed in output obtaining information that can be either garbage or something meaningful - this is very unsecure.
In order to solve this problem the `while` test condition was extended (`while (*str2!='\0' &&`

[5] it's common that some warnings appear after have cleaned some others. This happens because putting additional annotations in the code means inserting more constraints to be satisfied.

`i < max_size -1)`, where `max_size = 16`) and the null character explicit assignment was added - `str1[max_size-1]= '\0'`.

- `Possible out-of-bounds store: str1[max_size - 1]`. The warning comes from the fact that the operation `str1[max_size-1]= '\0'` should be handled carefully since we're storing something in a buffer position that may be out-of-bounds. To solve this we used again the `requires` annotation: `requires maxSet(str1)>=15`. This guarantee that `str1` must be large enough to hold a `15` character string.[6]

Furthermore, we can add an `@ensure@` annotation on `stringcopy` function. The `ensures` clause can be used to specify function post conditions, so is the counterpart of the `requires`.
In our case we use it to guarantee that after `stringcopy` ends: (i) the highest index accessible of `str2` (so the position of its "last character") is greater or equal than the one of `str1` and (ii) the highest index accessible of `str1` is lesser or equal than `15`: `@ensures maxRead(str2)>=maxRead(str1) /\maxRead(str1) <= 15 @`.

3.1.2 Flawfinder

Launching Flawfinder to `A.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ flawfinder --minlevel=0 A.c
Flawfinder version 2.0.6, (C) 2001-2017 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examining A.c

FINAL RESULTS:

A.c:18:  [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.

ANALYSIS SUMMARY:

Hits = 1
Lines analyzed = 18 in approximately 0.01 seconds (3597 lines/second)
Physical Source Lines of Code (SLOC) = 12
Hits@level = [0]   1 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   1 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 83.3333 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 0
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
(https://www.dwheeler.com/secure-programs) for more information.
```

Since Flawfinder shows only reported functions that have at least a risk level of `1`, I used the flag `--minlevel=0` in order to see which is the function that is considered quite harmful by the tool.

- `(format) printf: If format strings can be influenced by an attacker, they can be exploited`. A *format string* is a string that contains simple text and some special characters used to convert some variables into readable text. In our code, this warning comes from the following call: `printf("%s\n", buffer)`. When an attacker can modify an externally-controlled format string[7], this can lead to buffer overflows, denial of service or data representation problems. Buffer overflow was exactly what happens due to the bad implementation of `stringcopy`: the `printf` behavior was influenced by the vulnerability present in the `stringcopy` function - in fact when the `printf` occurs the buffer overflow was already done.
  Considering how we managed the buffer overflow problem in the previous section, we have already solved this warning so it's just a false positive.

Since we handled the warning, we can put the comment `//Flawfinder:ignore` near the `printf` in order to suppress it.

[6] without consindering the null character.
[7] as in this case since the string comes directly from the user in `argv[1]` without any form of input validation.

3.1.3 Solution

Secured code

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

static void stringcopy(/*@out@*/char *str1, char *str2)
/*@requires maxSet(str1)>=15 /\ maxRead(str2)>=0 @*/
/*@ensures  maxRead(str2)>=maxRead(str1) /\  maxRead(str1) <= 15 @*/;

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

/*@only@*/ /*@null@*/ void *malloc(size_t size);

void free (/*@only@*/ /*@out@*/ /*@null@*/ void *ptr);


static void stringcopy(char *str1, char *str2){
        int i = 0;
        int max_size = 16;

        while (*str2!='\0' && i<max_size-1){
                *str1++ = *str2++;
                i++;
        }
        str1[max_size-1] = '\0';
}


int main(/*@unused@*/int argc, char **argv){
        /*@owned@*/ /*@null@*/ char *buffer;
        buffer = (char *) malloc(16 * sizeof(char));

        if(buffer == NULL)
                return 0;

        if(argv[1]!=NULL){
                stringcopy(buffer, argv[1]);
                printf("%s\n", buffer); // Flawfinder: ignore
        }

        free(buffer);
        return 0;
}
```

3.2 B.c

Original code

```c
#include <stdlib.h>

void func(int fd){
        char *buf;
        size_t len;
        read(fd, &len, sizeof(len));
        if (len > 1024)
                return;
        buf = malloc(len+1);
        read(fd, buf, len);
        buf[len] = '\0';
}
```

3.2.1 Splint

Applying Splint to `B.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ splint B.c
Splint 3.1.2 --- 09 Feb 2018

B.c: (in function func)
B.c:7:1: Unrecognized identifier: read
  Identifier used in code has not been declared. (Use -unrecog to inhibit
  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)
B.c:13:1: Index of possibly null pointer buf: buf
  A possibly null pointer is dereferenced.  Value is either the result of a
  function which may return null (in which case, code should check it is not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   B.c:11:7: Storage buf may become null
B.c:14:2: 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:11:1: Fresh storage buf created

Finished checking --- 4 code warnings
```

that have been handled as follows:

- `Fresh storage buf not released before return.`
  `A memory leak has been detected.`. As in `A.c`, the function `func` allocated some memory but never released it, generating a memory leak. In order to solve it I just freed the memory before `func` ends.
- `Index of possibly null pointer buf: buf.`
  `A possibly null pointer is dereferenced.`. Splint complains about the fact that we're trying to access through an index some variable that may be null (`buf[len] = '\0'`). To solve the warning we can just add a null check before the assignment.
- `Unrecognized identifier: read`. Splint can't recognize the `read` function so we must import the library that contains it. Through `+posixlib` flag we make visible to Splint any function belonging to *C POSIX library*[8].

[8] `read` is declared into <unistd.h>, that is contained in the *POSIX* library.

Making the POSIX library visible, some other warnings appear:

```
[mmariani@dhcppc8 code]$ splint +posixlib B.c
Splint 3.1.2 --- 09 Feb 2018

B.c: (in function func)
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)
B.c:17:1: Return value (type ssize_t) ignored: read(fd, buf, len)

Finished checking --- 2 code warnings
```

- `Return value (type ssize_t) ignored: read(fd, &len, s...` and `Return value (type ssize_t) ignored: read(fd, buf, len)`. Both warnings comes from the fact that the return values are ignored. `read` function gives a lot of information through the return value: (i) it returns the number of bytes read up to some point[9], (ii) it returns `0` if it read all requested bytes from a file or (iii) it returns `-1` if it failed for any reason. So, the return value should be considered and handled appropriately.
  I decided to change the structure of the code to handle all these possible behaviors, in particular: `ssize_t bytesBuf` is used to collect the number of bytes read by `read` function up to some point and then it's used to increment the total number of bytes read by the caller (`size_t bytesRead`). The loop ends when `read` function returns either `0` or `-1`.
  In this way, the return values are handled and both warnings cleared.[10]

Now, let's consider the warnings related to buffers size:

```
[mmariani@dhcppc8 code]$ splint +posixlib +bounds Bsplint.c
Splint 3.1.2 --- 09 Feb 2018

Bsplint.c: (in function func)
Bsplint.c:11:16: Possible out-of-bounds store: read(fd, &len, sizeof((len)))
    Unable to resolve constraint:
    requires maxSet(&len @ Bsplint.c:11:25) >= sizeof((len)) @ Bsplint.c:11:37
    + -1
     needed to satisfy precondition:
    requires maxSet(&len @ Bsplint.c:11:25) >= sizeof((len)) @ Bsplint.c:11:37
    + -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)
Bsplint.c:22:20: Possible out-of-bounds store:
    read(fd, copy, (len - bytesRead))
    Unable to resolve constraint:
    requires len @ Bsplint.c:15:18 >= len @ Bsplint.c:22:36 - bytesRead @
    Bsplint.c:22:40 + -1
     needed to satisfy precondition:
    requires maxSet(copy @ Bsplint.c:22:29) >= len @ Bsplint.c:22:36 -
    bytesRead @ Bsplint.c:22:40 + -1
     derived from read precondition: requires maxSet(<parameter 2>) >=
    <parameter 3> + -1

Finished checking --- 2 code warnings
```

- `Possible out-of-bounds store: read(fd, &len, sizeof((len)))`.
- `Possible out-of-bounds store: read(fd, copy, (len - bytesRead))`.

Both warnings are related to the fact that we're possibly writing to an address beyond the allocated buffer(s). At the beginning I tried to solve those problems putting some constraints over `read` function as I did for `stringcopy` and `main` in `A.c` code. The annotations that I used were:

```
ssize_t read(int fd, void* buf, size_t size)
/*@requires maxSet(buf) >= size-1 @*/;
```

where the `requires` annotation guarantees that `buf` buffer can contain at least `size` bytes.
Since the code already guaranteed that those conditions were satisfied when both `read` were called, I thought

[9] notice that `read` function can be interrupted by a signal while reading some file.

[10] notice that the first `read` wasn't wrapped in any loop since we're reading just 8 bytes. I decided to `return` in both failure cases.

that this could be the correct way to remove the warnings. *However*, Splint continued to raise them. Considering how the code was modified - see `B.c` solution for more details - and the possibility that Splint could clean those warnings only by modifying the code in a way that could end up to be too cumbersome, I decided to explain why those warnings can be considered false positive in my solution.

- in the first case, can we write to an address beyond the allocated buffer `&len`? Since we're trying to write `size(len)` bytes - so 8 bytes - into a buffer `size_t len` that can contain exactly 8 bytes, we're safe for sure. Moreover, we're not using any loop that can violate this fact.

- in the second case, can we write to an address beyond the allocated buffer `copy`? Since in the solution we call this `read` in a loop, this scenario may happen: we have to guarantee that at each iteration the `(len-bytesRead)` bytes that we're trying to put in `copy` buffer are lesser or equal than the bytes that `copy` can actually store at that iteration.
  At initialization: (i) `copy` is a copy of the pointer `buf` that points to a buffer of `len+1` bytes and (ii) `bytesRead` is an unsigned data type set to 0. At first iteration, we're trying to copy exactly `len` bytes into `copy` and since this is initialized exactly to contain `len+1` bytes, this operation is safe. At each iteration, the number of bytes that should be copied into `copy` is dynamically adjusted: `len` remains always the same *but* `bytesRead` is incremented by a quantity equal to the bytes that the previous `read`(s) has/have read correctly (`bytesRead+=bytesBuf`). Considering that `len>bytesRead` holds in each iteration and `copy` pointer is adjusted with `bytesBuf` at each iteration in order to write correctly the buffer on the next step, we're safe.
  Notice also that `len-bytesBuf` can never be 0 since the loop will end before that due to standard `read` behavior.

3.2.2 Flawfinder

Launching Flawfinder to `B.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ flawfinder B.c
Flawfinder version 2.0.6, (C) 2001-2017 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examining B.c

FINAL RESULTS:

B.c:7:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
B.c:12:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).

ANALYSIS SUMMARY:

Hits = 2
Lines analyzed = 14 in approximately 0.01 seconds (1397 lines/second)
Physical Source Lines of Code (SLOC) = 12
Hits@level = [0]   0 [1]   2 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 166.667 [1+] 166.667 [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://www.dwheeler.com/secure-programs) for more information.
```

- `(buffer) read: Check buffer boundaries if used in a loop including recursive loops`. In our code this warning comes from the following operation: `read(fd, buf, len)`. Considering the description given by flawfinder and CWE-120, the warning says that there is the possibility that when using `read` function we copy `len` bytes to the output buffer `buf` without verifying if it can actually contain at least `len` bytes. However, this is a false positive since `buf` was allocated using `malloc(len+1)`. Moreover, in the original code this operation was not present in a loop.
  On the contrary, even if in our solution the `read` function is present in a loop, boundaries are controlled thanks to the condition `(len-bytesRead)` in `read` function. This allows to dynamically adjust the remaining bytes that need to be read in order to finish the read operation correctly, respecting the

boundaries of the buffer.
An other warning is expressed by CWE-20 related to improper input validation. Consider the parameters passed to `read` function: `len` is not negative (since is an unsigned data type) and is not susceptible to arithmetic overflow (thanks to the check `len <= 1024`), and `buf` is allocated with `len+1`. So, inputs given to `read` function are safe.

- `(buffer) read: Check buffer boundaries if used in a loop including recursive loops`. Same warning as before but on the other read `read(fd, &len, sizeof(len))`. Since the `read` is not used in a loop condition we're safe. Moreover, since we're trying to read 8 bytes (`sizeof(len)`) from a file and we're inserting those bytes into the same `len`, we're correctly respecting the boundaries.

Since both warnings are considered false positives, we can mark them with the comment `//Flawfinder:ignore` in order to suppress them.

3.2.3 Solution

Secured code

```
#include <stdlib.h>

/*@only@*/ /*@null@*/ void *malloc(size_t size);

void free (/*@only@*/ /*@out@*/ /*@null@*/ void *ptr);

void func(int fd){
    /*@null@*/ char *buf;
    char *copy;
    size_t len, bytesRead = 0;
    ssize_t bytesLen, bytesBuf;

    bytesLen = read(fd, &len, sizeof(len)); // Flawfinder: ignore
    if (bytesLen != 0 || len > 1024)
        return;

    buf = malloc(len+1);

    if(buf == NULL)
        return;

    copy = buf;
    do {
        bytesBuf = read(fd, copy, (len-bytesRead)); // Flawfinder: ignore
        bytesRead += bytesBuf;
        if(bytesBuf == -1){
            free(buf);
            return;
        }
        copy += bytesBuf;
    }
    while(bytesBuf != 0);

    buf[len] = '\0';
    free(buf);
}
```

3.3 C.c

Original code

```c
#include <stdlib.h>

void func(int fd){
        char *buf;
        size_t len;
        read(fd, &len, sizeof(len));
        buf = malloc(len+1);
        read(fd, buf, len);
        buf[len] = '\0';
}
```

3.3.1 Splint

Applying Splint to `C.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ splint C.c
Splint 3.1.2 --- 09 Feb 2018

C.c: (in function func)
C.c:7:1: Unrecognized identifier: read
  Identifier used in code has not been declared. (Use -unrecog to inhibit
  warning)
C.c:8:14: 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)
C.c:10:1: Index of possibly null pointer buf: buf
  A possibly null pointer is dereferenced.  Value is either the result of a
  function which may return null (in which case, code should check it is not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
    C.c:8:7: Storage buf may become null
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

Finished checking --- 4 code warnings
```

Since the code is almost identical to the one seen in `B.c`, the warnings found by Splint are the same encountered in `B.c`, so they can be handled in the same manner.

The only difference stays in the absence of the following condition: `if (len > 1024) return;`.
Notice that the value of `len` is taken from a file so it can be anything - even a *negative* number. Since `len` is declared as a `size_t` - that is an `8` byte *unsigned* data type - if some negative value is stored in it, then that value is converted into its complement because how modular arithmetic works.
Paraphrasing: if some small negative value is taken from a file and stored in `len`, `len` will become a huge positive number.

Let's consider two possible cases that may happen: (i) `len` is *almost* equal to the highest value that can be represented with 8 bytes (2^{64}-1) and (ii) `len` is *exactly* equal to the maximum value.

1. in the first case the variable `buf` will not be allocated by `malloc`. This because `len+1` is still a huge number and the malloc fails to allocate something that is quite big as the whole addressable memory in 64-bit systems. However, since a null check on `buf` was already inserted to remove a warning found by Splint, this case is handled correctly.
   Notice that is up to the user to give a *suitable* value for `len` through `fd`: we can insert an hard coded maximum value for `len` in order to guarantee that `malloc` call doesn't fail due the huge memory request (like `B.c` does), but I decided to follow a different strategy.

2. in the second case `len+1=0` (**arithmetic overflow**) and `malloc(0)` returns a pointer that is valid and unique but should *not* be dereferenced. Once `read(fd, buf, len)` occurs we are copying into an "empty" buffer `len` bytes. Since this behavior is allowed by the current code, we get a buffer overflow. In order to secure the code I put an extra test condition soon after the first `read`: if `len == __SIZE_MAX__` then I simply return, otherwise I'm allowed to proceed.

Handling the previous case, we introduced a new warning:

- `Unrecognized (possibly system) identifier: __SIZE_MAX__`. Splint doesn't recognize a lot of the functions/macros found in modern C library so it complains about `__SIZE_MAX__` symbol, that comes as a GNU extension. Splint has a flag (`+gnuextensions`) that enables the support to *some* GNU and Microsoft language extension, *however* this flag doesn't clean the current warning. Since `__SIZE_MAX__` is a well known macro, this is a false positive so I inhibit it through `-sysunrecog` flag.

Now, let's consider the warnings related to buffers size:

```
[mmariani@localhost code]$ splint +posixlib +bounds -sysunrecog Csplint.c
Splint 3.1.2 --- 09 Feb 2018

Csplint.c: (in function func)
Csplint.c:13:16: Possible out-of-bounds store: read(fd, &len, sizeof((len)))
    Unable to resolve constraint:
    requires maxSet(&len @ Csplint.c:13:25) >= sizeof((len)) @ Csplint.c:13:37
    + -1
     needed to satisfy precondition:
    requires maxSet(&len @ Csplint.c:13:25) >= sizeof((len)) @ Csplint.c:13:37
    + -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)
Csplint.c:24:20: Possible out-of-bounds store:
    read(fd, copy, (len - bytesRead))
    Unable to resolve constraint:
    requires len @ Csplint.c:17:18 >= len @ Csplint.c:24:36 - bytesRead @
    Csplint.c:24:40 + -1
     needed to satisfy precondition:
    requires maxSet(copy @ Csplint.c:24:29) >= len @ Csplint.c:24:36 -
    bytesRead @ Csplint.c:24:40 + -1
     derived from read precondition: requires maxSet(<parameter 2>) >=
    <parameter 3> + -1

Finished checking --- 2 code warnings
```

As can be seen, also these warnings are the same seen in `B.c` and they suffer from the same problem pointed out before: we can't clean those Splint warnings without modifying the code in a way that is too cumbersome.
Considering how it was managed the possible arithmetic overflow problem, we can conclude that those warnings can be treated as false positives thanks to the same reasoning done for `B.c`.

3.3.2 Flawfinder

Launching Flawfinder to `C.c` produced the following warnings:

```
[mmariani@dhcppc8 code]$ flawfinder C.c
Flawfinder version 2.0.6, (C) 2001-2017 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 223
Examining C.c

FINAL RESULTS:

C.c:7:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
C.c:9:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).

ANALYSIS SUMMARY:

Hits = 2
Lines analyzed = 11 in approximately 0.00 seconds (2242 lines/second)
Physical Source Lines of Code (SLOC) = 10
Hits@level = [0]   0 [1]   2 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 200 [1+] 200 [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://www.dwheeler.com/secure-programs) for more information.
```

These warnings can be described in the same way done for `B.c`.
The only difference is the following: considering the original code, there was the true possibility of copying more bytes than the ones that `buf` can actually store. This may happen due to the arithmetic overflow in `buf = malloc(len+1)`. However, since we already managed the problem, the code is safe.

3.3.3 Solution

Secured code

```
#include <stdlib.h>

static size_t max_value = __SIZE_MAX__; // 2^64-1

/*@only@*/ /*@null@*/ void *malloc(size_t size);

void free (/*@only@*/ /*@out@*/ /*@null@*/ void *ptr);

void func(int fd){
    /*@null@*/char *buf;
    char *copy;
    size_t len, bytesRead = 0;
    ssize_t bytesLen, bytesBuf;

    bytesLen = read(fd, &len, sizeof(len)); // Flawfinder: ignore
    if (bytesLen != 0 || len == max_value)
        return;

    buf = malloc(len+1);

    if(buf == NULL)
        return;

    copy = buf;
    do {
        bytesBuf = read(fd, copy, (len-bytesRead)); // Flawfinder: ignore
        bytesRead += bytesBuf;
        if(bytesBuf == -1){
            free(buf);
            return;
        }
        copy += bytesBuf;
    }
    while(bytesBuf != 0);

    buf[len] = '\0';
    free(buf);
}
```