SAPIENZA
UNIVERSITÀ DI ROMA

Security in Software Applications Proj 1

Alex Parri - 2140961 - Master's Degree in Cybersecurity

A.Y. 2024/25

Abstract

This is the report for the **first project** of the Security in Software Applications course directed by Daniele Friolo for the Academic Year 24/25 for the Master's Degree in **Cybersecurity** at Sapienza University of Rome. In this homework, the goal was to use the flawfinder tool to **statically analyze** the provided code fragment, **reason** on all of the tool's reports, and finally output a **corrected** version where all found vulnerabilities are removed.

Flawfinder

The flawfinder tool is a **simple, fast and lightweight** command-line utility tool for scanning C/C+ code for known security vulnerabilities. It is great as a **first check** for quickly identifying possible security problems within the code before it is sent to **more thorough** analysis tools.

Its biggest **weaknesses** lie in the fact that it is limited to static analysis only, has a high **false positive** rate, has no context awareness, and that it cannot be relied upon on its own.

Making the code compilable

The provided C code is **not compilable** due to multiple syntax errors, part of which are shown below

```
(proj1venv) alex@alex-MS-7C37:~/uni/ssa/proj1$ gcc project1_SSA24_compilable.cpp -o compiled_project1_SSA24
project1_SSA24_compilable.cpp: In function ‘void func2(int)’:
project1_SSA24_compilable.cpp:21:17: error: invalid conversion from ‘void*’ to ‘char*’ [-fpermissive]
   21 |     buf = malloc(len+1);
      |           ~~~~~~^~~~~~~
      |                 |
      |                 void*
project1_SSA24_compilable.cpp: In function ‘int main()’:
project1_SSA24_compilable.cpp:47:11: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   47 |     func4("fooooooooooooooooooooooooooooooooooooooooooooooooooooo");
      |           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
project1_SSA24_compilable.cpp:56:16: warning: format not a string literal and no format arguments [-Wformat-security]
   56 |         fprintf(stderr, message);
      |         ~~~~~~~^~~~~~~~~~~~~~~~~
(proj1venv) alex@alex-MS-7C37:~/uni/ssa/proj1$ gcc project1_SSA24.c -o compiled_project1_SSA24
project1_SSA24.c:2:9: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or ‘__attribute__’ before ‘<’ token
    2 | include <stdlib.h>
      |         ^
project1_SSA24.c: In function ‘func2’:
project1_SSA24.c:16:1: error: unknown type name ‘size_t’
   16 | size_t len;
      | ^~~~~~
```

In order to have a **more representative** output from the tool, it was decided to make the code compilable by fixing the errors in the **most faithful way** possible. The philosophy behind this idea is that even if every vulnerability was fixed in the noncompilable version, someone would **eventually need** to make the code compilable, which could lead to new vulnerabilities.

Most importantly, the code was converted into C++ to preserve the `trycatch` construct, as it does not exist in plain C. This is **not an issue** as the tool is able to analyze .cpp files either way.

Utilizing the tool

The tool was then ran on the **compilable version** of the code to look for potential vulnerabilities

```
(proj1venv) alex@alex-MS-7C37:~/uni/ssa/proj1$ flawfinder project1_SSA24_compilable.cpp --neverignore
Flawfinder version 2.0.19, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 222
Examining project1_SSA24_compilable.cpp
Warning: Skipping non-existent file --neverignore

FINAL RESULTS:

project1_SSA24_compilable.cpp:40:  [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).
project1_SSA24_compilable.cpp:56:  [4] (format) fprintf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification.
project1_SSA24_compilable.cpp: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.
```

The following is a **rundown** of all the 13 detected warnings ordered by their **[risk levels]**

`1. project1_SSA24_compilable.cpp:40  [4] (buffer) strcpy`

`Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy`

(**True Positive**) This alert is shown because the variable `foo` is copied into `buffer` with `strcpy()` without boundary checking the source's length. This can easily lead to heap buffer overflow and is extremely dangerous to leave as is.

`2. project1_SSA24_compilable.cpp:56  [4] (format) fprintf`

`If format strings can be influenced by an attacker, they can be exploited (CWE-134). Use a constant for the format specification.`

(**True Positive**) This alert raises the issue that the message thrown to handle the `isalpha()` failed check in `func3()` can be exploited by an attacker to perform format string due to missing format specifiers in `fprintf()` in the `catch` code block.

3. project1_SSA24_compilable.cpp:8 [2] (buffer) char

`Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120).`

(**True Positive**) This alert is concerning but not for the reason specified, as `dst` initialization is properly sized, but for the fact that if the attacker could input an arbitrarily long `src` then they could cause the stack to blow, causing a denial of service.

4. project1_SSA24_compilable.cpp:27 [2] (buffer) char

`Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120).`

(**False Positive**) This alert isn't concerning, because `fgets()` correctly prevents the user from inputting more than 1024 characters inside `buffer`. The only problem may be that `\n` is not removed upon copying, which isn't a security issue.

5. project1_SSA24_compilable.cpp:32 [2] (buffer) char

`Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120).`

(**False Positive**) This alert also isn't concerning, because `strncpy()` correctly prevents the user from inputting more than 1024 characters inside `errormsg`.

6. project1_SSA24_compilable.cpp:34 [2] (buffer) strcat

`Does not check for buffer overflows when concatenating to destination [MS-banned] (CWE-120). Risk is low because the source is a constant string.`

(**False Positive**) This alert also isn't concerning because `strcat()` is concatenating a constant string on the destination buffer `errormsg` and there is exactly enough space for it (20 bytes). It is, however, bad practice manually counting the bytes in the string.

7. project1_SSA24_compilable.cpp:8 [1] (buffer) strlen

`Does not handle strings that are not \0-terminated; if given one it may perform an over-read (could cause a crash if unprotected) (CWE-126).`

(**True Positive**) This alert shows that the code assumes that `src` is null-terminated, if so then `strlen()` works as expected, otherwise an attacker that has access to `src` may be able to copy strings that are longer than the code is expecting, leading to undefined behaviour.

8. project1_SSA24_compilable.cpp:9 [1] (buffer) strncpy

`Easily used incorrectly; doesn't always \0-terminate or check for invalid pointers [MS-banned] (CWE-120).`

(**True Positive**) This alert is a concern because `strncpy()` assumes that `src` is a null-terminated string, if so then the terminator `\0` is successfully copied, otherwise it is not, also leading the following null-termination to fail.

9. project1_SSA24_compilable.cpp:9 [1] (buffer) strlen

`Does not handle strings that are not \0-terminated; if given one it may perform an over-read (could cause a crash if unprotected) (CWE-126).`

(**True Positive**) This alert is a problem due to the fact that as already explained in **7.** the code is assuming `src` to be null-terminated, if this isn't the case then `strncpy()` will keep reading bytes from memory until it finds a terminator `\0` or terminates the bytes it has to read.

10. project1_SSA24_compilable.cpp:10 [1] (buffer) strlen

`Does not handle strings that are not \0-terminated; if given one it may perform an over-read (could cause a crash if unprotected) (CWE-126).`

(**True Positive**) This alert is also a concern because as already explained in **7.** and **9.** the code is assuming `src` to be null-terminated, if this isn't the case then the terminator `\0` would be appended later than expected and causing undefined behaviour, including crashes.

11. project1_SSA24_compilable.cpp:16 [1] (buffer) read

`Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).`

(**False Positive**) This alert does not raise such concern because the buffer in question (`len`) is not used within a loop.

12. project1_SSA24_compilable.cpp:22 [1] (buffer) read

`Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).`

(**False Positive**) This alert does not raise such concern because the buffer in question (`buf`) is not used within a loop.

13. project1_SSA24_compilable.cpp:33 [1] (buffer) strncpy

`Easily used incorrectly; doesn't always \0-terminate or check for invalid pointers [MS-banned] (CWE-120).`

(**False Positive**) This alert is very similar to **6.** but despite `strncpy()` not copying the terminator `\0` from `buffer`, in this specific case `strcat()` concatenates a constant string next to it which has it. If this wasn't the case, then it would have been an issue.

Undetected vulnerabilities

Flawfinder has its own *flaws* (pun intended) with unfortunately its own fair share of **false negatives**. The following is a list of **everything** it has missed.

```
// No checks on the return value of read()
void func2(int fd) {
    char* buf;
    size_t len;
    read(fd, &len, sizeof(len));

    if (len > 1024)
        return;
    ...
    read(fd, buf, len);
}
```

The `read()` function could return **partial data or fail**, and no check is performed to ensure its correctness. In the case of failure, data in `len` and/or `buf` may be **garbage or malformed values**.

```
// Potential null pointer dereferecing on buf
buf = (char*)malloc(len+1);
buf[len] = '\0';
```

Due to missing checks on `malloc()` return value, in case of failure the last line will try to dereference a `null` pointer, causing the problem to **crash**.

```
// Potential null pointer dereferecing on src
void func1(char* src) {
    char dst[(strlen(src) + 1) * sizeof(char)];
    ...
}
```

This function assumes that `src` is a non-null string, if this is not the case then the following line will try to dereference a `null` pointer, causing the program to **crash**.

```
// Potential invalid read() call
void func2(int fd) {
    char* buf;
    size_t len;
    read(fd, &len, sizeof(len));
    ...
}
```

The function assumes that `fd` is a **positive integer**, if this isn't the case then `read()` would fail and, being not handled by the program, would result in undefined behaviour.

```c
// Out-of-bounds on first iteration
int main() {
    int y=10;
    int a[10];
    ...
    while (y>=0) {
        a[y]=y;
        y=y-1;
    }
    return 0;
}
```

In this snippet the `while` loop starts iterating on a starting from `a[10]`, which is **outside the array** itself, causing an out-of-bound write on the **very first** iteration.

```c
// No checks on the return value of fgets()
void func3() {
    char buffer[1024];
    printf("Please enter your user id :");
    fgets(buffer, 1024, stdin);
    ...
}
```

The `fgets()` function in above code could return an **error** of an **EoF**, no check is performed to ensure its correctness. In the case of failure, data in `buffer` may be **garbage or malformed values**.

```c
// Potential null pointer dereferecing on buffer
void func4(char *foo) {
    char *buffer = (char *)malloc(10 * sizeof(char));
    strcpy(buffer, foo);
}
```

Due to missing checks on `malloc()` return value, in case of failure the last line will try to dereference a `null` pointer, causing the problem to **crash**.

```c
// Potential null pointer dereferecing on foo
void func4(char *foo) {
    ...
    strcpy(buffer, foo);
}
```

This function assumes that `foo` is a non-null string, if this is not the case then the last line will try to dereference a `null` pointer, causing the program to **crash**.

Corrected version

The following snippet contains the corrected version of the code with **comments** attached

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

void func1(char *src) {
    // added null check
    if(!src) {
        return;
    }

    // added maximum length for security
    const size_t MAX_SIZE = 1024;
    // function assumes src is null terminated
    size_t len = strlen(src);           // Flawfinder: ignore

    // check on maximum length
    if(len > MAX_SIZE) {
        len = MAX_SIZE;
        src[MAX_SIZE-1] = '\0';
    }

    // added +1 for more clarity and removed char
    char dst[len+1];                    // Flawfinder: ignore
    // removed "+ sizeof(char)"
    strncpy(dst, src, len);             // Flawfinder: ignore
    dst[len] = '\0';                    // instead of strlen(dst)
}

// changed "int fd" into "size_t fd"
void func2(size_t fd) {
    // added maximum length for security
    const size_t MAX_SIZE = 1024;
    char* buf;
    size_t len;

    // checking partial read
    size_t n = read(fd, &len, sizeof(len));     // Flawfinder: ignore
```

```
    if (n != sizeof(len)) {
        return;
    }
    if (len > MAX_SIZE) {
        return;
    }

    buf = (char *)malloc(len+1);     // (char *) to make it compilable
    // checking malloc failure
    if(!buf) {
        return;
    }

    // checking partial read
    n = read(fd, buf, len);         // Flawfinder: ignore
    if (n != len) {
        free(buf);
        return;
    }
    buf[len] = '\0';
    free(buf);
}

void func3() {
    // added size variable to make couting easier and more reliable
    const size_t BUF_SIZE = 1024;
    char buffer[BUF_SIZE];                          // Flawfinder: ignore

    printf("Please enter your user id :");
    if(!fgets(buffer, BUF_SIZE, stdin)) {
        // in case of error
        exit(1);
    }
    // buffer is already string terminated
    // here we substitute \n with string terminator instead
    buffer[strlen(buffer)-1] = '\0';                // Flawfinder: ignore

    if (!isalpha(buffer[0])) {
        // added to be sure of not failing the count
        const char* is_not_valid_ID = " is not  a valid ID";

        // changed into malloc() to allow for variable length array
        // is to avoid counting strlen(is_not_valid_ID) manually
```

```
        // not actually a vulnerability, it is to avoid a bad practice
        char *errormsg = (char *)malloc(BUF_SIZE + strlen(is_not_valid_ID)
        ↪  + 1);    // Flawfinder: ignore
        if(!errormsg) {
            exit(1);
        }

        // may not copy string terminator, added for good practice
        strncpy(errormsg, buffer, BUF_SIZE-1);      // Flawfinder: ignore
        errormsg[BUF_SIZE-1] = '\0';
        // this is ok: is_not_valid_ID has '\0' and strcat attaches it
        strcat(errormsg, is_not_valid_ID);          // Flawfinder: ignore
        // added otherwise function doesnt make sense
        throw errormsg;
        free(errormsg);
    }
}

void func4(char *foo) {
    // added null check
    if(!foo) {
        return;
    }

    // added size variable to make couting easier and more reliable
    const size_t BUF_SIZE = 10;
    char *buffer = (char *)malloc(BUF_SIZE * sizeof(char));
    // checking malloc failure
    if(!buffer) {
        return;
    }

    // copies at most 10 bytes
    strncpy(buffer, foo, BUF_SIZE-1);   // Flawfinder: ignore
    buffer[BUF_SIZE-1] = '\0';          // added string terminator

    free(buffer);
}

int main() {
    int i=10;       // renamed into i for better understanding
    int a[10];
```

```
    // does not cause malloc to crash anymore
    func4("fooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo");

    try {
        func3();
    }
    catch(char* message) {
        fprintf(stderr, "%s\n", message);       // added "%s"
    }

    // aFile not declared
    //fprintf(aFile, "%s", "hello world");

    // added predecrement
    while (--i>=0) {
        a[i]=i;
    }
    return 0;
}
```

By also removing abovementioned false positives, the corrected code results in the tool making **no more complains**, as shown in the following screenshot

```
(proj1venv) alex@alex-MS-7C37:~/uni/ssa/proj1$ flawfinder project1_SSA24_compilable_corrected.cpp
Flawfinder version 2.0.19, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 222
Examining project1_SSA24_compilable_corrected.cpp

FINAL RESULTS:


ANALYSIS SUMMARY:

No hits found.
Lines analyzed = 117 in approximately 0.00 seconds (40370 lines/second)
Physical Source Lines of Code (SLOC) = 77
Hits@level = [0]   2 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   2 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 25.974 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Suppressed hits = 11 (use --neverignore to show them)
Minimum risk level = 1

There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
(https://dwheeler.com/secure-programs) for more information.
(proj1venv) alex@alex-MS-7C37:~/uni/ssa/proj1$
```

In conclusion, Flawfinder's purpose is it **find vulnerabilities** so that they can be fixed. However, it shall be said that developers must know how to develop secure software **to begin** with before utilizing it, as *a fool with a tool is still a fool!*