**Strengths and weaknesses of Flawfinder by David Wheeler**

It is fast and it prioritizes the “hits” (potential flaws) based on the risk of the parameters passed to well-known error-prone functions, better than grepping the source code when looking for sinks, and it can analyze code that does not compile. Since it is just a lexical analyzer it does not know anything about control flows, it cannot find logical bugs, memory leaks and produces many false positives.

**Command output**: Flawfinder without useless verbosity (−D).

```
root@bf58ddd26853:/code# flawfinder -D project1_SSA24.c
Examining project1_SSA24.c

project1_SSA24.c:42:  [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.c: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.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.
project1_SSA24.c:28:  [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.
project1_SSA24.c:33:  [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.
project1_SSA24.c:35:  [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.
project1_SSA24.c:8:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
project1_SSA24.c:9:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
project1_SSA24.c:9:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
project1_SSA24.c:10:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
project1_SSA24.c:17:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
project1_SSA24.c:22:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
project1_SSA24.c:34:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
root@bf58ddd26853:/code#
```

**Analysis of the warnings**

- **strcpy()** function at project1_SSA24.c:42 is inside the **void func4(char *foo)** method which allocates a **buffer** of 10 characters in the heap and uses it as the destination parameter of the **strcpy(char *dst, char *src)**, while the source is the string **foo**: **strcpy(buffer, foo)**. The method **func4()** is called in the **main()** using a string of 53 bytes (including **'\0'**) and is much longer than the destination buffer, leading to a heap overflow. There are two ways of fixing this vuln: allocating a **buffer** of **strlen(foo)+1** chars or, if the application logic requires 10 bytes, limit the copy to the first 9 bytes of **foo** and then add the string terminator.
- **fprintf()** function at project1_SSA24.c:56 is inside a broken block of code in the **main()** function. It uses the **try-catch** which is not supported in C, so this is one of the reasons why this code does not compile. Going back to the potential vulnerability, **char *message** is not initialized, therefore it could contain garbage and when passed to the **fprintf(stderr, message)** as format string we have an undefined behavior, depending on the contents of **message**. If **message** was user controlled, then this could have been dangerous, allowing arbitrary memory read/write, but this is not the case. For sure is something to avoid, so fix this code with **fprintf(stderr, "%s", message)**.
- **statically-sized array dst** at project1_SSA24.c:8 is inside the **void func1(char *src)** function. Flawfinder says that statically sizing an array can improprerly restrict the size, leading to overflows. Since this **dst** buffer seems to be used to copy the contents of the **src** buffer, the static size given by **strlen(src)+1** is safe if and only if **src** is null-terminated (but this is part of another Flawfinder warning). In the "good" case it does not crash, **dst** will have the same size of **src** and is acceptable.
- **statically-sized array buffer** at project1_SSA24.c:28 is inside the **void func3()** function. The **buffer** is allocated with size **1024** and is used as destination buffer by the **fgets(buffer, 1024, stdin)** that uses the same size, thus this is safe.
- **statically-sized array errormsg** at project1_SSA24.c:33 is inside the **void func3()** function. In the worst case in which **buffer** has 1023 chars plus the string terminator, after **strncpy(...)** and **strcat(...)**, the buffer **errormsg** will contain 1042 chars plus the string terminator. The static size of the **errormsg** buffer is safe.
- **strcat()** function at project1_SSA24.c:35 is inside the **void func3()** method. Flawfinder says that **strcat()** does not handle overflows when concatenating to destination. In this case, the concatenation cannot cause a buffer overflow because the destination buffer is large enough and the concatenated string is null terminated.
- **strlen()** function at project1_SSA24.c:8 is inside the **void func1(char *src)** method. Flawfinder says that it may perform an over-read if the buffer is not null-terminated. In this case, **func1()** is never called, but it assumes that the buffer is null-terminated, therefore the developer must be sure to have a null byte in the **src** buffer before calling this function, otherwise a segmentation

fault could happen.

- **strncpy()** function at project1_SSA24.c:9 is inside the **void func1(char *src)** method. In this case, even if it is used incorrectly because the third parameter should be **sizeof(dst)** instead of **strlen(src)+sizeof(char)**, it turns out we are lucky that the sizes of the two buffers are the same. It would have been dangerous if the **dst** buffer was smaller or if we were in a multi-threaded context, because if the memory pointed by **src** changed between line 8 and line 9, then the **dst** buffer size wouldn't match anymore, potentially leading to buffer overflow!
  - Based on *https://www.man7.org/linux/man-pages/man3/strncpy.3.html*
  - *NOT* on *https://www.man7.org/linux/man-pages/man3/strncpy.3p.html*, devil in details.
- **strlen()** function at project1_SSA24.c:9 is inside the **void func1(char *src)** method. As mentioned in the previous point, this is not the correct use of the **strncpy()**, therefore we should replace this parameter with **sizeof(dst)**.
- **strlen()** function at project1_SSA24.c:10 is inside the **void func1(char *src)** method. It is good practice to make sure that there is a trailing null byte in the buffer and, again, in this case we are lucky that the two buffers are equal after the **strncpy()** call. However, if we were in a multi-threaded context, the **dst** buffer could be smaller than **src** if the latter changed between line 8 and 9, thus the **dst** buffer wouldn't include - within its size - the null byte after the **strncpy()** call and **strlen(dst)** would read more data, then the code would rewrite a **0x00** byte outside the **dst** buffer (should be already present if the program didn't crash on *strncpy()* because it wrote *strlen(src)+sizeof(char)* bytes). Prefer using **dst[sizeof(dst)-1]=0**.
- **read()** function at project1_SSA24.c:17 is inside the **void func2(int fd)** method. This read is safe because it writes an unsigned long inside the **size_t len** variable by using data coming from a file descriptor.
- **read()** function at project1_SSA24.c:22 is inside the **void func2(int fd)** method. A buffer **buf** is allocated in the heap with size **len+1**, then **len** bytes coming from the file descriptor are written to **buf**. This **read()** is safe and no overflow could occur.
- **strncpy()** function at project1_SSA24.c:34 is inside the **void func3()** method. In this case the function is used correctly, the third parameter is the size **1024** which is less than the size of the destination buffer **errormsg**, so no overflow could occur, plus the source string is automatically terminated by **fgets()**.

#### Vulnerabilities not flagged by the tool

In the **main()** function, at line 49 the dev declared a variable **int y=10** that is used as index for the array declared at line 50 **int a[10]**. The vulnerability arises in the loop that starts from line 60, more precisely at line 61 where the dev writes out of the bounds of the array by accessing the position **a[10]**, but the array is "usable" at indexes that go from **0** to **9**. This is a memory corruption bug that falls into *CWE-787: Out-of-bounds Write*.

### Code fixes comments

Other than fixing the vulnerabilities and improving the readability, the maintenability, I solved all the typos that were present in the original source code (missing # in directives, unbalanced parenthesis, missing semi-colons, missing function types, missing library inclusions, …).

Since I hate magic numbers in the code, I added two macros for the buffer sizes **MAX_BUF_SIZE** and **MAX_ERR_BUF_SIZE**. I added a very simple error handling mechanism: when passing pointers to functions, if these are NULL, return and log the error. Now every **malloc()** have the corresponding **free()**. I tried to give a meaning to **func4()** and the try catch for **func3()** that are used in **main()**.

Instead of using **strncpy()** / **strncat()** and then making sure to have a null terminator at the end of the buffer, I decided to use **strlcpy()** / **strlcat()** that automatically handles the truncation of the copy / concatenation and add a null terminator.

In some cases, I added the **/*Flawfinder: ignore*/** directive in the source code because even after the code fix, the tool was complaining about **strlen()**, **read()** and **statically-sized arrays** (the latter could have been fixed with mallocs, but there is no need) that are clearly false positives. It is ok to add this kind of directive because otherwise, if the code base was large, the dev would have to find the real flaws in a sea of false positives! As a good practice, I suggest running **flawfinder –neverignore source_code** sometimes to recheck all the findings, specially if the code base evolves over time: we want to prevent what has been found by the *iDEFENSE Security Advisory 03.01.05 for RealNetworks RealPlayer (link)*.

### Flawfinder output on fixed code

```
root@bf58ddd26853:/code# flawfinder project1_fix.c
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_fix.c

FINAL RESULTS:


ANALYSIS SUMMARY:

No hits found.
Lines analyzed = 99 in approximately 0.00 seconds (20735 lines/second)
Physical Source Lines of Code (SLOC) = 87
Hits@level = [0]   4 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   4 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 45.977 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0
Suppressed hits = 6 (use --neverignore to show them)
Minimum risk level = 1
```

### Fixed source code

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

#define MAX_BUF_SIZE 1024
#define MAX_ERR_BUF_SIZE 1044
#define LOG_ERROR(msg)                                      \
    do                                                      \
    {                                                       \
        fprintf(stderr, "[ERROR] %s:%d (%s): %s\n",         \
                __FILE__, __LINE__, __FUNCTION__, msg); \
    } while (0)

void func1(char *src)
{
    if(src == NULL)
    {
        LOG_ERROR("src is NULL");
        return;
    }
    size_t dst_size = (strlen(src) + 1) * sizeof(char); /* Flawfinder: ignore */
    char dst[dst_size]; /* Flawfinder: ignore */

    strlcpy(dst, src, dst_size);
}
```

```c
void func2(int fd)
{
    char *buf;
    size_t len;
    read(fd, &len, sizeof(len)); /* Flawfinder: ignore */

    if (len > MAX_BUF_SIZE)
        return;
    buf = (char *)malloc(len + 1);
    read(fd, buf, len); /* Flawfinder: ignore */
    buf[len] = '\0';

    free(buf);
    buf = NULL;
}

char * func3()
{
    char buffer[MAX_BUF_SIZE]; /* Flawfinder: ignore */
    printf("Please enter your user id :");
    fgets(buffer, MAX_BUF_SIZE, stdin);
    if (!isalpha(buffer[0]))
    {
        char *errormsg = (char *)malloc(MAX_ERR_BUF_SIZE);
        strlcpy(errormsg, buffer, MAX_ERR_BUF_SIZE);
        strlcat(errormsg, " is not a valid ID", MAX_ERR_BUF_SIZE);
        return errormsg;
    }
    return NULL;
}
```

```c
char * func4(char *foo)
{
    if (foo == NULL)
    {
        LOG_ERROR("foo is NULL");
        return NULL;
    }
    size_t buf_size = (strlen(foo) + 1) * sizeof(char); /* Flawfinder: ignore */
    char *buffer = (char *)malloc(buf_size);
    strlcpy(buffer, foo, buf_size);

    return buffer;
}

int main()
{
    int y = 9;
    int a[10];

    char *copied = func4("fooooooooooooooooooooooooooooooooooooooooooooooooooo");
    if(copied != NULL)
    {
        printf("%s", copied);
        free(copied);
        copied = NULL;
    }

    char *err_message = func3();
    if (err_message != NULL)
    {
        fprintf(stderr, "%s", err_message);
        free(err_message);
        err_message = NULL;
    }

    while (y >= 0)
    {
        a[y] = y;
        y = y - 1;
    }
    return 0;
}
```