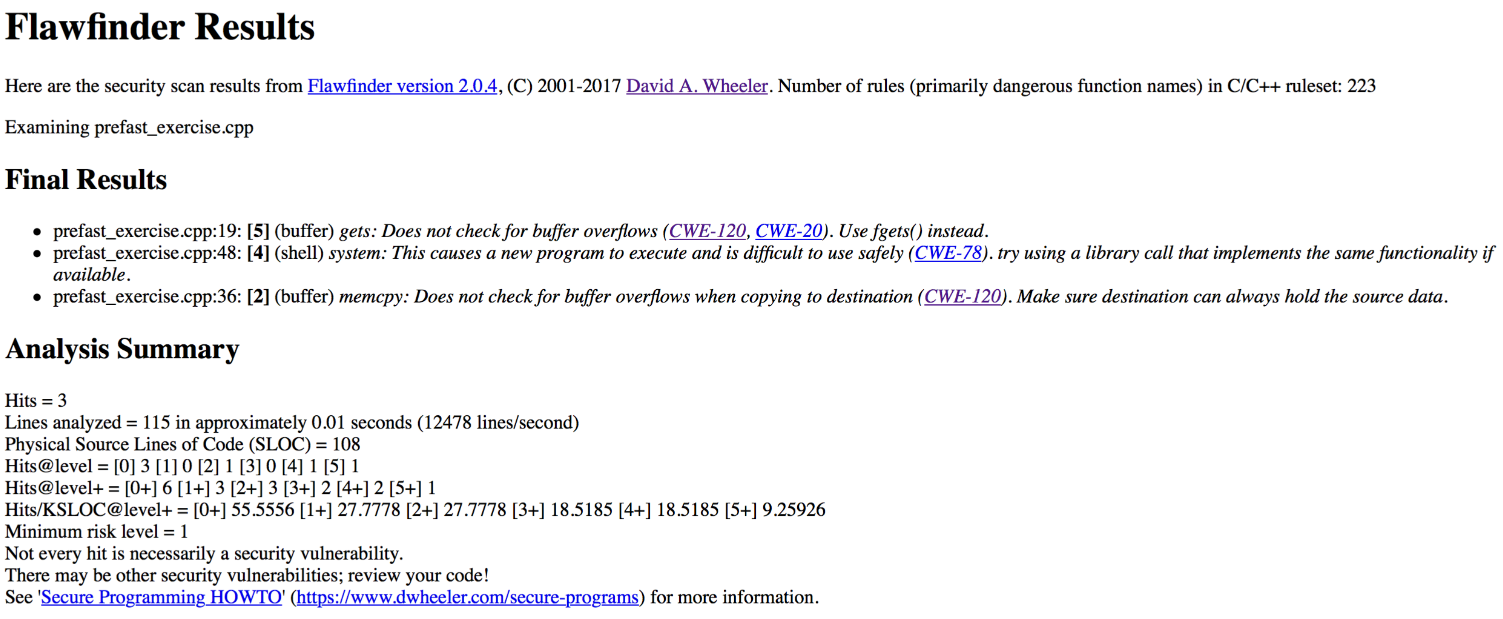
Description of the Tool

Flawfinder is an open-source static code analysis tool (written in Python).
It is used for finding the most common security weaknesses in C/C++ programs.
As mentioned in the Flawfinder official home page, it works primarily by comparing the source code with a built-in database containing C/C++ functions with well-known security problems, such as buffer overflow risks, format string problems, and so on. After the analysis, flawfinder produces a list of "potential" security flaws sorted by risk level and a brief description for each of them.
The tool is confined to analyse the code without execute any statement, however, it has the ability to ignore the text in comments and strings and it is able to adjust the risk level based on the types of function calls parameters (naturally, a constant is often less risky than fully variable parameter).
Among the advantages of the tool must be mentioned the ability to scan C/C++ files rather large in a small amount of time, its ease of use and installation, its portability, and the convenience to export generated reports into HTML files (using the —html option). As every static source code analysis tool, Flawfinder is prone to false positives and false negatives too, even if many of them are avoided thanks to the features described above (like the fact comments and strings are ignored).
One of the main drawback is the inability to check data or control flow, as explained by the author: the tool doesn't really understand the semantics of the code analysed.

Output Description



Flawfinder Results

Here are the security scan results from Flawfinder version 2.0.4, (C) 2001-2017 David A. Wheeler. Number of rules (primarily dangerous function names) in C/C++ ruleset: 223

Examining prefast_exercise.cpp

Final Results

- prefast_exercise.cpp:19: **[5]** (buffer) *gets: Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.*
- prefast_exercise.cpp:48: **[4]** (shell) *system: This causes a new program to execute and is difficult to use safely (CWE-78). try using a library call that implements the same functionality if available.*
- prefast_exercise.cpp:36: **[2]** (buffer) *memcpy: Does not check for buffer overflows when copying to destination (CWE-120). Make sure destination can always hold the source data.*

Analysis Summary

```
Hits = 3
Lines analyzed = 115 in approximately 0.01 seconds (12478 lines/second)
Physical Source Lines of Code (SLOC) = 108
Hits@level = [0] 3 [1] 0 [2] 1 [3] 0 [4] 1 [5] 1
Hits@level+ = [0+] 6 [1+] 3 [2+] 3 [3+] 2 [4+] 2 [5+] 1
Hits/KSLOC@level+ = [0+] 55.5556 [1+] 27.7778 [2+] 27.7778 [3+] 18.5185 [4+] 18.5185 [5+] 9.25926
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.
```

The output generated by Flawfinder, as shown above, can be viewed as two sections: results and a summary of the analysis.

For each “hit” made by Flawfinder a deeper analysis has been done and, when required, a possible mitigation has been described.

Use of **gets** — Is reported a possible buffer overflow vulnerability at line 19 due to the use of `gets()` function, which does not perform boundary checks. The problem can be solved by replacing `gets()` with `fgets()` and modifying the method consequently: adding an integer parameter for the size of the destination buffer and using this new parameter properly as input for the fgets function.

Use of **system** — The second possible vulnerability detected concerns command injection. A buffer is passed through the `system()` function, which spawns a shell and run the command. Unfortunately this function has no direct “safer” alternatives on Windows, so it should be replaced by library functions that performs the same expected actions but, unfortunately, the program seems supposed to run arbitrary user-supplied strings, so the `system()` call would remain the same.

Use of **memcpy** — Is reported a possible buffer overflow vulnerability at line 48 due to a memcpy function call. `memcpy()` doesn’t perform boundary checks and could lead to buffer overflows. It would be better to use memcpy_s, which takes one more parameter as input (the size of the dest buffer) and avoid the buffer overflow conditions. The function call and the method should be consequently modified in order to avoid buffer overflows.

One security weakness that Flawfinder didn’t detect is a possible buffer overflow at line 92. The subroutine `zero`, zero-ing the buffer, doesn’t know what is its length, which could lead to a buffer overflow. Unfortunately, the only way to avoid such a problem is to give to function the right length value for the specific buffer that is passed as input and change the <= with < (or decreasing by 1 the num. of characters to be zeroed out).

Flawfinder’s output ends with a summary of the lines analysed, time spent, other relevant informations and a short reminder for false positives/negatives and with the hope that the code will be “manually” analysed even if a static analyser is used.