Program Analysis with Flawfinder

In this project I had the chance to experiment the capabilities and limitations of static analysis tools for code vulnerabilities detection. I focused myself on the two proposed tools: Flawfinder and PREfast. In particular in this document (SSA P1A) I'll discuss some findings and considerations that I was able to make using Flawfinder.

1 Description of Flawfinder

Flawfinder is a program written in Python that examines C and C++ source code and reports possible security weaknesses sorted by risk level. It is useful to find potential security problems in a given software program. As the creator himself states Flawfinder is not a sophisticated tool. It is an intentionally simple tool, but people have found it useful. When using the tool it is fundamental to remember that not every hit is actually a security vulnerability, and not every security vulnerability is necessarily found.

Flawfinder comes with a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, potential shell metacharacter dangers, and poor random number acquisition. Flawfinder takes the source code text, and it works by doing simple lexical tokenization (skipping comments and strings) and looking for tokens in the database.

1.1 Strengths

As a tool it is very simple but even very easy to use. I found very interesting the feature of sorting the vulnerabilities by risk or filtering out warning which risk is lower than a specified threshold. I imagined how useful this feature could be when one is asked to analyze a very big project with many lines of code. 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.

I compared the warnings I received doing this project with Flawfinder to those of PREfast and I noticed that there's an overlap of one warning, but two out of three warnings were new. This is typical when dealing with multiple static analysis tools, some rules may overlap others may not, so this is not a real strength of this tool in particular but it was definitely useful to use both of them.

1.2 Weaknesses

Flawfinder brings with itself most of the problems related to static analysis tools. Unlike gcc's warning flags, Flawfinder does not use or have access to information about control flow, data flow, or data types when searching for potential vulnerabilities or estimating the level of risk. Thus, Flawfinder will necessarily produce many false positives for vulnerabilities and fail to report many vulnerabilities.

2 Description of the output

In order to run the tool one simply gives Flawfinder a list of directories or files. For each directory given, all files that have C/C++ filename extensions in that directory (and its subdirectories) will be examined. In my case, I simply fed it with C++ code, using the command `flawfinder prefast_exercise.cpp` from the terminal. What I expected was an output listing all the potential vulnerabilities and their associated risk levels, what I got is shown below.

```
MacBook-Pro-di-Matteo:cur_dir matteoprata$ flawfinder prefast_exercise.cpp
Flawfinder version 1.31, (C) 2001-2014 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 169
Examining prefast_exercise_original.cpp

FINAL RESULTS:

prefast_exercise_original.cpp:19:  [5] (buffer) gets:
  Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.
prefast_exercise_original.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_original.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 (11087 lines/second)
Physical Source Lines of Code (SLOC) = 108
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   1 [5]   1
Hits@level+ = [0+]   3 [1+]   3 [2+]   3 [3+]   2 [4+]   2 [5+]   1
Hits/KSLOC@level+ = [0+] 27.7778 [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 for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
```

As we can see before *final results* section of the output we have the **(1) version** of the tool, 1.31 in my case, and some **(2) copyright informations** including the name of the author David Wheeler. Something remarkable is the count of the **(3) number of rules** contained in the Flawfinder database. According to the tool there must be at least 169 functions that cannot be fully trusted in C and C++. With command `flawfinder -- listrules` it is possible to check the content of the database. It is obvious that this small database is far from being exhaustive and furthermore the last update of the database must not be so recent. After the analysis, the trust we should have on our code must also be influenced by these two factors.

The examination of C++ code starts, and after 0.01 seconds 115 **(4) lines of code** get analyzed with a rate of 11087 **(5) lines per second**, as accurately indicated in *analysis summery* section. It may be interesting to mention that the tool specifies also the number of lines of code using *Physical Source Lines of Code (SLOC)* metric, meaning that lines of code containing only comments are ignored from the count.

In the body we can see a **(6) list of hits** that are potential security flaws, sorted by risk. Hits that highlight more serious vulnerabilities are shown first. Each of the hits are presented in the following format:

```
<file name>:<line number>:  [<risk level>] <description of the flaw>
```

The risk level shown inside square brackets varies from 0, very little risk, to 5, great risk. As the documentation states, risk level depends not only on the function, but also on the values of the parameters of the function. This means for instance that functions that take as input constant strings are less risky than those getting as input variable strings.

The output also shows what is the current **(7) minimum risk level** set. Normally Flawfinder shows all hits with a risk level of at least 1, but one can use the `--minlevel` option to show only hits with higher risk levels. It is shown also a **(8) summary of the risks of the hits**, for each of the risks there's associated the number of warnings having that risk. In my example:

```
Hits@level = [0] 0 [1] 0 [2] 1 [3] 0 [4] 1 [5] 1
```

meaning 0 warnings having risk 0, 0 warnings having risk 1, 1 warning having risk 2 and so on. It is next shown the number of hits at a given risk level or larger (so level 3+ has the sum of the number of hits at level 3, 4, and 5). Hits per KSLOC is next shown to indicate hit density, hits per thousand lines of source code. As the creator states, if a program has a high hit density, it suggests that its developers often use very dangerous constructs that are hard to use correctly and often lead to vulnerabilities.

The summary ends with the **(9) reminder**: Not every hit is necessarily a security vulnerability, and there may be other security vulnerabilities not reported by the tool. Which is very important to remember!

3 Corrected Version of the Program

Flawfinder discovers 3 vulnerabilities in the file as shown above. Let us discuss about them.

1.
```
prefast_exercise_original.cpp:19:  [5] (buffer) gets:
  Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.
```

   This warning reminds the programmer that generally `gets` function is very risky. We can see that it has risk lever 5, the maximum. The possibility that this function can become dangerous is very high. Function `gets` is inherently unsafe because it blindly copies all input from standard input to the buffer without restricting how much is copied.

   I simply solved this warning by substituting `gets` with `fgets` which is safer since it allows to specify the maximum number of characters to be read. So line 19 becomes as shown below and the warning disappears when executing Flawfinder again.

```
19:  return (fgets(buf, buf_size, stdin) != NULL) ? SEVERITY_SUCCESS : SEVERITY_ERROR;
```

2.
```
prefast_exercise_original.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.
```

   This warning reminds the programmer that `system` function is quite risky. We can see that it has risk lever 4, it is one step lower with respect to the previous warning. After a deep research on the properties that characterize `system` function, I realized that the risk is liked to the way in which it executes the command. The function `system` will call a root shell as a separate process to execute the command that is sent as an argument, thus making the function very dangerous.

   A possible alternative would be `execv` function which has a similar semantic but it replaces the current process and does not call a shell. Clearly a common problem to both is that, when receiving an unsanitized or improperly sanitized command originated from a tainted source, their behavior could be equally dangerous. Flawfinder database contains `execv` function too, which is reasonable for what just said. It is not possible to easily remove the warning since the task of executing a command may easily introduce vulnerabilities, but using `execv` function would definitely decrease the risk. It is worth to remind that `exec` family of function is available in `unilib.h` header which is only available on Unix-like operating systems.

3.
```
prefast_exercise_original.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.
```

This warning reminds the programmer that `memcpy` function may cause a buffer overflow. It is important to be sure that destination can always hold the source data. What I did to fix this potential vulnerability was to check that we write in the destination buffer a number of characters that is equal to the minimum between the size of the destination and of the source. This will fix the potential vulnerability but it doesn't fix the Flawfinder warning, since Flawfinder for how it work (mostly like a grep), doesn't check the meaning of the if-close that I introduced to solve the vulnerability. So one solution could be:

```
36:  size_t min_size = size1 <= size2 ? size1 : size2;
37:  memcpy(buf2, buf1, min_size);
```

I even tried `memcpy_s` which removes the warning since unlike `memcpy` it can throw errors when destination size is less than source size, so Flawfinder considers it more secure. Eventually I fixed the code in this way:

```
36:  size_t min_size = size1 <= size2 ? size1 : size2;
37:  memcpy_s(buf2, size1, buf1, min_size);
```