Program Analysis with PREfast

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 P1B) I'll answer to the proposed questions by professor about PREfast.

1 Answers to the Questions

1. **One could check ecount or bcount annotations at runtime, rather then at compile-time, as PREfast does. Comparing these alternatives, name one advantage and two disadvantages on doing these checks at runtime.**

   We refer to compilation time to the period in which the code gets translated from higher level language to machine language by a program called compiler. We refer to runtime to the period in which the compiled code is in execution.

   Checking such annotations at compilation time means that the actual size of the buffer isn't specific, it remains a generic parameter, a placeholder. So checking annotations at compilation time will result in an inspection of the program for *all* possible run-time behaviors. The actual value of a parameter is inferred only at running time, when generally happen errors like divisions by zero, dereferencing null pointers and so on.

   One advantage of checking annotations at running time is that it is precise. Because we are working with an instance of the program, we can check easily wether a certain condition holds or not in a given moment. While checking it at compilation time is trickier because a condition may hold or not depending on the instance and thus this may lead to false positives warnings.

   Two disadvantages of checking annotations at running time could be: 1) Loss of generality, we cannot simply trust our program based on the result of running such a test only once, because we may be lucky and find one *good* instance; while to be safe it would be necessary to check the behavior at running time over all the inputs. 2) Negative impact on the performance of the program. Checks of annotations and rules may dominate the running time of an application and thus lead to bad performances, especially when the checks require a lot of computational power.

2. **Sometimes PREfast only warns about problems after you add annotations. For example, it does not complain about zero() until after you add a ecount annotation. An alternative tool design would be to produce a warning about zero() even if there are no annotations for it. Discuss the benefits and drawbacks of this tool design. As part of the discussion, mention at least one advantage of each approach.**

   PREfast warned me about the problem of function `zero(int *buf, int len)` only after I added the annotation `zero(_Out_cap_(len) int *buf, int len)`. With that annotation I bounded buffer `buf` to be of size `len`, so when writing to that buffer that bound shouldn't be exceeded. The function was causing a buffer overrun because of a wrong evaluation of the limit in a for-loop, one cell after the bound could be written. So annotations are reasonable to create a link between the buffer an its size explicitly. This method of annotating links works because it doesn't cause any ambiguity, despite it can be very laborious to annotate huge projects.

If we try to spot the same problems removing the possibility to add annotations, our new tool would need to distinguish between whatever pointer, to a buffer. Moreover it would be needed a method to link each buffer to its size. But for example there's no way to predict with absolute certainty, without annotations, that the parameter next to a buffer is its size. This make this new tool not doable. Annotations are a necessity in C and C++ to spot this kind of problems.

3. **Did PREfast point out problems in the last three procedures in the file? Are they correct? Does this change your opinion of PREfast?**

   PREfast didn't point out any problem to the last three procedures in the file (`zeroing`, `zeroboth2` and `zeroboth`) even though there's an evident problem. In the body of `zeroboth2` function the positions of the lengths of the two buffers, arguments of `zeroboth` are inverted. So in further calls a buffer will be written exceeding its actual size and thus cause a buffer overflow.

   This important problem wasn't highlighted by PREfast since its visibility is constrained by the body of a single function, it doesn't go deeper. It only finds bugs within a single procedure. In fact in our case PREfast has no idea of what may go wrong when `zeroboth2` calls `zeroboth` with those arguments.

   All this doesn't affect my opinion too much. A proper and aware usage of PREfast makes it useful. We shouldn't hope it can find all the vulnerability, if we know how it works. So an aware usage will make the difference in the process of discovering potential bugs.