TIC4303: Lab 3

Introduction

Memory errors in C programs occur due to undefined behavior in C. However, this has many subtle issues and tradeoffs. An undefined behavior is not necessarily an error. Furthermore, defining certain behavior as "undefined" is a **deliberate** choice as it allows a larger set of tradeoffs. This lab is to highlight two perspectives, the compiler's viewpoint, and the programmer's viewpoint (in particular, Linux kernel developers with respect to CVE-2009-1897).

Programmers may not understand that the ways in which the compiler is allowed to optimize the code. In particular, the resulting compiled instructions may not correspond in a straightforward fashion to the original code. The C standard allows the compiler (considerable) leeway to apply many optimizations, some of which can take advantage of "undefined" or "implementation specific" behavior, to get more highly optimized code. This lab shows that what the compiler (`gcc`) does can be quite different from a *naive* interpretation of the C code. A more sophisticated understanding of the C code can end up with `gcc`'s result. We can see that this issue with interpretation of the C code hit the Linux kernel developers with CVE-2009-1897.[1]

Exercise 1 (not graded)

Consider the code in `L4-util.c` and `L4-main.c`. Note that you can compile multiple files separately in one command line, e.g.

```
$ gcc -O1 L4-util.c L4-main.c
```

compiles to executable `a.out` and at optimization level `O1`.

The idea of the code in `f()` is that it does a double check that there is no null pointer being passed. For a certain reason (in the original Linux kernel code, there was a bug), it reads out the content of `p->b[1]` and saves into a local variable `x`. Because `x` is not used afterward, related code can be eliminated by the optimizer. For the null-pointer check, the programmer may asume (from his/her viewpoint), that it should be kept no matter whatever optimization schemes are applied. (Note: this view is *technically incorrect*). From the compiler's viewpoint, any allowed assumption, can be used to optimze the code (it depends on the optimization level). Thus, the compiler may generate binary code different from the programmer's expectation.

The `gcc` C compiler has optimization levels from `-O0` (default, when `-O` is not used) to `-O3`. Compile and run the code with all choices of the optimization levels. Do you see any differences in behavior? The differences in behavior are due to the compiler optimizations taking a particular interpretation of "undefined behavior" in C. This is completely **legal** by the C standard (even if it may confuse programmers). In later labs, we will use the LLVM `clang` compiler. Please revisit this lab with `clang`. Does `clang` behave the same as gcc on all optimization levels?

Due to CVE-2009-1897, `gcc` added a flag to control optimizations which may delete null dereferences, `-fno-delete-null-pointer-checks`. Note that the default for `gcc` is `-fdelete-null-pointer-checks` (without the "no"). After this CVE, the Linux kernel added this flag to the build process.

[1] So even experienced programmers may make such a mistake, not just junior programmers.

Exercise 2 (not graded)

Examine what happened with CVE-2009-1897. Some URLs:

`https://bugzilla.redhat.com/show_bug.cgi?id=512284#c5`
`https://lkml.org/lkml/2009/7/6/19`
`https://lwn.net/Articles/341773`