Lecture 24: Good Practices for Safe Programming in C

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/24-good-practices.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

The Problem

In this unit we described the benefits of C, as well as multiple use cases demonstrating that these languages are still extensively used. We also saw that programs written in C are prone to memory issues which is concerning for security. How can we avoid (as much as possible) the programming mistakes that lead to these security issues? Here we discuss a few tools that can help. And we will also see a few guidelines about how to write good code.

Extra Compiler Warnings

A first piece of advice is to enable the many compiler warnings that are disabled by default. This can help detect bugs. It can be done by using the following flags:

- `-Wall` will enable a first set of warnings.
- `-Wextra` will activate additional warnings on top of that.
- `-pedantic`, will enable warnings forcing you to write C code that conforms strictly to the ISO C standard.

All these options activate warnings that are particularly strict, and the programmer will need to reason a bit about the potential issues they point in the code. Are these things bugs or not? For example, `-Wall` will warn about variables that are declared but not used. While it is probably fine to have some during development, it is suspicious for e.g. a production build.

Analysis Tools

Code and program analysis tools generally fall within be classified into two main categories. **Dynamic analysis** tools work by instrumenting your program with additional checks and observing its behaviour at runtime. **Static analysis** tools analyse the code or other representations of your program (binary, compiler intermediate representation) without running it.

- In terms of dynamic analysis tools, we already discussed Valgrind, that focuses on the heap and is very practical to check for memory leaks, uninitialised memory reads, etc. There is also another popular tool named address sanitiser or ASAN, that can detect overflows, use after frees, leaks, etc. To use ASAN simply add the following compiler flags and run the program normally:

```
$ gcc -fsanitize=address -fno-omit-frame-pointer program.c -o program
$ ./program
```

The program will crash when ASAN detects a bad memory access, with a log of exactly what went wrong

- In terms of static analysis tools, a popular one is the Clang static analyser. Clang is the C frontend of a compiler named LLVM[1]. The analyser can check for things like division by zero, null pointer dereferences, usage of uninitialised values, etc. Other static program analysis tools include Frama-C[2](https://frama-c.com/), Coccinelle[3], Cppcheck[4] and many others[5].

An important thing to note is that static and dynamic tools have their pros and cons, and there is no silver bullet. No tool will detect all the memory errors in all programs. Moreover, even if one runs all the possible tools on a given program, it is not guaranteed that all the errors will be detected. (next slide)

Writing Good Code

Even if some tools can help, it is of course very important to write good code. The compiler does not catch all mistakes and the tools are not perfect. Writing good code will save a lot of debugging time, and of course will reduce the chances of introducing security vulnerabilities.

Undefined Behaviour

In C there is the notion of **undefined behaviour**. Once the program enters undefined behaviour, all the programmer's assumption are off: although the program may seems to work fine under certain circumstances, **it's a bug, and it needs to be fixed**:

- The program can crash, which in some sense is a good outcome because it forces the programmer to investigate and fix the issue.
- The program can behave in a strange way, which may be harder but is still possible to detect.
- But the program can also *seem* to execute fine, for various reasons, for example maybe the particular condition of a memory error are not encountered. This kind of difficult to detect bugs are the worst, and are the reasons why vulnerabilities sometimes live for decades undetected in codebases[6].

Memory errors lead a program into undefined behaviour. Examples include:

- Reading an uninitialised variable.
- Reading/writing out of the bounds of an array.
- Dereferencing a `NULL` (`0x0`) pointer.
- Overflows in signed integer arithmetic.
- Dereferencing a freed pointer.
- Freeing a pointer twice.
- etc.

Array/Buffers Sizes, Integer Overflows

Arrays in C do not embed their sizes, and it is the programmer's responsibility to keep track of the sizes. This is true for any array, including strings, as well as all types of buffers. If a function takes an array or a buffer as parameter, its size should probably also be passed. You should also be aware of the sizes of different types on the architecture you target to avoid overflows. Recall that `sizeof()` gives you that information. Unsigned overflow wraps over to 0 but signed overflow is undefined. In case of doubt, use wider types for arithmetics, check for overflow and if the result is indeed within bounds convert back to the smaller type.

The C standard Library

Concerning the C standard library, it is good to check out the `man` pages for its functions. Some libc functions allocate results with malloc, and it may be your responsibility to free the corresponding space. Also, these functions should never be used, they are almost always unsafe:

[1] https://llvm.org/
[2] [https://frama-c.com/]
[3] https://coccinelle.gitlabpages.inria.fr/website/
[4] https://cppcheck.sourceforge.io/
[5] https://en.wikipedia.org/wiki/Category:Static_program_analysis_tools
[6] https://blog.qualys.com/vulnerabilities-threat-research/2022/01/25/pwnkit-local-privilege-escalation-vulnerability-discovered-in-polkits-pkexec-cve-2021-4034

- `gets` (use `fgets`).
- `getwd` (use `getcwd`).
- `readdir_r` (use `readdir`).
- More here: https://github.com/intel/safestringlib/wiki/SDL-List-of-Banned-Functions.

String Manipulation Functions

- Regarding string manipulation functions, generally you should use the versions including `n`, that not only check for `\0` to find the end of a string but also allow a programmer-defined character limit. Use `strncpy` rather than `strcpy`, `snprintf` rather than `sprintf`, and so on. Even with the `n` functions, be careful about some particularities: for example `strncpy` does not ensure that the target buffer is terminated by `\0`. Consider the following code:

```
char string1[] = "hello, world";
char string2[32] = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
strncpy(string2, string1, strlen(string1));
printf("%s\n", string2); /
```

This will print `hello, worldxxxxxxxxxxxxxxxxxxx`, which is unlikely what the programmer intended.

Dynamic Memory Allocation

A few things regarding dynamic memory allocation:

- `malloc` return value should always be checked to avoid NULL pointer usage.
- After free, the pointer is invalid:
  - It cannot be dereferenced, (that is relatively obvious)
  - But even its value itself, the previously pointed address, cannot be used anymore for example for comparison with other addresses.
- When it fails, `realloc` returns `null` but does not free the old pointer. So, in effect, the following code is a memory leak:

```
ptr = realloc(ptr, new_size);
```

With this code, if `realloc` fails, the old address pointed by `ptr` is overwritten by `null` and, assuming no other pointer points to that space, it won't be possible to free it.

Going Further

This guide contains further advice:
https://docs.fedoraproject.org/en-US/defensive-coding/programming-languages/C/.