Bughunting in util-linux

Stefan-Octavian Radu

April 23, 2023

1 Introduction

In this report I present the process I went through to find various bugs in the `util-linux` collection of Linux utilities. I start by giving a bit of information about the tools used. After that, I describe the runtime analysis process using the `valgrind` memory checker and describe my findings. I finish the section with a solution to the identified issue. I go on to explain the static analysis process using the `cppcheck` tool. I give examples of many false positive findings, but also of a positively identified bug. In the end I present the conclusions of the experiment.

2 Preliminaries

For this analysis I used the memory checker tool that is bundled with `valgrind` [1] for runtime analysis and the `cppcheck` [2] tool for static analysis.

2.1 Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. There are also standard tools provided as part of Valgrind itself, which can be used for instrumenting programs. When using Valgrind for instrumentation, it runs the binary of the program in an sandboxed environment, on a virtual synthetic CPU. This allows the Valgrind core to pass the code to the specific tool that is in use. Memcheck, which is the specific tool referred to in this report, inserts code which checks every memory allocation and release made during the execution of the program. By tracking memory flow in this manner it can determine various types of memory related issues, such as memory leaks, or unreleased memory. It also provides detailed stack traces which aid in pinpointing the exact source of each problem. [3]

To use valgrind with the Memcheck tool, you just have to call the program and provide as an argument the path to your executable: `valgrind ./<binary>`. To get a more comprehensive result I also used two extra flags: `valgrind --leak-check=full --show-leak-kinds=all`

2.2 CppCheck

CppCheck is a static analysis tool designed for C/C++ programs. It claims to be different from other similar tools because it uses unsound flow sensitive analysis [2], as opposed to path sensitive analysis which is used by most other tools. At a high level, flow sensitive means that the tool will take into account the order of operations in the program and will compute an answer for each point in the program. Moreover it is considered unsound, meaning that it doesn't guarantee that all errors that exist in the program will be reported, despite the error types being among the ones supported by CppCheck [6].

To use CppCheck you just have to call the program and provide as an argument the path to the code you want to analyse and, optionally, an extra argument with the path of any `include` directory. In my specific use case I used the following command in the root of the project: `cppcheck -I ./include ./`.

3 Runtime Analysis

3.1 About the target

My analysis started from the `cal` terminal utility. Running the `valgrind` memory checker on the already installed instanced I could see that a few thousand kilobytes of memory were either lost or not properly freed upon the program exist. `cal` is a simple utility program that runs in the terminal and displays a calender with the current day highlighted using the `ncurses` library. It is part of a bigger suite of CLI utilities and programs bundled together in the `util-linux` repository [7]. The repository hosted on Github, which this paper is based on is used for specifically for development.

3.2 Investigation process

The first step was to clone the repository and compile the targeted program. Running `valgrind` on the compiled program leads to the results shown in Listing 1.

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 17,760 bytes in 20 blocks
        suppressed: 0 bytes in 0 blocks
```

Listing 1: Innitial Leak Summary

Seeing `still reachable` memory is sign of improper handling and freeing of allocated memory before a program exits. Using the additional flags `--leak-check=full` and `--show-leak-kinds=all` I can get a stack trace which allows me to track the calls which caused the leak.

Following the stack trace and testing the code along the way I understood that while initializing the necessary elements needed for colour support, there is also a check performed that verifies if the current terminal in use sup-

ports colours.

```
// cal.c
if (colors_init(ctl.colormode, "cal") == 0)
    ...
// colors.c
if (cc->mode == UL_COLORMODE_UNDEF
    && (ready = colors_terminal_is_ready()))
    ...
// colors.c
if (setupterm(NULL, STDOUT_FILENO, &ret) == 0
        && ret == 1)
```

Listing 2: Stack Trace

The `setupterm` function is part of the `ncurses` system library. While consulting its manual [4] I learned that, as the name suggests, `setupterm` is a routine that handles initialization of various low-level terminal-dependant structures and variables. Upon initialization, the `cur_term` global is set to point to the newly initialized memory segment. Releasing this memory however, is the responsibility of the developer who should call `del_curterm`. A careful inspection of the code shows that `del_curterm` is not called while using the `cal` program.

3.3 Solution

The solution I came up with was to ensure that when `setupterm` is called, `del_curterm` will also be called. For this I used `atexit` from the standard C library which calls a provided function when the program exits. I thus created a wrapper around the `del_curterm` function and pass it as an argument to the `atexit` call as seen in Listing 3.

```
/* atexit() wrapper */
static void colors_del_curterm(void)
{
  del_curterm(cur_term);
}
...

if (setupterm(NULL, STDOUT_FILENO, &ret) == 0
        && ret == 1) {
    ...
    atexit(colors_del_curterm);
}
```

Listing 3: Wrapper for atexit

3.4 Results

Following the proposed change, I recorded its effects by running `valgrind` again on the newly compiled binary. As shown in Listing 4, I managed to reduce the improperly released memory by 9640 bytes in a total of 15 blocks.

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 8,120 bytes in 5 blocks
        suppressed: 0 bytes in 0 blocks
```

Listing 4: Leak Summary after changes

Since the problem originated from the `colors.h` library, this change affects not only the `cal` utility, but also any other program from the repository which imports `colors.h`. This includes common utilities such as `fdisk`, or `hexdump`.

3.5 Further efforts

As it's obvious from Listing 4, there is still unreleased memory when the process finishes execution. By following the cues from `valgrind` I concluded the remaining problems originate in the `tigetnum("colors");` as well as in low level code in the `ncurses` library. After unsuccessful attempts of modifying to code from the `ncurses` library in a meaningful way and some more research on the topic, I came across an interesting finding in the ncurses FAQ page [5]. It seems that reports regarding memory still in use in programs which depend on the `ncurses` library are normal and expected. There are certain chunks of memory which are never freed for performance reasons.

The final verdict regarding the 5 blocks of memory still in use is thus inconclusive. There is a high probability that not properly releasing the respective blocks is intentional, but I couldn't find clear evidence for this.

4 Static Analysis

4.1 About the target

I thought the best course of action for the Static Analysis part was to verify as much code as possible. As such, I decided to use the CppCheck program on all the code that was available in the repository, thus checking for bugs in every available tool.

4.2 False Positives

Calling `cppcheck` in the root of the project and filtering for errors outputs more than 400 issues with the codebase. On a closer inspection I found the following:

- The majority of issues reported were "null pointer dereference" errors. Most of them were reported in scenarios where a function was called with a 'struct type' as a parameter. All turned out to be false positives.
- There were a few reports of "uninitialized variable". Analysing the specific scenarios showed that the respective variables were initialized in some initialization function to which the corresponding address was passed. False positive.
- There was one report of "double free". In that particular case the freed pointer was either reallocated or set to `NULL`, which wouldn't cause any issues. False positive.
- There was one report of "index out bounds access". In that particular case, there was an explicit boundcheck wrapping the memory access. False positive.
- There were a few "integer overflow" errors reported. All of the cases involved shifting to the left the number

1 with a value (`1<<K`), which turned out to be less than or equal to 31. This wouldn't cause an integer overflow. False positive.

- There was a very interesting "null pointer dereference" report which I emphasized in Listing 5. This is actually a declaration using the `__typeof__` construct called on a null pointer variable. This however is not incorrect. False positive.

```
__typeof__(ask->data.num) *num;
```

Listing 5: Declaration using ___typeof___

4.3 One identified bug

Out of the more than 400 reports, more than 99% of them were false positives. However, there was one report of "path with no return statement". Analysing the source code led me to conclude that a bug is indeed present there. Looking at Listing 6 we can see that there are 3 return paths handled but 4 present. If the conditional evaluates to false and none of the pre-processor checks hold, then the function will end without returning. This leads to undefined behaviour, which must be avoided. This is especially severe as it occurs in a function which deals with authentication related issues.

To solve this issue, an appropriate return statement should be appeneded to the function.

```
...
  if (su->suppress_pam_info
    && num_msg == 1
    && msg
    && msg[0]->msg_style == PAM_TEXT_INFO)
        return PAM_SUCCESS;

#ifdef HAVE_SECURITY_PAM_MISC_H
        return misc_conv(num_msg,
            msg, resp, data);
#elif defined(HAVE_SECURITY_OPENPAM_H)
        return openpam_ttyconv(num_msg,
            msg, resp, data);
#endif

    /* return expected */
}
```

Listing 6: Path with no return

CppCheck claims to report very few false positives. It is very surprising that despite this claim, more than 99% of my finding were false positives. I attribute this result to the surprising differences that can occur between the sytax of a piece of code and its semantics.

5 Conclusion

In this report I presented my efforts of finding bugs in the `util-linux` collection of Linux utilities. With the use of the `valgrind` memory checker I managed to identify and also fix an improper handling of initialized data upon the program exit. This previously resulted in a certain amount of memory not being released upon the program exit. Furthermore, with the use of the `cppcheck` static analysis tool, I managed to find a non-void function which has an execution path that doesn't end with a return statement. This can lead to undefined behaviour. This experiments prove that the use of both runtime and static analysis tools is invaluable during development. This is especially relevant when programming languages such as C, which do not have any memory safety guarantees.

References

[1] URL: `https://valgrind.org/` (visited on 04/23/2023).

[2] URL: `http://cppcheck.net/` (visited on 04/23/2023).

[3] URL: `https://valgrind.org/docs/manual/manual-core.html` (visited on 04/23/2023).

[4] URL: `https://man.archlinux.org/man/curs_terminfo.3x.en` (visited on 04/23/2023).

[5] Thomas E. Dickey. Mar. 16, 2022. URL: `https://invisible-island.net/ncurses/ncurses.faq.html#config_leaks` (visited on 04/23/2023).

[6] Michael Hicks. Oct. 23, 2017. URL: `http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/` (visited on 04/23/2023).

[7] *util-linux*. URL: `https://github.com/util-linux/util-linux` (visited on 04/23/2023).