20875 Software Engineering – Assignment 2

Due Sunday, November 14th 2023, 23:59

The assignment consists in finding and fixing bugs in the `GLPK` project. Specifically, we are targeting its latest version, `GLPK 5.0`, which can be downloaded at: `https://ftp.gnu.org/gnu/glpk/glpk-5.0.tar.gz`

GLPK is an open-source library for solving linear programming (LP) and mixed-integer programming (MIP) problems. It reads a problem instance from a file. That file must be in one of 5 supported formats. Then, it solves the problem. The parsing code for some of the formats contains bugs. Your task is to find such bugs, explain them, and propose fixes.

In what follows, a bug can be a crash, an assertion failure, any type of undefined behavior invocation, a memory leak, or a wrong result.

Write your answers directly on Blackboard (under "Assignment 2") by the end of November 14th.

Question 1. In this question, we target the parsing code for the "fixed MPS" and "free MPS" formats.

a. [1 mark] Create or find a file that triggers a bug in GLPK when parsed as an MPS-formatted file (either "fixed MPS" or "free MPS"; specify which format you used). If this file is smaller than 100000 bytes, upload it. Otherwise, describe the specificities of this file that make it trigger a bug in GLPK.

b. [1 mark] Explain the *consequences* of the bug and where they happen in the source code of GLPK (in which file and at which line).

   Example: "A crash happens at line 1234 in `example/example.c` in the function `cleanup()` because the pointer variable `p` is `NULL` and the expression `p[i]` dereferences a `NULL` pointer."

c. [2 marks] Determine the *causes* of the bug. Specifically, explain the chain of events that happens, starting from the particularities of the input file, and ending with the bug consequences described above.

   Example: "When an index is negative in the input file, the allocation at `example/example2.c` line 5678 in the function `initialize()` attempts to allocate a negative amount, fails, and returns `NULL`. This `NULL` pointer is then stored in `struct DATA *d`, specifically in the `d->pointer` field. When `cleanup()` is called, that pointer is stored in `p`, yielding the `NULL` pointer dereference."

   Be as general as possible in your description of the range of circumstances that can lead to the bug. For example, "when an index is negative" is more general than "when an index is -1".

   Be concise. The objective is that a reader familiar with `GLPK` understands just enough to then propose a bug fix. Between 2 and 5 sentences should suffice.

d. [2 marks] Propose a *fix* for the bug. Explain your strategy to address the root cause of the problem. You can include proposed changes to the source code (using `diff -u` or `git diff`). As an alternative, detail your proposed changes in plain English.

   The fix must be general, and correctly address the underlying problem. In most cases, just removing an assertion is not an appropriate fix (although it can happen in rare cases that an assertion is wrong).

   Be concise. The objective is that a reader familiar with `GLPK` understands just enough to then apply your proposed code changes. Between 1 and 3 sentences should suffice.

Question 2. In this question, we target the parsing code for the "GNU MathProg" format.

a. [1 mark] Create or find a file that triggers a bug in GLPK when parsed as a GNU MathProg file. If this file is smaller than 100000 bytes, upload it. Otherwise, describe the specificities of this file that make it trigger a bug in GLPK.

b. [1 mark] Explain the *consequences* of the bug and where they happen in the source code of GLPK. Same as Question 1b.

c. [2 marks] Determine the *causes* of the bug. Same as Question 1c.

d. [2 marks] Propose a *fix* for the bug. Same as Question 1d.

Bonus question. Find additional bugs in `GLPK` and fully describe them (and fix them) just as in Questions 1 and 2. One bonus mark per bug, up to a maximum of 3.

Rules. This is an individual assignment. You are allowed to talk about the assignment with your classmates. However, you must find input that triggers bugs on your own. You must also write your answers on your own. As a consequence, you will be able reproduce and explain all bugs upon request. **Do not contact the developers of `GLPK`.**

Hints:

A. General hints.

1. If you get stuck at any stage, immediately contact your lecturer and/or TA.
2. Try to keep notes of what you do and how you overcome issues. This will be useful later if you have to perform the same steps again.
3. It is probably better to prepare your answers in a text file and copy/paste them into Blackboard when you are done.
4. Your first steps should be
   a. Download and build AFL++.
   b. Read the documentation of AFL++ about how to use it effectively.
   c. Download and build `GLPK`. If you have trouble building it with AFL++ instrumentation enabled, try a normal build first.
   d. Understand how to use `GLPK` from the command line (the executable is `glpsol`).
   e. Run `glpsol` on an example file to check that it works.
   f. Start fuzzing `GLPK`.

B. Compiling

5. Use `GLPK`'s `./configure` script options to choose a compiler (e.g. AFL++'s compile-with-instrumentation scripts) and to choose compiler options (e.g. enabling ASan or UBSan).
   Run `./configure --help | less` to see how to.
6. Fuzzing will only work if you compile `GLPK` with *static* linking. Disable *dynamic* linking and enable static linking in the options of `./configure`. It may be convenient to create a small script that calls `./configure` with your chosen options.

7. Whenever you re-run `./configure`, you need to run `make clean` to make sure that the next compilation will re-build everything with your updated options.
8. The `-j` option of `make` allows parallel (hence faster) compilations. For example, use `-j 8` if your computer has 8 parallel threads.
9. The `glpsol` executable is in the `examples/` directory.
10. If `glpsol` was already installed on your computer, beware of *which* `glpsol` executable you are running (you probably want to run the one you just compiled).
11. MacOS-specific instructions
    1. Make sure your OS is up-to-date (e.g. Sonoma 14.1).
    2. Install homebrew
    3. Follow precisely the MacOS-specific instructions here: https://aflplus.plus/docs/install/
    4. Use `afl-clang-fast` as a compiler
    5. Despite the warnings, when fuzzing GLPK, AFL++ runs very fast on M1 and M2 Macs.

C. Fuzzing

12. The AFL++ documentation has many tips for effective fuzzing.
13. The `examples/` directory in the `GLPK` source code contains many `.mps` files in the "fixed MPS" format (`--mps` option of `glpsol`), and `.mod` files in the GNU MathProg format (`--math` option of `glpsol`). For example files in the "free MPS" format, see `https://miplib2010.zib.de/miplib3/miplib.html`.
14. For faster and more effective fuzzing, discard large input files, as well as input files that `GLPK` is slow to parse.
15. It is also easier to understand bugs when the input files are small.
16. One way to get smaller GNU MathProg files is to remove comments.
17. While fuzzing, you can compile with UBSan enabled to find more bugs, but do not enable ASan (it creates issues when used with AFL++). Once you found a crash, you can recompile with ASan enabled to help track down the bug (or you can use `valgrind` instead).
18. Since we focus on the input parsing code, use the `--check` option of `glpsol` to only run that code (for faster fuzzing).
19. AFL++ may give you warnings about your input size, or your computer configuration. In such cases, read the instructions carefully. You can choose to address the issues or ignore them (the instructions explain how the warning can be discarded).

D. Understanding and fixing bugs

20. Compilation with AFL++ instrumentation is slower. If you repeatedly modify the `GLPK` code and recompile it, it will be convenient to compile it without AFL++ whenever not fuzzing (remember to run `make clean` before or after `./configure` every time).
21. Assertion failures call `abort()`. A call to `abort()` ends a program. When run with `valgrind`, you get a stack trace at the point `abort()` was called (i.e. the functions that were called leading to `abort()`).
22. Feel free to modify the source code of `GLPK` to get more information about bugs. You can insert `printf()` statements, modify logic, etc.
23. The `grep` command is very useful for navigating a foreign codebase.

24. The GNU MathProg parser contains multiple bugs. If a bug seems too difficult to understand, try to find an other one. One way to make the fuzzer ignore a bug is to insert a regular `exit()` call before it is triggered. This will allow the fuzzer to find other bugs.

25. The memory dedicated to storing the stack has a fixed limited size. **Stack overflow** bugs happen when the stack exceeds that allocated size. The two most common causes are:

    a. An out-of-bounds access to an on-stack array (an array stored in a local variable in a function).

    b. Infinite (or excessive) recursion (when a function calls itself, directly or indirectly).

26. In case of stack overflow, ASan only displays the "top" of the stack, i.e. the most-recently called functions in the call stack, as opposed to the whole call stack (with `main()` at the "bottom").

27. You do not need to fully understand the `MPS` or `GNU MathProg` input formats. However, it will be useful to know *just enough* about the formats to better understand `GLPK`'s parser code.