Graded lab assignment: Memory safety

1 Overview

Description: In this exercise, you are going to analyze a simple but real JSON parser written in C. The parser contains one injected fault, which results in a buffer overread, and one real bug, which results in a buffer overwrite. The buffer overread occurs on the heap while the overwrite occurs on the stack.

You are going to use the *address sanitizer* (activated by the `-fsanitize` compiler switch) to detect the bugs. A correct version of the program should trigger no reports from that tool.

1.1 Installation, running tests

1. Unpack the sources:
   ```
   tar -xzf json.tar.gz
   cd json/
   ```
2. Compile the sources with `-g` (debug information):
   ```
   make
   ```
3. Compile with `-fsanitize` options:
   ```
   make fsanitize # address sanitizer
   make fsanitize-all # show all bugs ("recover" after violation)
   ```

1.2 fsanitize usage

To activate the address sanitizer, you have to compile the sources with a special compiler switch. This will *instrument* the code with extra instructions to analyze and guard memory accesses.

Note: If your code does not get re-compiled when it should be, use
```
make clean
```

Run the instrumented binary with the example input:

```
./parser test-comment-error.json # or a different example
```

You will see an error trace similar to this simplified output:

```
=================================================================
==72829==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x60e0000000da at pc 0x000103b729c4 bp 0x7ffeeceac550 sp 0x7ffeeceabcd8
READ of size 155 at 0x60e0000000da thread T0
    #0 0x103b729c3 in printf_common(void*, char const*, __va_list_tag*)
    #1 0x103b7336c in wrap_printf
    #2 0x102d575f2 in main parser.c:157
    #3 0x7fff71859cc8 in start+0x0
0x...da is located 0 bytes to the right of 154-byte region [0x...40,0x...da)
allocated by thread T0 here:
    #0 0x103b9717d in wrap_malloc
    #1 0x102d573f8 in main parser.c:136
    #2 0x7fff71859cc8 in start+0x0
SUMMARY: AddressSanitizer: heap-buffer-overflow in
printf_common(void*, char const*, __va_list_tag*)
Shadow bytes around the buggy address:
  0x1c1bfffffc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c1bfffffd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c1bfffffe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c1bffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c1c00000000: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
=>0x1c1c00000010: 00 00 00 00 00 00 00 00 00 00[02]fa fa fa fa
  0x1c1c00000020: fa fa fa fa fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c1c00000030: fd fd fd fd fd fd fd fa fa fa fa fa fa fa fa fa
  0x1c1c00000040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c1c00000050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c1c00000060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==72829==ABORTING
/bin/sh: line 1: 72829 Abort trap: 6           ./parser test-comment-crash.json
```

2 Analyzing the error

For the mandatory part of this exercise, analyze and fix this error first, and then the second error (the stack buffer overflow) reported by fsanitize.

Deliverables: For the mandatory part, submit the following:

1. Defect report: Write a short report that describes what the fsanitize report means. Which buffer is overrun? Look at the definition of the data type and how it is used. What happens if the data is not in the expected format?
2. Bug fix:
   (a) Create a copy of the source file you are going to modify.
   (b) Fix the program so fsanitize reports no errors when the parser is used with the second example:
   `./parser test-comment-crash.json`
   (c) Upload the report and a patch that fixes the program to Canvas.

2.1 Reading the report from fsanitize

The key parts of the report generated by fsanitize appear early on: The type of problem, the stack trace of the read or write access, and the stack trace of the memory allocation. These parts are highlighted in bold above. Open the source file(s) shown in the report and look at the two reported code locations (of the memory access and the allocation) to understand the problem.

2.2 Heap overread

The first bug is a buffer overread caused by `printf` (see the report from fsanitize for details). To fix the bug, keep in mind the following:

1. Strings in C must be terminated by a `NUL` ('\0') character.
2. The size of a string buffer must therefore be large enough to hold the actual string and the terminating character.

A fix involves changing the code in two separate places (one line of changes each).[1]

2.3 Stack overwrite

The second bug is only triggered with a specially formatted input file. It overruns an internal buffer. Check again the report:

1. The *write* happens in `sprintf`; this instruction writes formatted data to a string buffer, taking in this case two numbers and a string from another position in memory.
2. The buffer is stack-allocated in this case, and of fixed size. The report points to the beginning of the function (`json.c:241`), but the variable in question is easy to find.

[1] It is not necessary to provide a *minimal* fix to pass this exercise, but if you end up making many changes, it may be better to start over and look for a simpler solution.

A "perfect" fix that ensures that the string buffer is large enough would require dynamic memory allocation. In this case, it is enough if you write an error message of a fixed length, up to `json_error_max`.

To prevent a buffer overwrite, there is a safe companion function to `sprintf`: **`snprintf`**. That function takes an additional parameter that corresponds to the maximum number of bytes to be written in total. The final `NUL` character is included in this count, so `snprintf` writes at most one byte less than the maximum and automatically appends the `NUL` character in the right place, regardless of the length of the input.

This second error can therefore be fixed by changing one line of code.

3 Optional assignment: fuzzing

Tasks:

1. Install radamsa from its home page: `https://gitlab.com/akihe/radamsa`
   You do not need to run `make install`.
2. Set the environment variable RADAMSA so it points to the executable. Example:
   `export RADAMSA=${HOME}/dd2460/radamsa/bin/radamsa`
3. Create a small example (a small but valid JSON input file) of how `parser` can be used. Let radamsa perturb it.

Hints:

1. Radamsa can fuzz an existing file with the following command:
   `${RADAMSA} initial-fuzzing-input.json > input.json`
2. Run the instrumented `parser input.json`; repeat these two steps until you see an error.

You can also use the script `fuzz.sh` that automates this and try it with different initial inputs (see below).

Deliverables:

For the extra point, you also have to submit a description of fuzzing:

1. How did you create the original example (seed) data?
2. How many attempts did it take to reproduce the problem?
3. Is reproducing the problem with fuzzed data easy or hard? Why? If your example (seed) had a JSON comment (or not), how successful do you think fuzzing would be in the case without (or with) a short comment?

4 Grading criteria

Pass (G): The first part (mandatory assignment) is solved correctly.

VG (extra point): In addition to a correct and timely solution of the mandatory part, all optional parts solved correctly and submitted on time.