Secure and Dependable Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-566
Date: 2024-02-19
Due: 2024-02-26

Problem Sheet #2

Problem 2.1: *test coverages* (1+1+1+1+1 = 5 points)

The following Rust function calculates the Levenshtein distance between two strings, which is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one string into the other.

```
use std::cmp::min;

/// Calculates the Levenshtein distance between two strings.
///
/// # Arguments
///
/// * 'str1' - The first string
/// * 'str2' - The second string
///
/// # Examples
/// ```
/// let n = lved::lved::lved("hello", "world");
/// assert_eq!(n, 4);
/// ```
///
/// For more details, see the Wikipedia article:
/// https://en.wikipedia.org/wiki/Levenshtein_distance

pub fn lved(str1: &str, str2: &str) -> usize {
    let s1 = str1.chars().collect::<Vec<_>>();
    let s2 = str2.chars().collect::<Vec<_>>();

    let s1_len = s1.len() + 1;
    let s2_len = s2.len() + 1;

    let mut matrix = vec![vec![0; s1_len]; s2_len];

    for i in 1..s1_len { matrix[0][i] = i; }
    for j in 1..s2_len { matrix[j][0] = j; }

    for j in 1..s2_len {
        for i in 1..s1_len {
            matrix[j][i] = if s1[i-1] == s2[j-1] {
                matrix[j-1][i-1]
            } else {
                1 + min(min(matrix[j][i-1], matrix[j-1][i]), matrix[j-1][i-1])
            };
        }
    }
    matrix[s2_len-1][s1_len-1]
}
```

Your task is to define minimal test cases by adding tests (using the `assert_eq!()` macro) to the following test file.

```rust
#[cfg(test)]
mod tests {
    use crate::lved::lved;

    #[test]
    fn lved_func_coverage_tests() {


    }

    #[test]
    fn lved_stmt_coverage_tests() {


    }

    #[test]
    fn lved_branch_coverage_tests() {


    }

    #[test]
    fn lved_condition_coverage_tests() {


    }

    #[test]
    fn lved_boundary_interior_path_coverage_tests() {


    }
}
```

a) Which tests are necessary to achieve function coverage? Explain.

b) Which tests are necessary to achieve statement coverage? Explain.

c) Which tests are necessary to achieve branch coverage? Explain.

d) Which tests are necessary to achieve condition coverage? Explain.

e) Which tests are necessary to achieve boundary interior path coverage? Explain.

Problem 2.2: *clang libfuzzer* (3+2 = 5 points)

The `clang` compiler support a fuzzying API, which makes it very easy to fuzz C functions. Below is a simple example:

```c
#include <stdint.h>
#include <stddef.h>

static int memcmp(void *s1, const void *s2, size_t n)
{
    unsigned char *a = (unsigned char *) s1;
    unsigned char *b = (unsigned char *) s2;

    for (int i = 0; i < n; i++) {
        if (a[i] < b[i]) {
            return -1;
        }
```

```
        if (a[i] > b[i]) {
            return 1;
        }
    }
    return 0;
}

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    char *msg = "FUZZ";
    (void) memcmp(msg, data, size);
    return 0;
}
```

By compiling the code with `-fsanitize=fuzzer`, you obtain an executable that will feed fuzzed inputs to the function `LLVMFuzzerTestOneInput()`, from where you can call any function you want to test. It is usually a good idea to enable additional `clang` sanitizers by compiling the code with `-fsanitize=fuzzer,address,undefined`.

a) Fuzz the example shown above. What is the test case found by the fuzzer that causes the implementation of `memcmp()` to fail? What is the problem here? Explain.

b) Take a function of medium complexity that you wrote in the past and which is processing strings. (In the operating systems course you likely wrote a function (as part of the word guessing game) that selects a random word in a text string, which is then replaced by underscore characters and the word is returned as an allocated copy, `char* hide_word(char *text)`.) Implement a suitable fuzzying wrapper and report which bugs were found (if any).