Task 4 - Enhancing Library Robustness

In task 3 we have seen, that `fputs` crashes for nearly all "unusual" input values. In this task we will make `fputs` as robust as possible without touching the source code of `fputs` or the calling application.

Contents

Motivation

Do you remember our last experiment?

Each real application needs to behave robust which means there are general requirements for unspecified input values:

- „do not crash“
- „do not lose consistency“
- ...

```
Test crashed by signal 11
Startung TEST fputs (pointer to inaccessible memory, point to ...
Test crashed by signal 11
Tests ended.
8 out of 96 behaved robust (8.33%)
88 out of 96 crashed (91.67%)
0 out of 96 stopped (0.00%)
0 out of 96 timed out (0.00%)
# > _
```

The problem is that your application (must) use libraries, which are not as robust as you need them to be. But in general you cannot increase the robustness of libraries themselves (e.g. you don't have the source code).

There are of two alternative ways to increase the robustness of our programs:

- The first is to enhance the robustness in the program itself. This is very error-prone. You will have to adapt your application for each new version of your imported libraries. Furthermore you have to deal with different installations, different library versions with different levels of robustness. And you need access to the applications source code.

- The second alternative is to insert a robustness wrapper. The advantage is that you will not need to change the application or the library.

Wrapper Cookbook

Create a shared library - your wrapper:
`wrapper.cpp`

Implement each library function you want to wrap:
```
int fputs (const char* s, FILE* f) { ... }
```

- use exactly the same signature
- obtain the pointer to the original function:
```
typedef int (*Func_fputs)(const char*, FILE*);
Func_fputs org_fputs = dlsym (RTLD_NEXT, "fputs");
```
- check the input arguments:
```
if (f == null)
```
- run the original function if they are ok:
```
return org_puts (s, f);
```
- otherwise return a defined error code:
```
return EOF;
```
- compile your code as a shared library
```
g++ -fPIC -c -Wall wrapper.cpp -o wrapper.o
g++ -shared wrapper.o -ldl -o wrapper.so
```
- use the environment variable **`LD_PRELOAD`** to force your Unix/Linux to load the wrapper library before all other libraries, if you start your application
```
LD_PRELOAD=./wrapper.so ./yourApp
```
- the best way is to use a start script
```
#!/bin/bash
export LD_PRELOAD=$LD_PRELOAD:./wrapper.so
./yourApp
```

Your Task

In task 3 you built a tool for testing a C-Library function. The first step was to build a routine for a single test on a single function. Step 2 was to generate test values for testing one function.

The current task is to build a robustness wrapper for brittle library functions.

You shall build a robustness wrapper for the function **`fputs`** and test the wrapper with your robustness test from the previous task. **`fputs`** must not crash for any input. Instead **`fputs`** has to return **`EOF`** for each "non-robust" input.

Detect non-robust input values

You will have to treat both input values independently from each other with different testing strategies.

File-Handles

Whenever a file pointer does not point to a valid file handle returned by `fopen`, `freopen` or `fdopen` function `fputs` crashes. Furthermore the structure of the file pointer must be read- and write-able. So you must implement two checks for file handles:

1. maintain a list of all open file handles: `fopen`, `freopen`, and `fdopen` wrappers add to the list of open file handles. Remove closed handles in the `fclose` wrapper; a file handle is valid if it is in that list
2. check if the memory of the file pointer is read- and write-able; the size is given as the size of `FILE`

Strings

The string argument is read by `fputs`. So you must check that all bytes including the trailing `\0` character are readable.

Robust Test of Read- and Writability

The file handle and the string argument of `fputs` both need a robust test of memory read- and writability. You should use `setjmp` and `longjmp`, as well as `signal`:

- use `signal` to set the signal handler for signal 11 (segmentation fault) to your own signal handler; save the old signal handler
- call `setjmp` in an if statement
- in the "then branch" you read or write (with reading) the memory to test respectively (read until you find a `\0` character or read and write back `sizeof(FILE)` bytes, respectively); after this check you know, that the memory is accessible
- if the memory is not accessible, signal 11 will be raised and your signal handler is called
- execute a `longjmp` in your signal handler. This brings you to your previous `setjmp` call
- the `setjmp` now returns 1, so you go into the "else branch" of your if; there you know that the memory is not accessible
- in both cases: restore the original signal handler

Optimization: most hardware platforms only enforce memory access permissions **per page**. It is sufficient to test on byte per page. You do not need to implement that.

Hints

Here are some hints that might help you solve this task:

- read the man pages of `fopen`, `fclose`, `freopen`, `fdopen`, `dlsym`, `setjmp`, `signal` and `longjmp`
- test your wrapper with your solution of task 3
- you can maintain a list of open file handles using a C++ container class, e.g. **`std::set<T>`**
- You need Unix/Linux for this task, because of the **`LD_PRELOAD`** mechanism (Windows, even Cygwin does not provide it).
- if GCC complains about "unknown variable `RTDL_NEXT`" you can define it as:
  `#define RTDL_NEXT ((void*)-1)`

Summary

Write a shared library which wraps the function **`fputs (const char*, FILE*)`**

- your **`fputs`** must not crash for any passed parameter.
- call **`fputs`** for all "safe" arguments
- read the man page about **`fputs`** for the expected return values in the case of an unsafe argument

Test the wrapper with your robustness test for **`fputs`**.

Organizational Remarks

Deadline

Solve this week's task at home or in the lab. Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified day we will regard your solution as missing so you will not receive your certificate.

Directory Structure

We require a specific directory structure for your solutions. You get it with the initial svn check-out.

```
wrapper/
    Makefile              - Makefile
    wrap_fputs.cc         - C++ code to wrap C function fputs
```