INFO-0940: Operating Systems

Project 1: Tracer - Report

Goffart Maxime
180521

Joris Olivier
182113

Academic year 2020 - 2021

1 Implementation

In this section, we will present our implementation. We will start with the structure of the code. Then, we will briefly explained the data structures that we are using.

1.1 Code structure

Our implementation is composed of multiple source and header files. Our files are the following:

- `tracer.c` The file which contains the main function of the tracer. Based on the arguments given, it starts the adequate mode (profiler or syscall).
- `syscall{.c,.h}` Represents the mode of the program that traces the system calls made by the tracee.
- `profiler{.c,.h}` Represents the mode of the program that traces the function calls made by the tracee.
- `file_sys_calls{.c,.h}` Represents the module that allows to load the file that contains the mapping between the ids and the names of the system calls.
- `functions_addresses{.c,.h}` Represents the module that allows to load the mapping between the functions' names and their addresses. It requires nm and objdump to work.
- `load_param{.c,.h}` Represents the module that allows to parse the arguments of the program.

1.2 Data structures

Throughout the project, we are using multiple data structures. Here is a list of the data structure that we are using:

- In `profiler{.c,.h}`, we have:
  - `Profiler` represents the data of the profiler. After running `run_profiler(char*)`, it contains the data of the profiling.
  - `Func_call` represents a function called by the tracee. An explanation of the different fields is available in the source file.
- In `file_sys_calls{.c,.h}`, we have `FileSysCalls` which represents the mapping between the system calls' names and ids.
- In `functions_addresses{.c,.h}`, we have:
  - `FunctionAddresses` which represents, as a linked list of `Mapping`, the mapping the between the functions' symbols and addresses.
  - `Mapping` which represents, as a node of a linked list, a mapping between a function's symbol and address.

2 Specific questions

In this section, we will answer to the questions asked in the section 6 of the assignment.

1. **What are the opcodes that you have used to detect the `call` and `ret` insturctions?**
   To detect the `call` instruction, we used the opcodes: 0xe8, 0x9a, 0xffd$x$, 0xff1x, 0xff5x, and 0xff9x where x is an hexadecimal digit.
   To detect the `ret` instruction, we used the opcodes: 0xc2, 0xc3, 0xca, 0xcb, 0xf3c3, and 0xf2c3.

2. **Why are there many functions called before the "real" program and what is/are their purpose(s)?**
   Their purposes are to set everything in order for the process to be runnable (call to __libc_start_main) including setting up the thread inside the process (call to __pthread_initialize_minimal).

3. **According to you, what is the reason for statically compiling the "tracee" program?**
   In order to get the functions' names (including the ones from libraries), we are using nm and objdump. These two programs rely on the ELF file generated by the compilation process[1]. Yet, dynamic libraries are loaded at runtime where static libraries are integrated in the executable during the linking phase. Because we need the functions from libraries to be in the executable, due to the usage of nm and objdump, it forces us to use static libraries, which themselves implies static linking.

3 Estimation of spent time

Unfortunately, we were working, at the same time, on other projects, so we do not have a clear idea of how much time we spent on this project in particular.

[1]Here, by compilation process, we mean preprocessing, compiling, assembly, **and** linking.