CIA Lab 2 Assignment

Abstract
This week we will be looking at how an operating system starts up and related computer architecture.

Note: Since this lab is huge please be straight to the point, the grades of bonus will be counted in case you finished the regular tasks.

1. Booting the OS

Now that we have Ubuntu installed, we will look at the booting process.

1. What is a UEFI OS loader and where does the Ubuntu OS loader reside on the system? Hint: See the UEFI specification.
2. Describe in order all the steps required for booting the computer (until the OS loader starts running.) Ubuntu uses the UEFI OS loader to load and run the GRUB boot loader.
3. What is the purpose of the GRUB boot loader in a UEFI system?
4. How does the Ubuntu OS loader load the GRUB boot loader?
5. Explain how the GRUB boot loader, in turn, loads and run the kernel by answering these 3 questions:
    a. What type of filesystem is the kernel on?
    b. What type(s) of filesystem does UEFI support?
    c. What does the GRUB boot loader, therefore, have to do to load the kernel?
6. Do you need an OS loader and/or boot loader to load a Linux kernel with UEFI? Explain why or why not.
7. In an MBR-based system, the GRUB bootloader is distributed over the disk in multiple parts.
    a. How many parts (or stages) does GRUB have in such a system, and what is their task?
    b. Where are the different stages found on the disk?

2. Initializing the OS

After the machine has booted up, the chosen OS is started and goes through a (large) number of steps before you end up at the login prompt. We will be looking at the startup procedure of Ubuntu.

Questions:

1. Describe the entire startup process of Ubuntu in the default installation. The subquestions below are leaders to help you along, they must be answered but by no means represent the entire startup process of Ubuntu. . .
    a. What is the first process started by the kernel?
    b. Where is the configuration kept for the started process?
    c. It starts multiple processes. How is the order of execution defined?
    d. ...

3. Basic OS interaction

Now we will look at the different formats for executables, We use the “strace” command to identify system calls in binaries and we look at (parts of) the ELF file format.

➢ **Binaries and scripts**

Make a table listing five different binaries or interpreter scripts, and their type (Hint: use the “file” command in “/usr/bin”)

➢ **Tracing binaries**

1. Read the “strace” man page.
2. Use “strace” to find what other systems call besides “stat” “zsh” uses before executing an “execve” system call to start a given binary (e.g. “/bin/ls”)? (Hint: use the “-c” option of “zsh”)

➢ **Stracing strace**

1. Run “strace /bin/pwd” and save the result
2. Also run “strace strace /bin/pwd” and save the result. How are these outputs related? Explain the “2” in “write(2, ...”.

➢ **ELF format**

1. Read the “readelf” man page. Make sure you make your terminal as wide a possible.
2. Execute “readelf -Wh <<your favorite ELF binary>>”. Match the results for the ELF header with the information on ELF’s Wikipedia page. Is this a definitive source for the ELF format? If not, what is?
3. Also inspect the section and program headers, using the command “readelf -WlS <<your favorite ELF binary>>”.
    a. Explain which of the two header types (section or program) is used in which context (loader/runtime or linker/relocation)?
    b. Explain the use of each of the following sections: .text, .data, .rodata and .bss. (Hint: Consider Type and Flg)
    c. What does the program header with Type INTERP contain?
4. Read your favorite ELF binary with hexdump utility.
    a. show the virtual address and physical address.
    b. show/explain the entry point of the ELF binary.
5. Find the loaded libraries of your shell by executing “lsof -p $$”. Also, look at the memory image of your shell by executing “cat /proc/$$/maps”. Why is for instance the libc library mapped into memory multiple times?

4. The gcc compiler

Let us consider “Hello SNE!” - program:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Hello SNE!\n");
  return 2021;
  /* What is the actual exit status? Why? */
}
```

➢ Make your own version of the “Hello SNE!”-program, by just only varying the string. Can you come up with a creative text? Find all four phases in the compilation by running “gcc -v -o hello hello.c”.

Now we are going to do all the stages separately in order to inspect the different intermediate steps:

1. Run “gcc -E hello.c -o hello.i”, producing the file “hello.i”.
   What are the numbers at the end of the lines in “hello.i” supposed to mean?

2. Run “gcc -S hello.i”, producing the file “hello.s”
   Inspect “hello.s”. Where has “printf” gone?
   Remove the newline at the end of the string in “hello.c” and look again at the resulting “hello.s”. Explain.

3. Run “gcc -c hello.s”, producing the file “hello.o”.
   Use “objdump -rd hello.o” to inspect this relocatable object file.
   Explain why this piece of machine code is (not yet) executable.

4. Run “gcc -o hello hello.o”, producing the file “hello”.
   Use “ltrace ./hello” to see what library calls your program makes.
   Run “ltrace -S” to also see the system calls.
   What is the actual exit code of your program? Can you explain this?

Remark: You can generate all the intermediate files at once by using the “-save-temps” option of “gcc”

➢ **Stripped and non-stripped binaries (Bonus)**

1. Breifly explain what is stripped and non-stripped baniries and write at least two pros and cons.
2. Compile the below code in two versions e.g. stripped and non-stripped and prove your above explanation.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Stripped and non-stripped test\n");
  return 0;
}
```

3. Can you find entry point in a stripped binary file, if so, use a debug tool and find the entry point of the previously compiled stripped file. If no, why?

6. The file descriptor (Bonus)

Now we will look at the different file descriptors and learn what happens when a process opens a file.

1. Explain why we need file descriptors?
2. What is a file descriptor table in a process and what items it holds?
3. list file descriptors of your current bash session.

4. Where are the locations of file descriptor (linux)?
5. Write a code that creates a file and print the default file descriptors index and pointers in the file descriptor table of the process.

➢ Redirecting file descriptors.

Briefly explain and provide at least two examples of redirecting file descriptors.
i.e run a command (E.g. find / -name '*something*') and redirect the file descriptor.