Lab3: Reading Material

Lab 3 is dedicated to basic programming in Assembly language, getting familiar with basic instruction, addressing, direct system calls, and C function calling from Assembly code.

Goals

1. Basic proficiency in assembly language programming.
2. Understanding the interface to the operating system services.
3. Basic understanding of code manipulation, through implementation of executable file patches.

Motivation

Aside from the clear advantages implicit in the above goals, the resulting executable files are very small and simple, and require little space to execute. This has clear performance advantages, when such are desired. Additionally, the object and executable files have a very simple structure, and are easier to understand - something that will be very useful in later labs.

C functions

In this lab you are required to process the arguments passed to the main function and print them. Read the manual about the main function man main.

Assembly and NASM

In this course we use NASM, the Netwide Assembler: an assembler targetting the Intel x86 series of processors. Read the NASM manual. Review the lecture slides: Assembly Language Primer, and read about each instruction shown in the lecture in the NASM manual for better understanding. Make sure to understand the basic arithmetic commands, flow commands, effective addressing, variables "declaration", global, extern etc.
Read more about NASM registers registers.

Methodology

1. You will be provided with basic startup code in assembly language, (file start.s). The file contains just the "glue" code that starts up main( ) by getting argc and argv and passing them to your main( ). It also contains an example of an assembly language "glue" to access Linux system services using the C "CDECL" calling convention.
2. You should LOOK at the file first, and understand the code!
3. Understand how to generate stand-alone code that uses only your assembly language code and the "glue" from start.s, with no libraries whatsoever, using Linux system services: INT 0x80.
4. All that needs be done is compiling your code, assembling the glue code (start.s) and linking them together to produce an executable. (Note: in future labs, you will learn how to write a simplified version of ld - the linker - to create your executable from compiled object modules).

5. Exercise with some system calls, mainly file system related.

Additional Reading Material and references: system calls

Make sure you read and understand the following **before** attending the lab.

1. Read the "man 2" page for the following system calls: open, read, write, close (or re-read them if you have read them before, just to make sure you understand them). Although these man pages are for standard-library interfaces to the actual system calls, the functionality of these interfaces is the same as the actual system call, except for the issue of returned value and errors, and other minor issues. It might benefit you to note that the system calls open( ), close( ), read( ), write( ) are also quite similar to some of the functions that you have already used in previous labs: fopen( ), fclose( ), fread( ), fwrite( ), respectively. The difference is in the file descriptor (now int (index into an open file table held and managed by the system) rather than FILE* (which is a structure which also contains the above mentioned int index)), as well as some of the argument values, but the functionality is essentially the same. The version with the "f" (e.g. fopen, fread, fwrite) that you have used before, provides an additional level of abstraction/wrapping.
2. System calls: the basic mechanism for accessing OS services
   Note: Faulty Documentation! Since the documentation of the actual Linux 32 bit system calls is not so easy to read, we provide the documentation based on the Linux man (2) of the respective wrapper functions. They are functionally the same as the respective system call except for the **return value**. The actual return value when an error occurs is a negative value that is **not** necessarilty -1 as stated in the documentation and the presentation. That is because the raw system calls have no access to the errno variable. Instead, they return the negative value of the error code. Therefore, you need to check whether the returned value is less than 0 rather than equal to -1.
3. The exit system call.
4. Notice that the code in the man page is different from what we expect (they use a different method for system calls - syscall, we expect you to use the INT 0x80 system call directly. Again, make sure you do not include any external files. As always, If you use any code you didn't write yourself, make sure you understand everything 100%.
5. For getdetns see directory: "Some Linux system calls: presentations",getdends

Additional Reading Material and references: hexedit

We will be working with "binary" files starting with this lab. Make sure you have hexedit installed and read about it sufficiently to be able to use it to examine a file and exit.