Lab 9 (ASM)- Executable file virus

This lab is for students who take the full architecture and SPlab course, i.e. are familiar with 80X86 assembly language. If you are taking only SPlab, you should be doing the **other** lab 9 (in C).

Introduction

The goal of this lab is to understand the structure of executable ELF files and how to manipulate them. To make things interesting, this will be done by hands-on learning and implementation of a rudimentary computer virus.

In this lab you will implement a simple virus that infects some ELF executable files. Your virus will be able infect static executable files with **two** program header, which are executable files that do not use dynamic library code. In particular, your virus will copy its own code to end of the target ELF file, and modify the ELF file program headers to make the injected code load, and the ELF header so that the virus code will run when the file is executed. For simplicity, your virus will only infect one file, with a constant file name ELFexec.

In order to make this work, your code will have to be in position-independent code, written in assembly language, and use only direct system calls.

Then, using code from previous labs, you will implement a program that detects your virus, and "disinfect" the infected file(s).

Task 0

This task comprises two subtasks: the first is an introduction to program headers in ELF executables, while the second is preparation for the virus by re-implementing a subtask from lab 4. You should do these tasks and understand all the related material before the lab.

If for some reason you are **unable** to do task 0 seriously (especially task 0b) before the lab, we suggest that you state this up-front to your TA, and you will get a reduced set of tasks (at a reduced grade, of course). That is because arriving at the lab without doing task 0 practically guarantees that you will be unable to proceed, in this lab more than in any of the previous labs.

Task 0a

Write a program, which receives a single command-line argument: the file name of a 32bit ELF-format executable file. Your program should implement the *readelf -l* functionality.

Your task is check if this is an ELF executable file, and if true, to loop over the program headers in the file. For each program header, print all the information residing therein, using the corresponding `Elf32_Phdr` structure.

The output should look roughly similar to *readelf -l*, but do not waste too much time on the quality of the output:

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x04048034 0x04048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x04048134 0x04048134 0x00013 0x00013 R   0x1
  LOAD           0x000000 0x04048000 0x04048000 0x008a4 0x008a4 R E 0x1000
  LOAD           0x0008a4 0x040498a4 0x040498a4 0x0011c 0x00120 RW  0x1000
  DYNAMIC        0x0008b0 0x040498b0 0x040498b0 0x000c8 0x000c8 RW  0x4
  NOTE           0x000148 0x04048148 0x04048148 0x00020 0x00020 R   0x4
```

Since the purpose of this task is merely to familiarize yourself with program headers, you should implement this program in C, using standard library as usual. Note that it is **not** advisable to do this task without the C standard library, even if it is possible. Also, you may print just a numerical value as the type and flags, instead of the labels shown in the above example, in order to simplify your code. However, regardless of your implementation, make sure you really understand what the fields in the program header mean.

Task 0b

This lab relies on code you wrote near the end of lab 4 in assembly language, where you attempted to implement a virus. The code you wrote opened an executable file, and added the code to the end of that executable file. Now you need to repeat that lab 4 task, but this time you must implement your code using strictly position independent code (PIC). That is, you must use only system calls to access the file, and make sure all references are either relative addresses in the same (.text) section, or are on the stack, or you must fix their references at runtime using the current IP (e.g. using the get_my_loc() function from the class on PIC).

Specifically, your code should:

- Print a message to stdout, something like "This is a virus".
- Open an ELF file with a given constant name "ELFexec". The open mode should be RDWR, rather than APPEND.
- Check that the open succeeded, and that this is an ELF file using its magic number.
- Add the code of your virus at the end of the ELF file, and close the file. You should use lseek() to get to the end of the file, as it returns a useful number: the number of bytes in the file.
- Exit: with code 0 if no error in any of the above, otherwise with some non-zero exit code.

In order to make your code shorter, you should use this file as a skeleton. We provide macros so that you can activate all needed system calls easily (using one line each), and also sets up the structure of your virus, which should all be in section .text. The skeleton also provides definitions for most constants related to the ELF file format needed for this lab.

Since you will be unable to use .data or .bss sections, any data you need to store which is not read-only should be placed on the stack, as "local" variables. That is, set up EBP and ESP appropriately (as done in the skeleton code we provided) and refer to your data using [EBP-x], with appropriate values of x. **All your code henceforth must be written for NASM, compiled into 32-bit ELF executables, and linked using ld without any libraries.**

Note frequently observed error: you may have used a **label** in order to denote a "buffer address" for system calls in lab 4. Now that you are placing the "buffer" on the stack (say at EBP-BUF_OFFSET) you **cannot** pass the argument "[EBP-BUF_OFFSET]" to the system call macro! That is because this gives the system call (usually) an uninitialized pointer, the **value** stored at address EBP-BUF_OFFSET (on the stack)! Instead you need to provide a register that has a value equal to EBP-BUF_OFFSET. Can be done by subtraction or by LEA.

Task 1: infection, and making the virus run

Download this ELF executable file and run it to see what it does. This is the type of ELF executable which your virus should infect. Keep an additional "clean" copy of this file, as you will be modifying it multiple times using your virus as you develop it. Examine this file using readelf, especially noting the program headers and entry point, which you will be modifying during the lab tasks.

In this first task, you will be extending the preliminary virus code from task 0b to make the virus code run when the infected file ELFexec is executed. Note that we also need to make sure that the virus code is loaded into memory when ELFexec is executed, but for now we will assume that the entire file is "magically" loaded into memory (and not just the part indicated by the program header). Indeed, for this ELFexec it indeed will happen, **why?**

First, check that your code from task 0b works correctly, using hexedit on ELFexec before and after your code runs. Then add code to that of task 0b, as follows:

- Code to copy the ELF header into memory (on the stack). Do **not forget** to make sure that sufficient local storage is allocated on the stack before doing so.
- Code to modify the copy of the ELF header of ELFexec that resides in memory, updating the entry point to the first instruction of your virus. Note that this must be the place it resides in memory when the **infected ELF file** is executed, and not the first instruction in the currently executing virus. In order to

find the address to which this file loads to in main memory, you need data from the **first** program header. At this point you may find its value using readelf and plug it into your code as a constant value.
- Code that writes back the ELF header to the beginning of the ELFexec file.

After compiling and running your new virus, as usual, check using readelf that the ELFexec was modified as you expected. Now, try to run the infected ELFexec. What can you observe? Check with the TA for correctness at this stage. You may write a program which prints the entry point and use this script to check your virus.

Task 2: getting back to the infected program after virus completes

Now, in the infected file, the virus code is run, but the original program which you attacked does not. The goal is now to make the original program run after the virus executes. In order to do that, the virus should, instead of exiting, jump to the address indicated by the **previous** entry point. In order to do that, when infecting the ELFexec file, your virus should save the previous entry point somewhere before modifying it. The simplest scheme is to write it at the very end of the file after the virus code, replacing the value at that location in the skeleton code.

Then, assuming we are running a copy of the virus, memory at an appropriate location at runtime will contain the previous entry point, so you can jump to that address when the virus terminates. A minor complication is that the **original** version of the virus, the one you are compiling and linking (not in an infected file), does **not** have a "previous entry point", so you in this case you simply need to **exit**. This is solved if that value in the original code just points to a location in the code where the virus exits (as in the skeleton code).

Now your virus appears to be fully functional. Starting from a fresh ELFexec file, infect it using your virus. Now run ELFexec. It should run the virus code and then the original code. Now, we want to check if the instance of the virus in the infected file also infects files.

- Can it infect itself? Why, or why not?
- Copy the infected ELFexec to some other file name, and run this new version. What effect are you getting, and why? (check using hexedit and readelf.)
- Do this **again** (copy ELFexec to a new file name, and run this new file) keeping the infected version of ELFexec. What effect are you getting, and why?

Finally, infect the ELFexec file multiple times. After ELFexec becomes infected with too many copies of the virus, it crashes when you try to run it.

Why is that, and how many copies of the virus does it take to make it crash?

Find the answer to the last question **theoretically** (compute it), and empirically (keep re-infecting until the infected file crashes), and compare these results.

Task 3: making sure the virus code is loaded

In tasks 1 and 2 the virus seemed to work correctly. But in fact the virus code loaded only "accidentally", due to the fact that complete blocks of 4K bytes are mapped into memory, and the total size of the executable file including the virus is less than 4K bytes. Try your virus with this slightly longer ELFexecLong file. What happens when you try to run original and the infected file? In this case your virus code was **not** loaded correctly into memory, and we need to modify program headers to make sure that it loads.

Since this ELF executable has **two program headers, we have to modify the second program header** in-place so as to make sure the loader loads the entire file, including the virus code, into memory. This will be done by modifying the "file size" and "mem size" fields in the **second** program header accordingly.

In order to do that, you need to do the following steps:

- Add code to copy the second program header of ELFexec into memory (on the stack).
- If in task 1 and 2 you used a constant value as the address at which the program loads to in main memory, fix this to use the data from the program header (that is, unless you already did it in tasks 1 and 2).

- Add code to modify the **second** program header in memory. You should set the "file size" and "mem size" fields to: file size + virus code size - program header offset. (Note that "program header offset" means the value of the offset field in the relavant (2nd) program header, **not** the offset of the program header table.)
- Add code to write back the modified program header into its original place in the ELFexec file.
- Run your new virus, and check using readelf that it updated the program headers correctly. Compare to the new file size.

After all this is done, run both ELFexec and ELFexecLong after infecting each of them using your virus. Note that every try of running your virus you need to start with a fresh version of these executable files, as these attempts will corrupt the executable files. Once your virus is completed correctly, try to infect an execytable file multiple times, as before, to check that it still works as expected.

Task 4: updating your anti-virus to detect your virus

Now it is time to represent the other side, the honest user or system administrator. You need to write a program that will detect your virus in an ELF file, and find a way to deactivate it. Fortunately, you already wrote a virus detecting program in lab 3, so all you need to do is use it. In order to use it to detect your virus, you should update the virus signatures file from lab 3 (or create a new one, if you wish) that will have a signature of your virus. You can do that using hexedit, or even better, use your hexeditplus from lab 8 in order to copy your virus code into your signature file.

Now use your virus detector using a fresh version of ELFexec, as well as infected version, to check for viruses. Try also a multiply-infected version of ELFexec (resulting from more than one execution of the virus code).

Finally using hexedit, deactivate a singly infected version of ELFexec. The simplest way to do it is to change the entry point back to the original entry point. Do that and check whether ELFexec is restored. Note that in this way, the virus code is still inside the ELFexec, but it is deactivated.

Deliverables

This lab is rather challenging, even though there is not so much code to write. Therefore it **may be done in pairs** (or solo if you prefer). You should complete everything from task 1 to task 3 during lab hours. Task 4 may be done in a completion lab if you run out of time.

If you run into extreme difficulty in the lab, you may (after telling the TA of your problems, and coordinating this with the TA) do Task 4 before Tasks 2 and 3 for partial credit.