--- ---

REVERSE ENGINEERING: ASSIGNMENT 2

Tentative due date: Friday, 1st June 2018 3pm

NOTE:

- When the disassemblies are asked, focus only on disassemblies of **main**, and any other function defined by us. Do not bother all other stuff in the disassembly.
- Questions 1 - 8 will help you in understanding basic analysis of linux binaries.
- Note about Questions 9 - 12 :
    - These questions will introduce you to vulnerable functions in the C library, what segfaults are, what a stack overflow is and much more. Basically, it is an introduction to binary exploitation.
    - These questions involve entering random input strings, long input strings. Do not waste time in typing long input strings. Open a python interpreter in another terminal and generate the string of desired length. For these questions, only length of input matters, not what it is.

Questions:

---

1. Refer **code1** executable file for this question.

   a. What architecture does this executable run on? How did you find it out?

   b. What is a magic number of a file? What is the magic number for this executable?

   c. Open any **pdf** file in a text editor. It is a binary file. An executable file is also a binary file. So, can you run a pdf file as an executable? Try running a pdf file and report what you get.

   d. GIve reason on why whether a pdf file can be executed or not.

---

2. Refer **code2_1** ,**code2_2** and **code2_3** executable files for this question.

   a. Run all the 3 files. Did you find any difference in the output?

   b. What is the difference among the 3 files? Use the **file** command to inspect these files and Report the differences.

   c. Why do you think, there is such a big size difference between **code2_1** and **code2_2** ?

   d. What is the meaning of a **stripped** executable? Did you encounter the word **stripped** during analysis? Check out the differences between disassemblies of **code2_1** and **code2_3** and report them.

---

3. Refer **code3.c** sourcefile and **code3** executable for this question.

   a. Have a look at code3.c and understand what it does. Run the **code3**.

   b. Write assembly code for code3.c in a **rough** manner(Dont need the exact code, but something like pseudocode). (On a serious note, do not get the disassembly and copy it , because you will miss out on the essence of the question)

c. Now, Get the disassembly of **main** function. Compare it with your assembly code. What are the differences?

d. Has the compiler done something to your code??If yes, What has it done?

---

4. Refer **code4** executable for this question.

   a. What does this executable do?

   b. Check out the **/proc/PID** entry for this process. Note down the different Address spaces given to this process.(you have to find out where you can get this).

   c. Look at the address spaces . Can you give names(names like text segment, data segment etc.,) for each address space depending the addresses ?

   d. Look at the permissions of all the address spaces - (**r-wp**, etc.,). Do you find a **-wxp** permission in any of those permissions? What do you think could be the reason for your answer?

   e. Run code4 multiple times(probably 3-4 times). For each time make note of address spaces(Take a screenshot). Focus on **stack** address range. Is it changing **everytime** you run the program? Write down the stack address range for every time you run.

---

5. Refer **code5** for this question.

   a. Analyzing the strings present in an executable file will help you get to know what the executable is doing. Use the **strings** tool to get the strings of code5. Go through them and see if there are any meaningful strings.

   b. The task is to find a **flag**. One string among those strings in the executable when **xored** with one of the digits from 1 to 9 will give a meaningful string. That meaningful string is the flag. Find the flag.

   NOTE: Manual xoring of each string will be difficult. Try writing a C program or a python script to get the job done. The **flag** is of the form **_____ {_____}** , where there is some text instead of underscores.

Make a list of tools and techniques used to answer the above 5 questions. Making a list will help you which technique/tool you should use to start the analysis, because always the same approach will not work everytime.

The above 5 questions, were based on **static analysis**, meaning, you don't have to run the executables to know what they are doing. Just by looking at their headers, looking at their disassembly , strings, etc., you can figure it out.

The next 7 questions will be based on **dynamic analysis** , which means you have to run the executable(normally or using a debugger) to answer the questions. You might have to do some static analysis also before getting into dynamic analysis as you might get some early lead.

These are the 2 techniques used to analyze most of the malware. Most of the malware you take, are executables . You perform both static and dynamic analysis on the malware samples, and find out how it is affecting/harming the computer.

Now, to the questions.

---

6. Refer **code6** for this question.

a. What is the output when you run code6? Write the simplest C program which can give the same output.

b. Perform static analysis on code6 and report whether the code6 is same as the program you wrote.

c. Run the code6 using a debugger. Report the functions present in code6. What instructions constitute construction and destruction of the stack frame?

d. What is a **nop** instruction? What does it's name suggest about it's functionality? Do you find it in any of the functions in code6?

e. This is optional. Write a C program which will generate an executable similar to code6.

---

7. Refer **code7** for this question.

   a. Write how arguments are passed into a function at the assembly level.

   b. You have to find a flag. How should you run code7 so that it prints the flag? (Look at the disassembly and find out).

   c. Is there any quick hack to get the flag? Report atleast 2 methods.

I hope the above questions have brushed your basics. The next 5 questions will be on applying what we have discussed so far.

---

8. Refer **code8.c** for this question.

   - In code8.c, the recursive and iterative versions of finding the factorial of a number is defined. Go through the sourcefile, compile it, run it and understand what exactly the program does.

   a. For n = 10,

   - Find the number of assembly instructions executed and total stack space used when the iterative version is chosen.(Consider only the factorial_iter instructions. Ignore the instructions in main, and consider call ret , push and pop as 2 instructions each)
   - Find the number of assembly instructions executed and total stack space used when the recursive version is chosen. (Consider only the factorial_rec instructions. Ignore the instructions in main, and consider call ret , push and pop as 2 instructions each)
   - You can count it by hand, but use the debugger to get an accurate answer.

   b. Find general formulas for (i) total number of assembly instructions (ii) total stack space used , both as functions of n . Then use the formula to calculate the number of instructions for n = 7, n = 4, n = 3, n = 2, n = 1.

   c. Which do you think is efficient, recursive or iterative version of factorial? (Based on the total number of instructions and stack space usage)

---

9. Refer to **code9.c** for this question.

   a. Go through the code9.c and Report what it does.

   b. Compile code9.c in the following manner

```
$ gcc code9.c -o code9 -fno-stack-protector
```

We are removing a security feature and compiling it. Removing it will make our experimenting easy. .

c. Explore(google it and find out) the following command :

```
gdb-peda$ x/100xw $rsp
```

Before running the executable, execute this command. We are removing another security feature . Run the instruction as **root**.

```
# echo "0" > /proc/sys/kernel/randomize_va_space
```

d.

- Perform static analysis and write down the **return address** of the print_buffer function.
- Then open the executable using a debugger. Make note of the **rbp** value pushed when print_buffer is executed. Run the program atleast 2 more times, and make note of **return address** and **rbp** value pushed onto the stack.
- Is the return address noted from static and dynamic analysis same?
- Run the executable normally with random inputs. Make note of the **size of the buffer** the string you entered is getting copied into. Now, enter inputs and run the program. Give an input which can give a **segmentation fault**. Report that input.
- This is optional. Run the program with debugger, along with the input that generates a segfault. Give the stack images before and after copying of string(before and after execution of **strcpy() function**). Can you find the **reason** behind the segmentation fault?

  Once you finish everything, enable back the security feature we had disabled(It is important), as root.

  ```
  # echo "2" > /proc/sys/kernel/randomize_va_space
  ```

---

10. Refer **code10.c** for this question.

    - Go through the sourcefile and know what it does. Compile code10.c in the following manner. We are removing a security feature added during compilation.

      ```
      $ gcc code10.c -o  code10 -fno-stack-protector
      ```

    a. While compilation, do you get any warnings? If yes, what is the the warning?

    b. Run the executable using a debugger. Make note of the **rbp value** pushed at the beginning of the main function. Also make note of the **return address** .

    c. Similar to previous question, run the program with 5 random inputs. Make note of **size of the buffer** you are entering your input. Now again start giving inputs. Give an input which can generate a **segmentation fault(segfault)** and stop the program.

    d. Why do you think the program segfaulted? Does it have to do anything with size of buffer or size of the input you entered?

    e. Give the length of smallest input which can segfault the program. Find out the reason for this using the debugger. Go to qn 11, subquestion c, part 1, to get a clue.

---

11. Referring to questions 9 and 10.

    a. Did the programs code9 and code10 segfault for some input?

b. In both the cases, if they segfaulted, why do you think they did for certain inputs? . Were the usage of some functions a reason for this? If yes, name the functions, and a small reason to your answer.

c. Refer only question 10 / code10.c / code10 for this part(This is optional)

- What is the return address of the main function?(check the top of stack just before **ret** is executed)
- From the **e** part of question 10, is the string of that length doing something to the above return address? What happens to the return address if you give a string with length more than that smallest length?

---

12. Refer **code12.c** for this question.

    Before running it, execute this command. We are removing another security feature as **root**. Do sudo su root, and then this command.

    ```
    # echo "0" > /proc/sys/kernel/randomize_va_space
    ```

    - Go through the sourcefile and understand what the program does.

    a.Compile the source code like this.

    ```
    $ gcc code12.c -o code12_1 -fno-stack-protector
    ```

    Start running it with random inputs. Check if you can get a segfault. If yes, Report the length of the smallest input string which can segfault the program.

    b. Now compile code12.c normally like this.

    ```
    $ gcc code12.c -o code12_2
    ```

    Do the same. Check if you can get a segfault. If yes, Report the length of the smallest input string which can segfault the program.

    c. When the program segfaults, generally, you will get an output something like this:

    ```
    Segmentation fault (core dumped)
    ```

    But, is there any difference between the error outputs in question a and question b when the program segfaults? Report what that difference is.

    d. Now, you have 2 executables of the same C sourcefile. One compiled with **-fno-stack-protector** and one without it. What does this flag say? Can you relate the meaning of the flag to the segfault output in question b ?

    e. Take a look at the disassemblies of **func1** function of both **code12_1** and **code12_2** executables. Do you find any difference in them? If yes, Report the exact differences between the 2 **func1** functions.

    f. Can you now find a link between the usage of segment registers, **-fno-stack-protector**, output in subquestion b ? Give a guess on why segment register is used for here.

    g. (This is optional)Run the code12_2 using a debugger. What is stored at the location **fs:28** ? Will it change everytime you run the program? If yes, Write down outputs of 5 attempts. What is the value of **fs**?

THE END