CSSE 132 – Introduction to Systems Programming
Rose-Hulman Institute of Technology

Lab 4 – Question Sheet

For each of these questions, clearly explain your answers and write out any commands the questions request.

Name (Print):____________________ RHIT Username:____________________

Part 1: Hacking via Pointer

Read the Lab 4 instruction Part 1 on the website first before start doing this part.

1. (**Breakpoint: Line 24)** Right before calling `add` function in `main` function, you can check the Assembly code to find the return address of `add function`, i.e., the address of the instruction right after the `bl` instruction. What is that return address?

   (HINT: it is a `str` *instruction.)*

2. (**Breakpoint: Line 12)** At the beginning of the `add` function,

   (a) What is the value of register `lr`? Verify if it is the same as the address you put in the previous question.

   (b) Examine the stack to find out the location storing the return address mentioned above. In the GDB `stack` window, due to the limitation of the window size, you cannot see the complete stack. Instead, you can use the following command to examine the stack.

   ```
   dereference $sp
   ```

   Run this command, write the memory address where the return address is stored. Write the relative address with regard to `sp`, e.g., `sp + N` (`N` can be a hexadecimal number)?

3. (**Breakpoint: Line 16**) Right after declaring and initializing `int* xp`,

   (a) What is the address of the variable `result` (use `dereference $sp` to check the stack again)?
   (HINT: It is where 3 is stored.)

   Write the address in both the absolute address and the relative address with regard to `sp`.

   (b) Check the value of the variable `xp` (by typing `print xp`). To see if it is the same as the address you wrote down above? If not, redo the previous step.

4. Given the information above, we first need to adjust `xp` to point to the place where the return address is stored.
   To do that, we need change the line below in C code from

   ```
   int* xp = &result;
   ```

   to:

   ```
   int* xp = &result + N;
   ```

   Based on the answers you put down in the previous questions, what should the `N` be?
   (HINT: `&result` *is actually a pointer* `int*`*. Incrementing this pointer by 1 will increase the absolute memory address by 4.)*

5. Change your code with `N` calculated above and run code in GDB again. As the final step to finish the hack, we need to modify the data that `xp` points to so we can overwrite the return address. To find out the address of `surprise`, in GDB (no matter which line you are at now), you can use `print <function_name>` to print the address of any function. Run this command and write down the address of the `surprise` function.

6. Finally, we can use `*xp = ______;` to modify the data where `xp` points to. In our case, we want to put the address of `surprise` in that place. Change the line below in C file from

   ```
   *xp = 0;
   ```

   to:

   ```
   *xp = <address of surprise>;
   ```

   (HINT: No need to convert the address to decimal. Use `0x` *to directly write the address in hexadecimal.)*

   After you make the change, compile and run the code again using `make run`. Are you able to see a message `Surprise!!!!!`? If not, redo the questions above.

   If you successfully finish the hack, ask your instructor to verify your answers and sign off this part.

   Verified:______________________________ Date/Time__________________

Part 2: Hacking via Buffer Overflow

Read the Lab 4 instruction Part 2 on the website first before start doing this part.

7. Let's first find out where is that `buffer` array is stored. This is how: Run this command

   ```
   ./run_part2.sh ABCD -g
   ```

   By doing so we are using `GDB` to run the program `part2` with the input `ABCD`. In `GDB`, run to the place inside of `doIt` where just finishes `copyIntoBuffer` (Hint: Breakpoint Line 38). Now on the stack of `doIt`, you should see where that four bytes `ABCD` (`0x44434241` in hexadecimal) are stored. Then, what is the starting address of that `buffer`?

   Write the relative address with regard to `sp`, e.g., `sp + N` (`N` can be a hexadecimal number)?

8. On the stack of `doIt`, what is the address where the return address is stored?

   (Review Part 1 Problem 1 and Problem 2 to solve this problem. Basically, you need to find out the return address of `doIt` *and then localte it on the stack)*

   Write the relative address with regard to `sp`, e.g., `sp + N` (`N` can be a hexadecimal number)?

9. Assume the command-line input bytes we will give can be divided into two parts represented as `AAA...ABBBB`, where the `AAA...A` are some random bytes used to fill memory spaces, and the `BBBB` are the four-byte address of `holyGrail` that we will use to replace the return address of `doIt` on the stack.

   Given your answers for Problem 7 and Problem 8, how many bytes in that `AAA..A` part you need to fill the spaces such that the `BBBB` part can be placed right at where the return address of `doIt` is stored?

10. To verify this, run the program with the number of As you calculated and followed by the `BBBB` part as `0x41424344` (a temporary value). You can run the program by typing:

    ```
    ./run_part2.sh $'AAAA\x41\x42\x43\x44' -g
    ```

    NOTE: We are using many character `A` here as the random filling bytes. You need to replace the `AAAA` with as many `A`s as you calculated in Problem 9. You literally need to type many `A`s manually, like `AAAAAAAA`.

    After launching the program with your modified command-line input (with the correct amount of `A`s), run the code to the end of `doIt`, check the stack and answer the **two** questions below:

    (a) Are the four bytes `0x41424344` placed at the exactly where the return address was at (check your previous answer to recall where the return address was stored)?

    If not, redo the questions above until you see the four bytes `0x41424344` *are stored at the place where the return address was stored.*

    (b) If your answer is *yes* to the previous question, then is the four bytes `0x41424344` stored in the same order as you typed in? What does this tell you about how you should type the function address as the `BBBB` part later?

11. What is the function address of `holyGrail`? (Review Problem 5 if you forget how to do this)

12. Given the answers above, what is the final command-line input you figured to hack the program? Write down the complete input including the $".

13. Run the program with the input you wrote above without GDB, i.e.,

    ```
    ./run_part2.sh <your input>
    ```

    Are you able to see a message `YAY! You DID IT!`? If not, redo the questions above.

14. Change in `copyIntoBuffer` to prevent this kind of hacking—buffer overflow? (Edit the function below) (*HINT: Please avoid using function* `sizeof`. *You can add more function input arguments as you see fit.*)

```c
void copyIntoBuffer(char* src, char* dest) {

    int i=0;

    while(src[i] != '\0') {

        dest[i] = src[i];

        i++;
    }
}
```

If you successfully finish the hack and also manage to fix the vulnerability, ask your instructor to verify your answers and sign off before turning in this sheet.

Verified:______________________________ Date/Time________________