Homework 2

Due Date: **Friday, 10/10, 11:59pm**

Instructions

1. Unless otherwise specified, any assignment involving programming may be completed with the programming language of your choice. If asked, you should be able to explain the details of your source code (e.g. program design and implementation decisions).
2. You are bound by the Stevens Honor System. All external sources must be properly cited, including the use of LLM-based tools (e.g. ChatGPT). Your submission acknowledges that you have abided by this policy.
3. Solutions are accepted only via Canvas, where all relevant files should be submitted as a **single `.zip` archive that expands into a directory**. This directory should include your ***typed* answers as a `.pdf` file** and **source code of any programming used in your solutions** with the following structure:

```
<lastname>_<firstname>.hw<#>/
├── solutions.pdf
└── src/
    └── <source code files>
```

   Your `src` folder should document all necessary steps for reproduction. If we are unable to reproduce your results, you may lose credit.

   There is a script to zip your homework files for you based on prompts.

   There is a python validation script you should use to confirm it is correct.

Problem 1 - (*25 pts*) Robust Refactor

Complete *Exercise 16* from the *Robust Programming* reading. You may base your code on the `qlib` implementation in the `robust.zip` files, but that is not guaranteed to be free of issues.

- **Report what changes you made to the library and why, noting any new sanity and consistency checks you needed to make in particular.**
- *Include your header and source files in your submission, with the names `dqlib.h` and `dqlib.c`, respectively.*
- **Bonus (2pts):** What is wrong with the `qlib` implementation as provided?

Problem 2 - (*25 pts*) Bypassing Memory Safety Features

Consider the following code snippet:

```c
1   void foo(void) {
2     char foobuf[256];
3     fgets(foobuf, 257, stdin);
4   }
5
6   void bar(void) {
7     foo();
8   }
9
10  void main(void) {
11    char * buf = (char *) malloc(128);
12    fgets(buf, 128, stdin);
13    printf("%p", buf);
14    bar();
15  }
```

Assume the following:

- **ASLR is enabled**.
- All other memory safety features disabled.
- Assume ASLR will always randomize the value of the saved base pointer of `foo` to have a LSB in the inclusive range `0x20 - 0xf8`.
- You can represent the output of `printf` on line 13 with a variable `ADDRESS`. This is an 8-byte, little-endian byte string.
- You can represent shellcode with a variable `SHELLCODE`, which has a maximum length of 120 bytes.
- *Note: SFP = saved frame pointer = saved base pointer (rbp)*

Answer the following questions:

1. **(6pts)** Complete the following stack diagram assuming the program is paused at line 3 (before `fgets` is called).

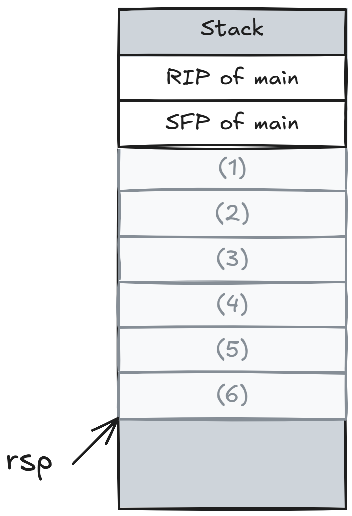

Figure 1: Stack state at line 3

2. **(8pts)** Provide the byte string inputs to `fgets` on line 12 and `fgets` on line 3 that will lead to shellcode execution. You can craft these inputs using Python syntax. For example:

```py
>>> b'\x31' * 8 + b'\x0b\xad\xc0\xde'
b'11111111\x0b\xad\xc0\xde'
>>> ADDR = b'\x00\x00\x7f\xff\xff\xff\x80\x80'
>>> ADDR + b'\xc0\xde'
b'\x00\x00\x7f\xff\xff\xff\x80\x80\x00\xc0\xde'
```

3. **(11pts)** Explain why this exploit is only guaranteed to work if the saved base pointer of `foo` has a LSB in the range `0x20 - 0xf8`.

Problem 3 - (*25 pts*) No Overflow?

Consider the following CTF challenge:

```c
1  void challenge(void) {
2    int8_t size;
3    char flag[16];
4    char buf[128];
5
6    // implementation not shown
7    load_flag(flag);
8
9    memset(buf, 0, 128);
10
11   fread(&size, 1, 1, stdin);
12
13   if (size >= 128) {
14     return;
15   }
16
17   fread(buf, size, 1, stdin);
18   printf("%s", buf);
19 }
```

The `load_flag` function loads the flag data from an access-controlled file on the system.

Answer the following questions:

1. **(5pts)** Complete the following stack diagram assuming the program is paused at line 8.

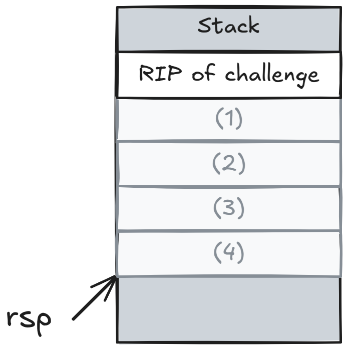

Figure 2: Stack state at line 8

2. **(5pts)** Provide the byte string inputs to `fread` on line 11 and `fread` on line 17 that will give you the flag.

3. **(5pts)** Explain how your solution to the previous part works.
4. **(5pts)** Provide the byte string inputs to `fread` on line 11 and `fread` on line 17 that would allow you to execute the largest possible shellcode payload. Assume all memory safety features are disabled and the compiler allocates 8 bytes of space for `size`.
   - *Use a variable named `SHELLCODE` that contains the shellcode payload as a byte string.*
   - *Use a variable named `BUF_ADDRESS` that contains the address of `buf` as a byte string (presumably reverse-engineered with `gdb`).*
   - *You must specify the length of `SHELLCODE`.*
5. **(5pts)** Explain how your solution to the previous part works.

Problem 4 - (*25 pts*) Python has Problems Too

Just because we've been mostly looking at problems in C doesn't mean other languages don't have their own problems.

A well-known dangerous implement in Python (aside from the use of the `eval` function) is the `pickle` library. The `pickle` library is Python's native module for *serializing* and *deserializing* its objects, which can be quite useful for caching purposes.

Per the documentation, however, unpickling an object may also involve executing arbitrary code. You'll be crafting a pickle-based attack in this problem and creating a short vulnerability report and description of your exploit.

1. **(15pts)** Complete the *Pickle Rick* challenge in the *Pickle* module (originally part of the *Hanto Dojo*).
   - **Hint 1**: Find out how pickle exploits work (Google it), then read the `pickle-rick.py` source code to find out how the pickled data is given to the program.
   - **Hint 2**: Your goal is still to read `/flag`.
2. **(10pts)** Describe the vulnerability in the `pickle-rick.py` script and the steps you took to exploit it.
   - *The "Vulnerability Description" section here and the "Real World Demonstration" section here are good examples of what we'd be looking for: point out the affected code, what input the attacker controls, and describe the steps taken to reproduce the exploit.*