Some questions in the written assignment come from the textbook: `Information Security Principles and Practice`. This assignment counts 10% of your final grade.

Written Assignment

1. (12pt) Reverse engineering.

   (a) (3pt) What is the typical threat model for adversarial reverse engineering.

   (b) (3pt) Explain what is linear disassembling.

   (c) (3pt) Explain what is recursive disassembling, and what is the advantage of recursive disassembling comparing to linear disassembling.

   (d) (3pt) Briefly clarify the procedure of control-flow recovery in modern C/C++ decompilers.

2. (21pt) The C function `strcat` appends a copy of the **source** string to the **destination** string. The terminating null character in **destination** is overwritten by the first character of **source**, and a null-character is included at the end of the new string formed by the concatenation of both in **destination**. **destination** is the return value. The definition of the interface of `strcat` is:

   char * `strcat` (char * destination, const char * source);

   (a) (3pt) Is `strcat` safe? Why?

   (b) (10pt) `strncat` is a function with similar functionality of `strcat`. `strncat` appends the first **num** characters of **source** to **destination**, plus a terminating null-character. **destination** is the return value. Please give an implementation of `strncat`. Its interface is

   char * `strncat` (char * destination, const char * source, size_t num);

   (c) (3pt) What problem is solved by `strncat`?

   (d) (5pt) Is `strncat` safe? Please explain your answer.

3. (20pt) In addition to stack-based buffer overflow attacks, heap overflows can also be exploited. Consider the following C code, which illustrates a heap overflow.[1]

```
int main()
{
  int diff, size = 8;
  char *buf1, *buf2;
```

[1]Hint: try this online compiler to compile and run the code: https://www.onlinegdb.com/online_c_compiler.

```
    buf1 = (char *)malloc(size);
    buf2 = (char *)malloc(size);
    diff = buf2 - buf1;

    memset(buf2, '2', size);
    printf("BEFORE: buf2 = %s ", buf2);

    memset(buf1, '1', diff);
    printf("AFTER: buf1 = %s ", buf1);

    return 0;
}
```

(a) (2pt) Compile and execute this program. What is printed?

(b) (8pt) `memset` and `printf` are invoked twice, respectively. Please list all possible security issues of them?

(c) (10pt) In terms of C/C++ memory management, what is the difference between stack and heap? In particular, which one is allocated/deallocated automatically, and which one needs programmers to take care of (you can search materials online but shouldn't directly copy)?

4. (10pt) Integer overflows can also be exploited. Consider the following C code, which illustrates an integer overflow.

```
int get_item(int idx)
{
    int array[1000];
    // initialize array
    ...
    // end initialization
    if (idx >= 1000) return -1;
    return array[idx];
}
```

(a) (6pt) What is the potential problem with this code? Besides integer overflow, which security issue is triggered, stack overflow or heap overflow?

(b) (4pt) How to solve this problem?

5. (16pt) Recall that an opaque predicate is a "conditional" that is actually not a conditional. That is, the conditional always evaluates to the same result, although it is not obvious.

(a) (6pt) Please provide two conditions of opaque predicate as example, and explain how to use them. (Please do not use too complicated conditions.)

(b) (5pt) A side effect of inserting opaque predicates is that they can slow down the execution speed. Please explain the reason of the side effect, and how to alleviate it?

(c) (5pt) A side effect of inserting opaque predicates is that they can increase the size of the executable. Please explain the reason of the side effect, and how to alleviate it?

6. (21pt) Considering the following C++ code. Function `number_ratio` calculates the ratio of number characters in the input string `s`.

```
#include <string>

float number_ratio(string s)
{
    int n = 0;
    for (int i = 0; i < s.size(); ++i)
        if (s[i] >= '0' && s[i] <= '9')
            ++n;
    return n / s.size();
}
```

(a) (3pt) Describe how to launch fuzz testing towards this function.

(b) (9pt) What bugs would you expect a fuzzer to identify in this function? Why? And how to fix this bug?

(c) (9pt) What bugs would be more difficult for a fuzzer to find in this function? Why? And how to fix this bug?

Programming Assignment – Buffer Overflow

For this assignment, we provide two programs named `login1.cpp`, `login2.cpp` and `login3.cpp`. These three programs check if the user provided username and password match the stored information in `password.txt`.

Your task is to perform buffer overflow attack towards these two test programs, by providing a username and password that is **different** from information in `password.txt` to bypass the identity check. On success of the attack, you should see message "Login successful!" (Please take a look at the code which is self explanatory on the output message). Also, you should not use any information in the `password.txt` file: it should be deemed as "secret".

`login1.cpp`, `login2.cpp` and `login3.cpp` check your username/password against the secret in `password.txt`. Note that `login2.cpp` uses a hard-coded canary to detect buffer overflows, just like stack canaries. `login3.cpp` mimics a "random" canary computed during runtime (but this one is still *less* challenging than the real-life scenarios).

To avoid plagiarism, the provided username must start with YOUR OWN student id. For example if your student ID is XXX, then the username you provide to trigger buffer overflow must start with XXX.

The content of file `password.txt` will be changed during scoring.

- (15 pt) Using a buffer overflow attack to successfully exploit `login1.cpp` or explain why that's not feasible. If feasible, submit your username and password in a file called `login1.txt`, with exactly two lines, the first line being the username, and the second line being the password.

- (25 pt) Using a buffer overflow attack to successfully exploit `login2.cpp` or explain why that's not feasible. If feasible, submit your username and password in a file called `login2.txt`, with exactly two lines, the first line being the username, and the second line being the password.

- (50 pt) Using a buffer overflow attack to successfully exploit `login3.cpp` or explain why that's not feasible. If feasible, submit your username and password in a file called `login3.txt`, with exactly two lines, the first line being the username, and the second line being the password.

When grading, we will 1) manually check `login1.txt` and `login2.txt` and `login3.txt`, and 2) try to reproduce the attack with your inputs on our end. On the other hand, if you believe certain attacks are not feasible, we will read your answers to grade accordingly.

Submission Instructions

All submissions should be done through the Canvas system. You should submit a pdf document with your answers for written component, and two files `login1.txt` and `login2.txt` for the programming component.

It is important to name your files correctly. Please check out the late submission policies on the course website (`https://course.cse.ust.hk/comp3632`) in case you didn't attend the first lecture.