Homework 2

CSIT 5730

Some questions in the written assignment come from the textbook: `Information Security Principles and Practice`. This assignment counts 4% of your final grade.

Written Assignment

1. (10pt) Reverse engineering.

   (a) (6pt) What are the two most typical disassembly techniques? Explain them briefly.

   (b) (4pt) How can developers make software reverse engineering more difficult in practice?

   (a) (**answer**): Linear disassembly and recursive disassembly. Linear disassembly starts from the first byte in the code section of the executable file to decode each byte (map one or a sequence of bytes to its corresponding assembly instruction), until the end. Recursive disassembly starts from the entry of the code, and disassembles the bytes according to the control flow.

   (b) (**answer**): Anti-disassembly techniques or Code obfuscation or any other reasonable techniques.

2. (20pt) 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) (4pt) Why is an opaque predicate a useful defense against reverse engineering attacks? Please provide one condition of opaque predicate as example.

   (b) (8pt) Suppose someone tries to use Large Language Models (LLMs) to construct opaque predicts. What would be the advantage of such a design? What would be the general concern for such a llm-based obfuscation? Or you think that's not even feasible? Explain your answer.

   (c) (8pt) What are the potential side effects of inserting opaque predicates? How to alleviate them?

   (a) (**answer**): Opaque predicates might force an attacker to analyze much more code. Example: $2 \times rax + 1 \equiv 1 (mod\ 2)$.

   (b) (**answer**): The good thing is that at this time, the path condition becomes much more complex for analysis. But it is unclear whether the model is "stable" enough such that every time this model is evaluated, it can guarantee to give the same output (recall the property of opaque predicate is that it is a "condition" that always be evaluated to true or false. (full grades for mentioning the "complexity" The general concern part is an open question and full grades as long as the answer is reasonable).

(c) (**answer**): Since the inserted code for computing the branch conditions will be executed, the additional code can result in the slow execution.

Alleviate: We can insert less obfuscation code in the original code that is executed frequently, like loops and recursive functions.

The additional code can increase the size of the executable.

Alleviate: Inserting less code for the always false branch, and designing short and effective opaque conditions.

3. (30pt) The C function `strcat` appends a copy of the source string to the destination string, which has been shown unsafe. `strncat` is a safer version with similar functionality of `strcat`. `strncat` appends the first `n` characters of `src` to `dest`, plus a terminating null-character. If the length of the `src` is less than `n`, only the content up to the terminating null-character is copied and the length of the `dest` becomes `strlen(src) + strlen(dest)`. `dest` is the return value. Its interface is

```
char *strncat(char *dest, const char *src, size_t n);
```

(a) (5pt) Consider the interface of `strcat` as "`char *strcat(char *dest, const char *src)`", why is it unsafe?

(b) (15pt) Give a concise implementation of `strncat` according to the given description of its functionality (note that this question is a "written component" in the sense that you need to embed your code within your submitted PDF file). C implementation is preferred, yet other languages (e.g., Python) is also fine.

(c) (4pt) What problem of `strcat` is solved by `strncat`?

(d) (6pt) Can `strncat` still lead to buffer overflow? If so, what additional checks need to be done to avoid buffer overflow?

(a) (**answer**): `strcat` can cause buffer overflow, since a user may input a non-terminated char array `src`.

(b) (**answer**):

```
char* strncat(char* dest, const char* src, size_t n)
{
    char* ptr = dest + strlen(dest);

    while (*src != '\0' && n--) {
        *ptr++ = *src++;
    }

    *ptr = '\0';
     return dest;
}
```

(c) (**answer**): The length of string buffers are restricted by `n`.

(d) (**answer**): Yes, the user must specify `n`, and if it was larger than the target array, an overflow will still occur.

4. (18pt) In addition to stack-based buffer overflow attacks, heap overflows can also be exploited. Consider the following C code, which illustrates a heap overflow.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main()
{
  int diff, size = 16;
  char *buf1, *buf2, *buf3;

  buf1 = (char *)malloc(size);
  buf2 = (char *)malloc(size + 8);
  buf3 = (char *)malloc(size + 16);
  diff = buf3 - buf1;

  printf("DIFF: %d\n", diff);

  memset(buf2, '1', size+8);
  printf("BEFORE: buf2 = %s ", buf2);

  memset(buf3, '2', size+16);
  printf("BEFORE: buf3 = %s ", buf3);

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

  return 0;

}
```

(a) (6pt) Compile and execute this program. What is printed?

(b) (6pt) Explain the results you obtained in part (a).

(c) (6pt) 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)?

(a) (**answer**): DIFF: 64
BEFORE: buf2 = 1111111111111111111111111 BEFORE: buf3 = 22222222222222222222222222
AFTER: buf1 = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa22222

(b) (**answer**): The `memset` may not write to the correct memory addresses. When the value of `diff` is larger than 16, `memset` will write beyond the size of `buf1`.

The 4rd `printf`: `buf1` is treated as a string. However, the `buf1[15]` is over wirtten, and this `printf` can result in buffer overflow (or over read).

(c) (**answer**): Stack is automatically allocated and reclaimed by the OS, while programmers have to explicitly use malloc and free to manage memory on the heap region.

5. (22pt) Considering the following C++ code. Function `fuzzTest` takes a string `s` as input. It returns the ratio of characters `A-Z` and `a-z` in `s`.

```
#include <string>
#include <iostream>

float fuzzTest(std::string s)
{
    int n = 0;
    for (int i = 0; i < s.size(); ++i)
        if (s[i] >= 'A' && s[i] <= 'z')
            ++n;
    return n / s.size();
}

int main(){
    std::string str;
    getline(std::cin, str);
    fuzzTest(str);
    return 0;
}
```

(a) (2pt) Describe how to launch fuzz testing towards function `fuzzTest`.

(b) (8pt) How many bugs are there in function `fuzzTest`? Explain them separately.

(c) (6pt) Among these bugs, which one can be detected by fuzzing? Why? How to fix this bug?

(d) (6pt) Why are the other bugs harder to detect by a fuzzer? And how to fix these bugs?

(a) (**answer**): Basically just randomly mutate the value of string *s*.

(b) (**answer**): Three. (1) Divided by zero in the return statement. (2) n should be cast to float type before computing n / s.size(). (3) The condition should be$>= Aand << Z|| >= aand << z$.

(c) (**answer**): The divided by zero bug can be detected. Because it causes a crash during execution.

Insert following code at the beginning of this function.

```
if (s.size() == 0)
    return 0.0;
```

(d) (**answer**): Generally the bug always returning 0 and condition error would be difficult to find, since they lead the logic error, but no crash or program hang.

`n` should be cast to `float` type before computing `n / s.size()`.

The condition should be $>= Aand << Z|| >= aand << z$.

Programming Assignment – Buffer Overflow

For this assignment, we provide three 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 three 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". We will use a different `password.txt` when grading your solution.

`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).

Important Note: To avoid plagiarism, the provided username must start with your own student id. For example if your student ID is 20918289, then the username you provide to trigger buffer overflow have to be like 20918289abcdefg. Answers does not comply this rule receive no marks.

- (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`, `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 three files `login1.txt`, `login2.txt` and `login3.txt` for the programming component. Please do NOT zip your file, submit four individual files to Canvas, your solution to written component, `login1.txt`, `login2.txt` and `login3.txt`.