CS 4337 — Programming Language Paradigms, Homework 4

https://iniyanjoseph.com | iniyanijoseph@gmail.com

Identify and explain if there is any defect(s) in the following segment of code:

```
int fd = open("foo.txt", O_RDONLY);
char* buf;
int i, len;
read (fd, &len, sizeof(len));
if (len < 0 ) {
  error("negative length");
  return;
}
buf = malloc(len+5);
read(fd, buf, len);
buf[len]='\0'; //null terminate buf
```

- There is no error handling for if the file does not exist, or if there is an error reading the file. The program needs to check if `fd == -1` and respond appropriately.
- There is no error handling for the `read` function.
- Additionally, there is no check to make sure that the string read is truly an integer. It may simply be another character or string being treated as an integer literal (which is unsafe!).
- `buf` is never `freed` from memory
- 4 extra bytes are allocated at the end of `buf`, which are not used. Although they can be unsafely accessed by manually looking at `buf[len+1], buf[len+2]`, etc, safe string methods which halt at a null terminator will never look at those locations in memory.

In the following function, the programmer intends to copy a substring of str into the buffer buf. The substring starts at position pos1 and ends at pos2. The programmer carefully checks the length of the substring before copying it.

```
int parse_string(char* str, int pos1, int pos2){
  ...
  char buf[1024];
  if (pos2-pos1+1 > 1024) exit(-1);
  strcpy(buf, src);
  ...
  return 1;
}
```

Will this function cause a buffer overflow? Give an explenation.

There are exactly 1024 bytes in the buffer. If pos2-pos1+1 > 1024, then pos2-pos1 > 1023. In a valid substring, the largest value of pos2-pos1 is 1023. Thus if pos2-pos1 is positive, the function will not cause a buffer overflow in the strcpy (assuming that src is a substring [pos1, pos2]). However, if pos2 < pos1, then this value may be negative, which could lead to a buffer overflow.

How would you fix the above function to avoid buffer overflow?

To fix the above function, I would use `strncpy` rather than `strcpy`, ensure that `buf` is null terminated, and check that pos1 <= pos2.

Given the following C function, the function is designed to “combine”two strings together by alternately taking the characters from two strings. For example, calling the function to combine “ABC” and “def”. should return “AdBeCf”.

```
char* zip(char* a, char* b){
  char* result;
  int len, i;
```

```
  len = strlen(a);
  result = malloc(2*len);
  for (i=0; i < len; i++){
    result[2*i]=a[i];
    result[2*i+1]=b[i];
  }
  return result;
}

int main(){
  char* a ="...";
  char* b ="...";
  char* merged = zip(a, b);
  printf("Merged string:%s\n", merged);
  return 0
}
```

Is this function implemented securely? Given an explanation.

No, this function is not implemented securely. If the length of b is less than the length of a, then unknown values will be written to result (buffer overflow). Additionally, result is not null-terminated.

Provide a safer implementation for tihs function. Give an explanation

```
#define min(a,b) (((a)<(b))?(a):(b))

char* zip(char* a, char* b){
  char* result;
  int len, i;
  len = min(strnlen(a), strnlen(b)); // Use strnlen instead of strlen. The strings are
defined as literals so this should not be a problem in general, but if "..." is changed
to be another string which may not be null terminated, this could be problematic. Also,
use the smaller of the two strings as the length, to ensure that no buffer overflow
occurs.
  result = malloc(2*len + 1);
  for (i=0; i < len; i++){
    result[2*i]=a[i];
    result[2*i+1]=b[i];
  }
  result[2*len] = '\0'; // Add null terminating character to end of the result.
  return result;
}

int main(){
  char* a ="...";
  char* b ="...";
  char* merged = zip(a, b);
  printf("Merged string:%s\n", merged);
  return 0
}
```

Consider a scenario where an attacker exploits a vulnerability in a program to overwrite a legitimate function pointer (func_ptr) with a malicious function's address. The attacker's goal is to divert the program's control flow to the malicious function, which could potentially execute harmful actions.
With CFI enabled, describe in detail how it prevents the attack and ensures the control flow integrity

With CFI enabled, the program maintains a label `L` to `func_ptr` of the original value of `func_ptr`. When `func_ptr()` is called in `main`, the label of the function called by the program must match `L`. Since there is no way to reference `malicious_function` in the original program from `main`, it's label will not match, and an error will be thrown by the program.

The attacker's objective is to execute the system command /bin/sh by exploiting the vulnerable function using a ROP (Return-Oriented Programming) attack.
Craft a ROP chain and explain in detail each step of performing a ROP attack.

The first step of ROP is to use a stack-smashing attack to overwrite the return adress of a vulnerable function. Arbitrary memory in the stack can be overwritten using `strcpy` by simply providing a sufficiently large value of `input` to `vulnerable_function(char* input)`. Second, we must overwrite the return adress and point it to a gadget. Point it to `pop rdi; ret;`. Next, we must write `/bin/sh` to the stack as the next value. Thus, when `pop rdi; ret;` is executed, the value stored in `rdi` will be `/bin/sh`. Finally, the `system()` call should be executed so that a shel is spawned.

Explain in detail a real-world example of a DOM-based XSS attack and its impact.

One real-world example of a DOM-based XSS attack is the British Airways Breach in 2018. Attackers used a DOM-based XSS vulerability to incject JavaScript into BritishAirways' website. They were able to use this to steal credit credit card information from customers.

An attacker wants to bypass a server-side filter that removes

tags to prevent XSS. Describe one technique they might use to inject a payload that still executes JavaScript, and explain why it might work.

```
<img src="http://thiswebsitemaynotexist.com" onerror=alert(document.cookie);>
```

Although there is no script tag, the malicious cookie will still be executed when the src is not found.