IERG 4130 - Introduction to Cyber Security (2022 Fall)
Assignment 1: Software Security - Solution

Q1. In which memory segments are the variables (including variables str and j) in the following code located?

```
1 int a = 0;
2 void func(char *str, int b)
3 {
4       int c;
5       long d;
6       char buf[64];
7       char *ptr = malloc(sizeof(int));
8 }
```

Answer:

.data segment or .bss: a

stack segment: c, d, buf, str, b, ptr

heap segment: *ptr

Tips:

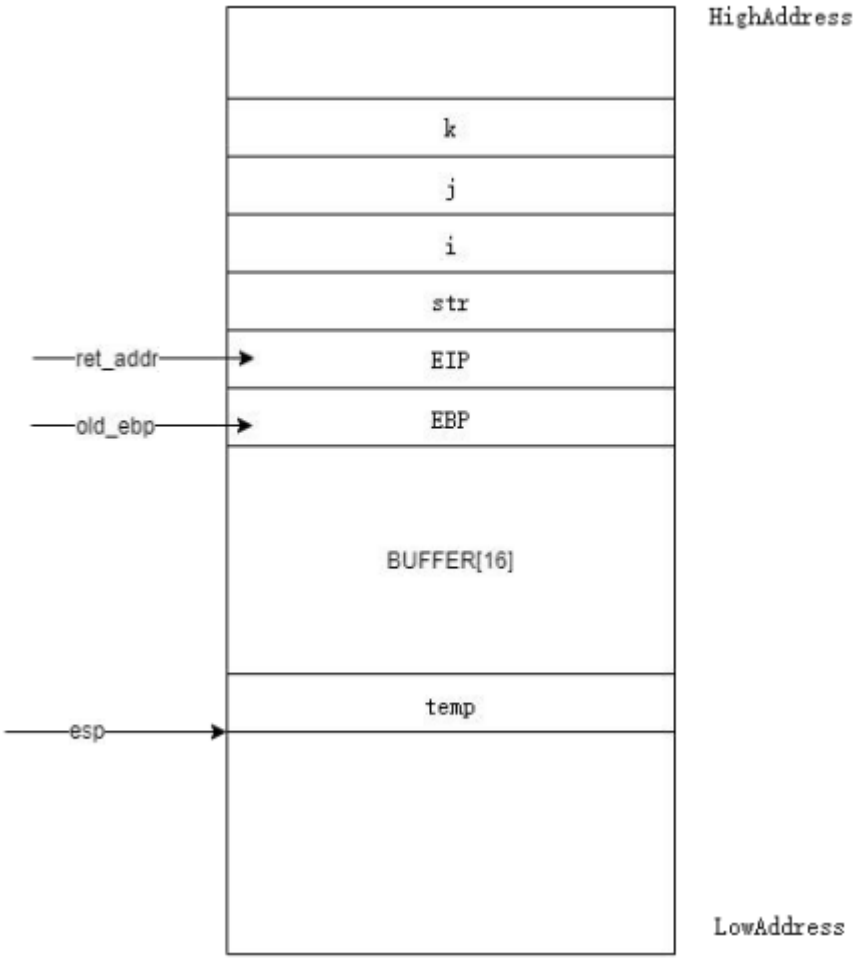
Initialized global variables are stored in .data segment.
Uninitialized global variables are stored in .bss segment.
Arguments passed to the callee are stored in the stack.
Non-static local variables (not dynamically allocated) are stored in the stack. (*ptr* itself)
Dynamically-allocated variables are stored in the heap. (**ptr*)

Q2. Please draw the function stack frame for the following C function.

```
1 int bof(char *str, int i, int j, int k)
2 {
3       char buffer[16];
4       int temp = i + j + k;
5       strcpy(buffer, str);
6       return temp;
7 }
```

Answer:

Students should at least include parameters passed to the routine, return address, old_ebp, temp and buffer in the stack frame.
They should also notice the difference in size between integer variable, temp, and character arraya, buffer. Besides, in this case, saved registers are not necessary and stack alignment can be ignored.



Marking: the order should be correct, and the buffer should be longer than temp

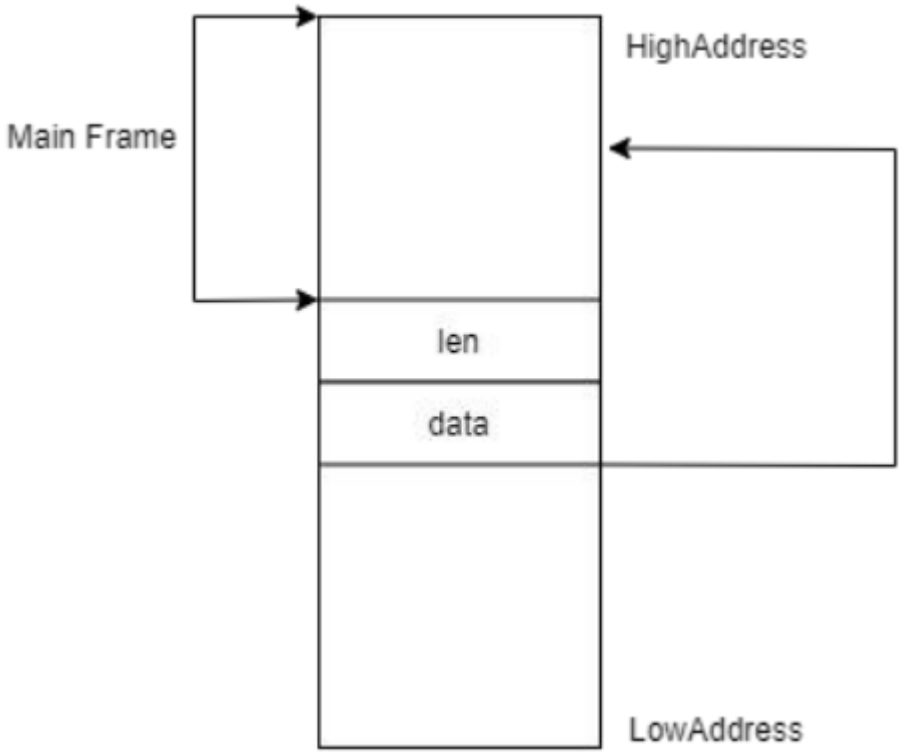
Q3. Is this function safe? Justify your answer and propose possible mitigation by modifying the source code with the necessary explanation in your answer. (10')
Hint: void *memcpy (void *destination, const void *source, size_t num);

```
1 void func(char *data, int len)
2 {
3       char buf[64];
4       if (len > 64)
5             return;
6       memcpy(buf, data, len);
7 }
```

Answer:
the difference between data type size_t and int. Type size_t is considered an unsigned integer featuring enough bytes to accommodate any size type.

This function is unsafe. Because when the length of the source passed into memcpy function, int len, is longer than the bytes in the source string, memcpy function will actually read int len bytes data even if it meets the special symbol '\0'. Thus, if int len is more than the bytes of source code it will

read the memory locations protected by OS and cause buffer overflow leading to a program crash. As shown in the figure below, the frame of the function calling func will potentially be corrupted.. Additionally, if the value of the variable, int len, passed into func is negative, it will be decoded as an unsigned integer and the value will be considered as a very big number by program. The memcpy function will copy memory locations beyond the boundary, which will cause a program crash or some other unexpected consequences.



One solution is to use a safe implementation of memcpy function named memcpy_s which will check the boundary to prevent the stack from being overflowed. .
Another solution is to add proper sanitization in source code such as a length restriction.
e.g., the code in fourth line can be replaced by :

```
if(len > 64 || len > sizeof(data) || len < 0)
    return; // or any other exit statement
```

Q4. The following function is called in a privileged program. The argument str points to a string that is entirely provided by users (the size of the string is up to 300 bytes). When this function is invoked, the address of the buffer array is 0x66DD0010, while the return address is stored in 0x66DD0050. Please write down the string that you would feed into the program, so when this string is copied to buffer and when the bof() function returns, the privileged program will run your code. In your answer, you don't need to write down the injected code, but the offsets of the key elements in your string need to be correct. (10')

```
int bof(char *str)
{
   char buffer[24];
   strcpy(buffer,str);
```

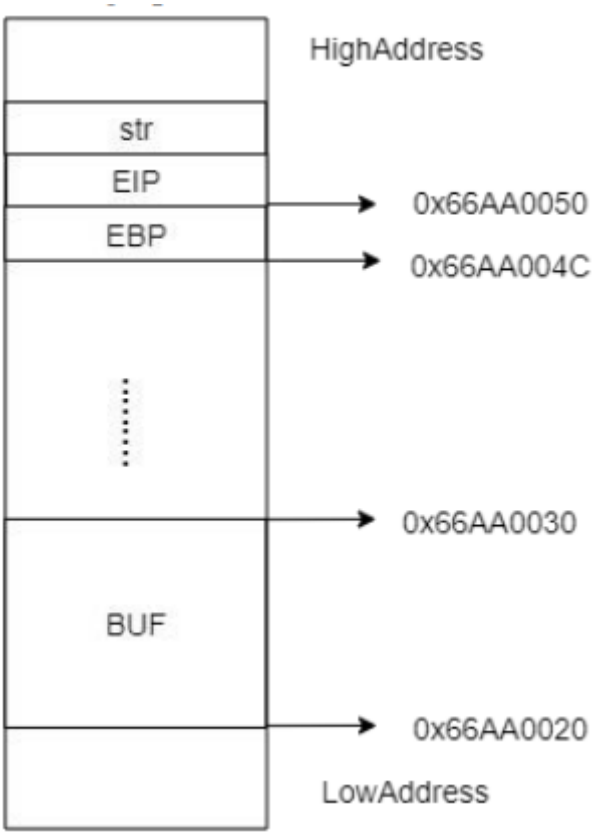
```
    return 1;
}
```

Answer:

0x66DD0050 - 0x66DD0010 = 0x40 = 64

str[64..67] contains the target address which you want to overwrite the original return address by.

scope of the New value is 0x66DD0054~0x66DD0010 + 300 - sizeof(shellcode)

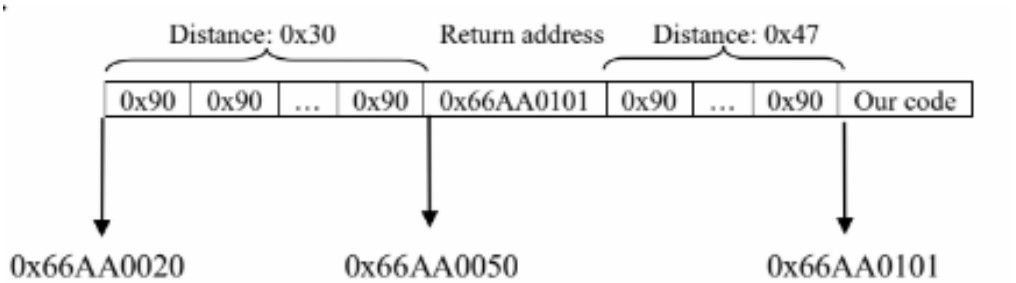


The “0x90” is NOP (No Operation) instruction in x86 assembly.

For example, here we set the return address as 0x66AA0101, which is not unique in this case and can be changed to another address.

There is also a limitation that the value should be larger than 0x66aa0054 and smaller than (0x66AA0020 + 300 – sizeof(shellcode)).

Note that due to the property of the strcpy() function, the string shouldn’t contain ‘\x00’, otherwise strcpy() will stop just when it meets ‘\x00’.



Q5. Please answer the following short questions about software security.

1) What are the common grounds between stack overflow and heap overflow? (3’)
2) Which kinds of variables can be overwritten by a stack overflow attack? (3’)

3) Pls discuss the limitations of the following control-flow hijack defense techniques respectively.
Hint: How to defeat each kind of mitigation technique?

a. StackGuard(Canary) (3')
b. No-Execution-Bit (3')
c. Address Space Layout Randomization (ASLR) (3')

1. In order to hijack the control flow, both stack overflow and head overflow are to indirectly update the value of the instruction register (EIP in our case) by what the hacker wants. The value of EIP cannot be manipulated directly, so the hacker always tries to find a way to exploit the memory read and written in either stack or heap to indirectly make the EIP changed by the carefully selected address.
2. Variables assigned to contiguous memory address space. In stack segment, for example, some array-like variables like built-in arrays with types including but not limited to basic types; format string, and point variables with different types.
3. Q: How to defeat each kind of mitigation?
    a) The StackGuard inserts a random value just before the return address and old ebp, to detect whether an overflow has occurred by checking the random code before and after function frame initialization and destroy. It also has a Terminator symbol which can stop most string operators. The attacker can still change the return address by overwriting a specific pointer, so as not to modify the value of the canary to achieve the attack.
    b) Non-Execution-bit (NX-bit) prohibits the execution of contents in stack segment by making it read-only. For example, if the stack is no-executable, it may be that after jumping to the shellcode, the shellcode cannot be executed, and the system will immediately trigger an exception, indicating a segmentation fault or other pre-designed error signal. However, it's often defeated by ret2libc attacks. To bypass the NX-bit protection of the stack segment via ret2libc attack, the attacker exploits some carefully selected function existing in the main memory as a member of the standard C library (libc) by jumping to those libc functions with arguments passed from the current stack frame.
    c) ASLR will randomize the memory address layout at each time of loading the executable program so that the address layout of the program cannot be guessed easily as before. The attacker can't know those important addresses for control flow hijacking such as the base address of buffer variables, the element of the stack frame which make it much harder to design the payload. . To defeat it,
        i) Partial EIP overwrite - Only overwrite part of EIP, or use a reliable information disclosure (i.e., format string vulnerability) in the stack to find what the real EIP should be, then use it to calculate your target. We still need a non-ASLR module for this though.
        ii) Bruteforce - If you can try an exploit with a vulnerability that doesn't make the program crash, you can brute-force 256 (applicable for 32-bit machines) different target addresses until it works.
        iii) Actually, there are more advanced and clever attacking approaches to defeat ASLR but far beyond the scope of this course. Understanding

the big ideas of the aforementioned two approaches with some practices in lab sessions is good enough.

Q6. A student proposes to change how the stack grows. Instead of growing from high address to low address, the student proposes to let the stack grow from low address to high address. This way, the buffer will be allocated above the return address, so overflowing the buffer will not be able to affect the return address. Please comment on this proposal by combining the following program. (bonus 5')
(Hint: what will happen if a too-large string is passed to the bar() function.)

```
1 void bar(char *str)
2 {
3       char c[7];
4       strcpy(c, str);
5 }
6 void foo()
7 {
8       bar("overlflow");
9 }
```

Answer:

Actually, this is not a good idea. Here we just ignore all historical reasons but focus on whether it can be effective to prevent buffer overflow. The root cause of a buffer overflow vulnerability is not the growth direction of a stack, but the lack of border checking.

Even if the stack were growing upwards, the return address of a routine can still be overwritten. For an upward stack, an attacker can overflow a local variable in the caller's stack to overwrite the return address of the callee.