Return-to-libc Attack and ROP

S5.1. After using the "`-z noexecstack`" option to compile a C program, a buffer-overflow attack that causes the vulnerable program to return to the code on the stack is supposed to fail, but some students find out that the attack is still successful. What could be the reason? The students did everything correctly.

S5.2. In the function epilogue, the previous frame pointer, which is stored in the area below the return address, will be retrieved and assigned to the `ebp` register. However, when we overflow the return address, the previous frame pointer region is already modified, so after the function epilogue, `ebp` contains some arbitrary value. Does this matter?

S5.3. Instead of jumping to the `system()` function, we would like to jump to the `execve()` function to execute "`/bin/sh`". Please describe how to do this. You are allowed to have zeros in your input (assume that `memcpy()` is used for memory copy, instead of `strcpy()`)

S5.4. As we know, the `system()` function calls `/bin/sh`, which is a symbolic link to `/bin/bash`. Recent versions of `bash` will drop the privilege if it detects that the effective user ID and the real user ID are different. Assume that we still want to use `system()` in our Return-to-libc attack, please describe how you can overcome this challenge. You are allowed to have zeros in your input (assume that `memcpy()` is used for memory copy, instead of `strcpy()`)

S5.5. When launching the return-to-libc attack, instead of jumping to the beginning of the `system()` function, an attacker causes the program to jump to the first instruction right after the function prologue in the `system()` function. Please describe how the attacker should construct the input array.

S5.6. Can address space layout randomization help defeat the return-to-libc attack?

S5.7. Does ASLR in `Linux` randomize the addresses of library functions, such as `system()`?

S5.8. ★
Assuming that we do not have the function `system()` that we can return to in our Return-to-libc attack, but we know the instruction sequence A_1, ..., A_m, B_1, ..., B_n, C_1, ..., C_t can spawn a shell for us. If all the instructions in this sequence are located in contiguous memory, we can simply jump to its beginning. Unfortunately, that is not the case. The instructions in this sequence actually come from three sub-sequences, A, B, and C, each of which is found at the end of a function, i.e., there is always a `ret` instruction at the end of each sub-sequence. Each sub-sequence's address is given below:

```
0xAABB1180:   A1, ..., Am, ret
0xAABB2290:   B1, ..., Bn, ret
0xAABB33A0:   C1, ..., Ct, ret
```

Obviously, when we overflow a buffer, we will place `0xAABB1180` in the return address field, so when the function returns, it will jump to the beginning of the sub-sequence A. Please describe what other values that you would place on the stack, so when the sub-sequence A returns, it will jump to the sub-sequence B, and when the sub-sequence B returns, it will jump to the sub-sequence C.

What is described above is called Return-Oriented Programming (ROP), which is a generalized Return-to-libc technique. The Return-to-libc technique depends on the availability of some functions such as `system()`; if such functions are not in the memory, the technique will not work. With the ROP technique, an attacker can carefully choose machine instruction sequences that are already present in the machine's memory, such that when these sequences are chained together, they can achieve the intended goal. These sequences are called `gadgets`, which typically end in a return (`ret`) instruction and are located in a subroutine within the existing program and/or shared library code. The `ret` instruction is necessary, because the ROP technique depends on it to jump from one sub-sequence to another. ROP gadgets can be chained together to allow an attacker to perform arbitrary operations. Using ROP, the attacker does not need to call functions to mount an attack.

S5.9. Function `foo()` has a buffer overflow problem when copying your input to a buffer that is inside its stack frame. We would like to get it to return to a sequence of function calls: `bar()` ➞ `bar()` ➞ `bar()` ➞ `xyz(3, 5)` ➞ `exit()`. Assuming we know their address. Please describe how you would use the buffer overflow problem to construct the stack before letting `foo()` return. You should provide a stack diagram in your answer.

```
|                    |
|   provide          |
|   your             |
|   answer           |
|                    |
|                    | <-- ebp of foo()
```

S5.10. Function `foo()` has a buffer overflow problem when copying your input to a buffer that is inside its stack frame. You would like to get it to return to a library function `xyz(0)`. You cannot skip `xyz()`'s function prologue); nor can you put any zero in your input. There is another library function called `setzero(addr)`, which can set the 4-byte memory at address `addr` to zero. Please describe how you would construct your input. We do not care whether the program will crash or not after `xyz(0)` returns. You should provide a stack diagram in your answer.

```
|                    |
|   provide          |
|   your             |
|   answer           |
|                    |
|                    | <-- ebp of foo()
```

S5.11. The problem adds an additional requirement to Problem S5.10.. Function `foo()` has a buffer overflow problem when copying your input to a buffer that is inside its stack frame.

You would like to get it to return to a library function sequence `xyz(0x11111111)` ➞ `xyz(0)` ➞ `xyz(0x22222222)`. You cannot skip `xyz()`'s function prologue); nor can you put any zero in your input. There is another library function called `setzero(addr)`, which can set the 4-byte memory at address `addr` to zero. Please describe how you would construct your input. We do not care whether the program will crash or not after `xyz(1)` returns. You should provide a stack diagram in your answer.