x86 Lab 4
Buffer Overflows

Part 1:

Compile exploitable.c using the gcc command below:

```
gcc -g -o exploitable exploitable.c
```

For each of the commands below, predict what the program's output will be and explain your answer. Run the program. Does the output match your prediction? If not, explain why the program produces the output that it does.

Command	Predicted Output	Explanation	Actual Output	Explanation
`./exploitable`	`bufferA: aaa` `bufferB: bbb`	bufferA is initialized to "aaa\0", and bufferB to "bbb\0". Neither is changed before they are printed out.	`bufferA: aaa` `bufferB: bbb`	(Example)
`./exploitable` `<ESC> 3f`	`bufferA: aaa` `bufferB: fff`	bufferA is initialized to "aaa\0", and bufferB to "fff\0". Neither is changed before they are printed out.	`bufferA: aaa` `bufferB: fff`	bufferA is initialized to "aaa\0", and bufferB to "fff\0". Neither is changed before they are printed out.
`./exploitable` `<ESC> 5f`	`bufferA: aaa` `bufferB: fff`	bufferA is initialized to "aaa\0", bufferB to "fff\0". Because the size of bufferB is limited so that output size will limited	`bufferA: aaa` `bufferB: fffff`	because the StackGuard, So, it can expand the dimension.

In the third command, we run into a protection method used by modern compilers to avoid stack-smashing exploits: StackGuard.

StackGuard places 'canary' values on the stack in between the local variables and the control data (old_ebp and RIP). If these canaries are overwritten, as in a stack smashing exploit, the program detects the change and exits with an error.

Part 2:

Recompile the exploitable.c program using the following command to turn off the stack protection:

```
gcc -g -fno-stack-protector -o exploitable exploitable.c
```

For each of the commands below, predict what the program's output will be and explain your answer. Run the program. Does the output match your prediction? If not, explain why the program produces the output that it does.

Command	Predicted Output	Explanation	Actual Output	Explanation
`./exploitable <ESC> 5f`	`bufferA: aaa` `bufferB: fff`	bufferA is initialized to “aaa\0”, bufferB to “fff\0”. Because the size of bufferB is limited so that output size will limited	`bufferA: f` `bufferB: fffff`	for bufferB ”fffff” for bufferA “f\0a\0” bufferA starts to point to the last of f in bufferB
`./exploitable <ESC> 4f`	`bufferA: aaa` `bufferB: fff`	bufferA is initialized to “aaa\0”, bufferB to “fff\0”. Because the size of bufferB is limited so that output size will limited	`bufferA:` `bufferB: ffff`	for bufferA ‘”\0aa\0” show “\0” for bufferB “ffff” show “\0”

Part 3:

1. Where in the stack is the Return Instruction Pointer (RIP) stored?
   ebp+0x4
2. How is the RIP used when a function returns? Where should this cause the program to continue execution?(some stack at top store encode function need to decide which position to return)
   RIP use to save the pointer, which the function is called. Then the function is calling the program returns to the Register that it pointing before
3. What are the hexadecimal values of the ASCII characters ‘a’, ‘b’, ‘f’, and ‘\0’ (the NULL character, used to end a string in C)?
   ‘a’: 0x61
   ‘b’: 0x62
   ‘f’: x66
   ‘\0’: 0x00

Load the ‘exploitable’ binary into gdb and set a breakpoint at line 12. Run the program once using the command in each row of the table below, and fill in the values of the following memory locations when the program stops at the breakpoint.

Command (in gdb!)	Value (in hex) of bufferA (x/4bx bufferA)	Value (in hex) of bufferB	Value (in hex) of old_ebp	Value (in hex) of the RIP
run	0x61 0x61 0x61 0x00	0x62 0x62 0x62 0x00	0x28 0xf7 0xff 0xbf	0xc6 0x84 0x04 0x08
run <Esc> 3f	0x61 0x61 0x61 0x00	0x66 0x66 0x66 0x00	0x18 0xf7 0xff 0xbf	0xc6 0x84 0x04 0x08
run <Esc> 17f	0x66 0x66 0x66 0x66	0x66 0x66 0x66 0x66	0x66 0x00 0xff 0xbf	0xc6 0x84 0x04 0x08
run <Esc> 24f	0x66 0x66 0x66 0x66	0x66 0x66 0x66 0x66	0x66 0x66 0x66 0x66	0x66 0x66 0x66 0x66