# Homework 2 Challenge Writeup

**Reverse Engineering Part 2**

Lindsay Von Tish

lmv9443@nyu.edu

02/14/2024

Table of Contents

[Homework 2 Challenge Writeup 1](#_Toc158817196)

[Challenge Details 2](#_Toc158817197)

[Bridge of Death 2](#_Toc158817198)

[Overview 2](#_Toc158817199)

[Details 2](#_Toc158817200)

[Attempt 10](#_Toc158817201)

[Dora 11](#_Toc158817202)

[Overview 11](#_Toc158817203)

[Details 11](#_Toc158817204)

[Attempt 15](#_Toc158817205)

[Appendix A: Student Information 16](#_Toc158817206)

[Appendix B: Tools 16](#_Toc158817207)

[Appendix C: Postage 17](#_Toc158817208)

[Overview 17](#_Toc158817209)

[Details 17](#_Toc158817210)

[Challenge Attempt 19](#_Toc158817211)

[Appendix D: BoD\_Remote.py 23](#_Toc158817212)

[Appendix E: Dora\_BF.py 25](#_Toc158817213)

# Challenge Details

## Bridge of Death

### Overview

|  |  |  |
| --- | --- | --- |
| Bridge of Death | | |
| **150 Points** | Flag Value | flag{@\_W1tch\_W3'v3\_G0t\_@\_W1tch!!!!!!!!!} |
| Location | offsec-chalbroker.osiris.cyber.nyu.edu 8005 |
| Lore | Monty Python and the Holy Grail |

### Details

The program asks the user a series of questions that anyone familiar with Monty Python and the Holy Grail should recognize. On the first run of bridge\_of\_death, I fell to my death.

|  |
| --- |
| gdb ./bridge\_of\_death  ...omitted for brevity...  Reading symbols from ./bridge\_of\_death...  (No debugging symbols found in ./bridge\_of\_death)  (gdb) r  Starting program: /home/kali/Desktop/2-Week/bridge\_of\_death  What is your name?  Juneau  What is your quest?  Pass this class!  kek  kek  kek  kek  Auuuuuuuugh!  [Inferior 1 (process 5082) exited normally]  (gdb) q |

I Fall to my Death

The bridge\_of\_death main() method, shown below after decompilation with *Ghidra*, calls three different "question" methods. Each method returns a boolean value that the method uses to determine whether the program continues or throws the user into the Gorge of Eternal Peril.

|  |
| --- |
| undefined8 main(EVP\_PKEY\_CTX \*param\_1)  {  int iVar1;    init(param\_1);  puts(  "Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.\n\nWhat is your name?"  );  iVar1 = question1();  if (iVar1 == 0) {  throw\_into\_gorge\_of\_eternal\_peril();  }  puts("What is your quest?");  iVar1 = question2();  if (iVar1 != 0) {  throw\_into\_gorge\_of\_eternal\_peril();  }  puts("What is the air-speed velocity of an unladen swallow?");  iVar1 = question3();  if (iVar1 != 0) {  throw\_into\_gorge\_of\_eternal\_peril();  }  puts("Right. Off you go.");  print\_flag();  return 0;  } |

Main Method

#### Question 1: What is your name?

The bridge\_of\_death binary stores the answer to question 1 in plaintext.

|  |
| --- |
| strings bridge\_of\_death| grep -i "Lancelot"  My name is Sir Lancelot of Camelot. |

Strings

The question1() method compares the user-entered text to the string "My name is Sir Lancelot of Camelot."

|  |
| --- |
| void question1(void)  {  long in\_FS\_OFFSET;  char guess [136];  long local\_10;    local\_10 = \*(long \*)(in\_FS\_OFFSET + 0x28);  fgets(guess,0x80,stdin);  strcmp("My name is Sir Lancelot of Camelot.",guess);  if (local\_10 != \*(long \*)(in\_FS\_OFFSET + 0x28)) {  /\* WARNING: Subroutine does not return \*/  \_\_stack\_chk\_fail();  }  return;  } |

Question 1

However, subsequent testing showed that question1() would return true regardless of the name entered.

|  |
| --- |
| nc offsec-chalbroker.osiris.cyber.nyu.edu 8005  Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.  What is your name?  My name is Sir Lancelot of Camelot.  What is your quest?  ...omitted for brevity...  nc offsec-chalbroker.osiris.cyber.nyu.edu 8005  Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.  What is your name?  Juneau  What is your quest? |

Name Demonstration

#### Question 2: What is your quest?

The question2() function uses get\_number() to take in two user-entered integers. The get\_number() function will return 0 if the entered value is not a valid integer. It is the same function used in Postage, shown for reference in [Appendix C](#_Appendix_C:_Postage).

|  |
| --- |
| bool question2(void)  {  undefined4 uVar1;  int iVar2;  int iVar3;    uVar1 = get\_number();  iVar2 = get\_number();  iVar3 = func2(uVar1,0,0x14);  return iVar2 != iVar3;  } |

Question 2

After reading in the numbers, the function calls func2() and checks to see if the return value is equal to the second user-entered integer. The method takes three inputs: the user-entered guess (p1) and two other numbers.

|  |
| --- |
| int func2(int p1,int p2,int p3)  {  int v1;  int v2;  puts("kek");  v1 = p2 + (p3 - p2) / 2;  if (p1 < v1) {  v2 = func2(p1,p2,v1 + -1);  v1 = v2 + v1;  }  else if (v1 < p1) {  v2 = func2(p1,v1 + 1,p3);  v1 = v2 + v1;  }  return v1;  } |

Func2

The recursion in func2() is a red herring; the important part of the problem is the math used to calculate v1. If p1 is equal to v1, then func2() skips the recursion entirely and returns the value of v1. The last two parameters func2() receives are hardcoded by question2(), making it easy to solve for the correct value of p1.

|  |
| --- |
| a = Int('a')  b, c = Reals('b, c')  #g = Int('g')  s = Solver()  s.add(b == 0)  s.add(c == 20)  s.add(a == b + (c-b)/2)  print(s.check())  print(s.model())  >> sat  >> [a = 10, c = 20, b, = 0] |

Solver Script

The correct value of p1 is 10, which should be entered for both guesses to answer question 2.

|  |
| --- |
| What is your name?  Juneau  What is your quest?  10  10  kek  What is the air-speed velocity of an unladen swallow?  I have no clue  Auuuuuuuugh! |

Success

#### Question 3

The question3() function is much longer and more complex than the previous questions.

|  |
| --- |
| undefined8 question3(void)  {  long lVar1;  uint guess1;  uint guess2;  undefined8 flag;  long in\_FS\_OFFSET;  int counter;    lVar1 = \*(long \*)(in\_FS\_OFFSET + 0x28);  counter = 1;  do {  if (9 < counter) {  flag = 0;  LAB\_0010159d:  if (lVar1 != \*(long \*)(in\_FS\_OFFSET + 0x28)) {  /\* WARNING: Subroutine does not return \*/  \_\_stack\_chk\_fail();  }  return flag;  }  guess1 = get\_number();  guess2 = get\_number();  if ((0xff < guess1) || (0xff < guess2)) {  flag = 1;  goto LAB\_0010159d;  }  if (counter != (char)forestOfEwing[(ulong)guess2 + (ulong)guess1 \* 0x100]) {  flag = 1;  goto LAB\_0010159d;  }  counter = counter + 1;  } while( true );  } |

Question 3

The important part of the code is in the highlighted if statement.

At each iteration of the loop, question3() compares the number of the current iteration (counter) to a character in the array forestOfEwing. The character's position is determined using two user-entered integers (guess1 and guess2).

The question3() disassembly gives more insight into how bridge\_of\_death calculates the character's location in forestOfEwing.

|  |
| --- |
| 0x000055555555554b <+98>: mov -0x94(%rbp),%edx  0x0000555555555551 <+104>: mov -0x98(%rbp),%eax  0x0000555555555557 <+110>: shl $0x8,%rax  0x000055555555555b <+114>: add %rax,%rdx  0x000055555555555e <+117>: lea 0x2abb(%rip),%rax #0x555555558020 <forestOfEwing>  0x0000555555555565 <+124>: add %rdx,%rax  0x0000555555555568 <+127>: movzbl (%rax),%eax  0x000055555555556b <+130>: movsbl %al,%eax  0x000055555555556e <+133>: cmp %eax,-0x9c(%rbp)  0x0000555555555574 <+139>: je 0x555555555584 <question3+155> |

Question 3 Disassembly

The program calculates the index in the array using the two guesses and then adds it to the address of forestOfEwing. After saving the address in RAX, the program uses it to load the character value into the EAX register before the comparison. Setting the value of RAX to the address of the counter will ensure that the values are equal.

|  |
| --- |
| gdb ./bridge\_of\_death  (gdb) break \*0x0000555555555568  Breakpoint 4 at 0x555555555568  (gdb) c  ...omitted for brevity...  What is the air-speed velocity of an unladen swallow?  1  2  ...omitted for brevity...  Breakpoint 4, 0x0000555555555568 in question3 ()  (gdb) info registers rax  rax 0x555555558122 93824992248098  (gdb) set $rax=$rbp-0x9c  (gdb) info registers rax  rax 0x7fffffffdd24 140737488346404  (gdb) c  Continuing.  1  2  ...omitted for brevity...  Breakpoint 4, 0x0000555555555568 in question3 ()  (gdb) info registers rax  rax 0x555555558122 93824992248098  (gdb) set $rax=$rbp-0x9c  (gdb) info registers rax  rax 0x7fffffffdd24 140737488346404  (gdb) c  Continuing.  Right. Off you go.  ERROR: no flag found. |

Debugging Solution

Unfortunately, this method only works on a local instance of the program. Because of the size limitations on the guesses, the address used must point to a character in forestOfEwing. The program code contains the forestOfEwing array, which is 65535 characters long.

After exporting the forestOfEwing data from the bridge\_of\_death disassembly in *Ghidra*, I put it in a Python list and used a script to search for characters matching each possible counter value.

|  |
| --- |
| def question3(p):  i = 0  index = [[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]]  for value in f0e:  if value == 0x1:  index[0][0] = i//256  index[0][1] = i%256  elif value == 0x2:  index[1][0] = i//256  index[1][1] = i%256  elif value == 0x3:  index[2][0] = i//256  index[2][1] = i%256  elif value == 0x4:  index[3][0] = i//256  index[3][1] = i%256  elif value == 0x5:  index[4][0] = i//256  index[4][1] = i%256  elif value == 0x6:  index[5][0] = i//256  index[5][1] = i%256  elif value == 0x7:  index[6][0] = i//256  index[6][1] = i%256  elif value == 0x8:  index[7][0] = i//256  index[7][1] = i%256  elif value == 0x9:  index[8][0] = i//256  index[8][1] = i%256  i +=1  i = 0  while i < 9:  msg = str(index[i][0]).encode()  p.sendline(msg)  msg = str(index[i][1]).encode()  p.sendline(msg)  i += 1  return index |

Solver Script

The script uses the location of each matching value to calculate the integer values for both guesses.

|  |
| --- |
| python3 Q3-search.py  [[64, 234], [4, 44], [132, 146], [14, 148], [41, 138], [170, 133], [173, 99], [12, 9], [73, 199]] |

Results

|  |
| --- |
| gdb ./bridge\_of\_death  GNU gdb (Debian 13.2-1) 13.2  ...omitted for brevity...  Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.  What is your name?  l  What is your quest?  10  10  kek  What is the air-speed velocity of an unladen swallow?  64  234  4  44  132  146  14  148  41  138  170  133  173  99  12  9  73  199  Right. Off you go.  ERROR: no flag found. |

Local Success

### Attempt

After verifying the results locally, I attempted the remote challenge. The solver script is available in [Appendix D](#_Appendix_D:_BoD_Remote.py).

|  |
| --- |
| python3 BoD\_Remote.py  [+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 8005: Done  Answering Question 1  Juneau  Answering Question 2  ['10', '10']  Answering Question 3  [[64, 234], [4, 44], [132, 146], [14, 148], [41, 138], [170, 133], [173, 99], [12, 9], [73, 199]]  b"@\_W1tch\_W3'v3\_G0t\_@\_W1tch!!!!!!!!!}\n"  [\*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 8005 |

Results

## Dora

### Overview

|  |  |  |
| --- | --- | --- |
| Bridge of Death | | |
| **150 Points** | Flag Value | flag{mmaped\_some\_fresh\_pages} |
| Location | offsec-chalbroker.osiris.cyber.nyu.edu 1250` |
| Lore*-a* | Dora the Explorer |

### Details

The first run of Dora made it clear that the wrong input would cause a segmentation error.

|  |
| --- |
| gdb ./dora  What's the key?  13  Program received signal SIGILL, Illegal instruction.  0x00007ffff7fc2000 in ?? ()  (gdb) q  gdb ./dora  What's the key?  111  Program received signal SIGSEGV, Segmentation fault.  0x00007ffff7fc2001 in ?? ()  (gdb) q |

First Run

The decompiled Dora main() method is complex but provides valuable insight into the program's operations.

|  |
| --- |
| undefined8 main(EVP\_PKEY\_CTX \*param\_1)  {  undefined8 \*puVar1;  long lVar2;  undefined8 uVar3;  ulong counter;    init(param\_1);  puVar1 = (undefined8 \*)mmap((void \*)0x0,0x1000,7,0x22,-1,0);  uVar3 = DAT\_00104028;  \*puVar1 = read\_flag;  puVar1[1] = uVar3;  ...omitted for brevity...  puVar1[8] = DAT\_00104060;  puVar1[9] = uVar3;  puts("What\'s the key?");  lVar2 = get\_number();  if ((lVar2 < 0) || (0xff < lVar2)) {  puts("That key is out of range :( Try again?");  uVar3 = 1;  }  else {  for (counter = 0; counter < 0x50; counter = counter + 1) {  \*(byte \*)(counter + (long)puVar1) = \*(byte \*)(counter + (long)puVar1) ^ (byte)lVar2;  }  (\*(code \*)puVar1)();  uVar3 = 0;  }  return uVar3;  } |

Main Method

The program creates a memory map (mmap) in the calling process’s virtual address space. Then, it adds data to the mmap from different locations in the program memory before taking in and validating a user-input integer.

A screenshot of a computer code

Description automatically generated  
Stored Data

After this setup, the program iterates through the saved data, performing an exclusive or (XOR) operation on each byte and the user-entered guess and saving the output.

The main() method disassembly reveals the registers containing the guess and mmap byte, which are stored in ESI and EDX, respectively.

|  |
| --- |
| 0x000055555555538d <+275>: mov -0x28(%rbp),%rcx  0x0000555555555391 <+279>: mov -0x30(%rbp),%rax  0x0000555555555395 <+283>: add %rcx,%rax  0x0000555555555398 <+286>: xor %esi,%edx  0x000055555555539a <+288>: mov %dl,(%rax)  0x000055555555539c <+290>: addq $0x1,-0x30(%rbp) |

Disassembly

Using a debugger, we can see the values in each register before the XOR operation, revealing the mmap data character by character. Initially, the mmap data appears to match the data stored in read\_flag.

|  |
| --- |
| (gdb) break \*0x0000555555555398  Breakpoint 2 at 0x555555555398  ...omitted for brevity...  (gdb) c  Continuing.  What's the key?  23  Breakpoint 2, 0x0000555555555398 in main ()  (gdb) info registers edx  edx 0x97 151  (gdb) info registers esi  esi 0x17 23  (gdb) c  Continuing.  Breakpoint 3, 0x000055555555539a in main ()  (gdb) info registers edx  edx 0x80 128  (gdb) c  Continuing.  Breakpoint 2, 0x0000555555555398 in main ()  (gdb) info registers edx  edx 0x46 70  (gdb) c  Continuing.  Breakpoint 3, 0x000055555555539a in main ()  (gdb) info registers edx  edx 0x51 81  (gdb) c  Continuing.  Breakpoint 2, 0x0000555555555398 in main ()  (gdb) info registers edx  edx 0x23 35  (gdb) c  Continuing.  Breakpoint 3, 0x000055555555539a in main ()  (gdb) info registers edx  edx 0x34 52  (gdb) q |

Disassembly

To get more information about the stored data, I wrote a script to extract every value from EDX.

|  |
| --- |
| def main():  # Start gdb session  p = process('/bin/bash')  p.sendline('gdb ./dora -q')  p.sendline('break \_start')  p.recv() # GDB response with one line indicating that the breakpoint is set  p.sendline('r')  p.sendline('break \*0x0000555555555398')  p.recv()  p.sendline('clear \_start')  p.recv()  p.sendline('c')  p.recvuntil(b'What\'s the key?')  p.sendline(b'23')  p.sendline('c')  data = []  for i in range(80):  p.recvuntil(b'Breakpoint 2')  p.recvline()  p.sendline('info registers edx')  c = cleanLine(p.recvline())  r = re.split("\s+", c)  data.append(r[2])  p.sendline('c')  print(data) |

Extraction Script

|  |
| --- |
| ['0x97', '0x46', '0x23', '0x34', '0x4d', '0x8a', '0xc4', '0x7e', '0x7c', '0x7c', '0x7c', '0x73', '0x79', '0x97', '0x47', '0x22', '0x34', '0xf5', '0xbb', '0xc6', '0x83', '0x7c', '0x7c', '0x7c', '0xc4', '0x7c', '0x7c', '0x7c', '0x7c', '0x73', '0x79', '0xc3', '0x7d', '0x7c', '0x7c', '0x7c', '0xc6', '0x83', '0x7c', '0x7c', '0x7c', '0xc4', '0x7d', '0x7c', '0x7c', '0x7c', '0x73', '0x79', '0xc3', '0x7c', '0x7c', '0x7c', '0x7c', '0xc4', '0x40', '0x7c', '0x7c', '0x7c', '0x73', '0x79', '0x94', '0xbd', '0x83', '0x83', '0x83', '0x1a', '0x10', '0x1d', '0x1b', '0x52', '0x8', '0x4', '0x8', '0x7c', '0x94', '0xbc', '0x83', '0x83', '0x83', '0x0'] |

Data

After extracting the data, I attempted to brute force the solution by looping through each possible guess and performing a XOR between the guess integer and each byte in the extracted data.

|  |
| --- |
| def bruteForceMagic():  for i in range(256):  data = bytes(c ^ i for c in test\_chars)  print(data)  return 0 |

Brute Force Script

Initially, the results appeared to be nonsense data, but after using *grep* to search for common terms, I discovered the string “flag.txt” in the script output.

|  |
| --- |
| python3 Dora\_BF.py | grep -i "flag"  b'\xcb\x1a\x7fh\x11\xd6\x98" /%\xcb\x1b~h\xa9\xe7\x9a\xdf \x98 /%\x9f! \x9a\xdf \x98! /%\x9f \x98\x1c /%\xc8\xe1\xdf\xdf\xdfFLAG\x0eTXT \xc8\xe0\xdf\xdf\xdf\\'  b'\xeb:\_H1\xf6\xb8\x02\x00\x00\x00\x0f\x05\xeb;^H\x89\xc7\xba\xff\x00\x00\x00\xb8\x00\x00\x00\x00\x0f\x05\xbf\x01\x00\x00\x00\xba\xff\x00\x00\x00\xb8\x01\x00\x00\x00\x0f\x05\xbf\x00\x00\x00\x00\xb8<\x00\x00\x00\x0f\x05\xe8\xc1\xff\xff\xffflag.txt\x00\xe8\xc0\xff\xff\xff|' |

Script Output

An update to the script revealed which input value resulted in the useable data.

|  |
| --- |
| def bruteForceMagic():  for i in range(256):  data = bytes(c ^ i for c in test\_chars)  if 'flag'.encode() in data:  return i  return 0  >> python3 Dora\_BF.py  >> 124 |

Brute Force Script

The full solver script for this challenge is available in [Appendix E](#_Appendix_E:_Dora_BF.py).

### Attempt

After discovering a possible input value using good old-fashioned brute force, I validated the guess against a local instance of Dora.

|  |
| --- |
| $ gdb ./dora  ...omitted for brevity...  (gdb) r  What's the key?  124  |[Inferior 1 (process 90978) exited normally]  (gdb) q |

Local Success

This value also worked for the remote instance, which revealed the flag.

|  |
| --- |
| nc offsec-chalbroker.osiris.cyber.nyu.edu 1250  What's the key?  124  flag{mmaped\_some\_fresh\_pages} |

Flag

# Appendix A: Student Information

|  |  |
| --- | --- |
| Lindsay Von Tish | |
| Email | [lmv9443@nyu.edu](mailto:lmv9443@nyu.edu) |

# Appendix B: Tools

|  |  |
| --- | --- |
| Name | URL |
| EDB | <https://www.kali.org/tools/edb-debugger/> |
| GDB | <https://www.gnu.org/software/gdb/gdb.html> |
| Ghidra | <https://ghidra-sre.org/> |
| Netcat | <https://netcat.sourceforge.net/> |
| PwnTools | <https://github.com/Gallopsled/pwntools> |

# 

# Appendix C: Postage

### Overview

|  |  |  |
| --- | --- | --- |
| Postage | | |
| **200 Points** | Flag Value | flag{i\_hope\_ur\_ready\_4\_some\_pwning\_in\_a\_few\_weeks} |
| Location | offsec-chalbroker.osiris.cyber.nyu.edu 1247 |

### Details

After downloading the postage binary, its execution results in a text prompt awaiting user input. The first execution resulted in a segmentation fault, shown in the following figure:

|  |
| --- |
| gdb ./postage  (gdb) r  Starting program: /home/kali/Desktop/1-Week/postage  Can you tell me where to mail this postage?  No  Program received signal SIGSEGV, Segmentation fault.  0x000000000040195e in main () |

Segmentation Fault

The program's main method reveals its base functionality, making it useful for discovering the source of the error. The code below was disassembled using *Ghidra*, and the variable names have been changed for clarity. After printing a message asking for user input, postage uses the get\_number function to save that number as a pointer value. Essentially, the user input is a memory address. Next, the program takes the data stored at that address and saves it in the val variable. Finally, the program compares val to the hardcoded value 0xd000dfaceee and prints either the flag or a "try again" message based on whether or not the values match.

|  |
| --- |
| bool main(EVP\_PKEY\_CTX \*param\_1)  {  long \*pointer;  long val;    init(param\_1);  puts("Can you tell me where to mail this postage?");  pointer = (long \*)get\_number();  val = \*pointer;  if (val != 0xd000dfaceee) {  puts("That doesn\'t look right... try again later, friend!");  }  else {  puts("Got it! That\'s the right number!");  print\_flag();  }  return val != 0xd000dfaceee;  } |

Main Method

Running postage with another debugger, such as *edb*, as shown in the following figure, reveals more information about the segmentation fault. The segfault occurred when the program attempted to access memory at the address 0x0000000.

A screenshot of a computer error

Description automatically generated  
Segmentation Fault Data

A memory address of 0 is outside of the program's memory space; attempting to read data from it results in a segmentation fault. Based on the postage main method, the error most likely occurred when the program attempted to save the data at the user-input address in the val variable.

The get\_number function, shown decompiled below, gives more insight into acceptable user input. The function uses fgets to save the user input as a string. Then it calls strtol, a C function that converts that user input string to a base ten long. If the string data cannot be converted, like if it has non-numerical ASCII characters, strtol will return 0.

|  |
| --- |
| void get\_number(void)  {  long in\_FS\_OFFSET;  char input [136];  long check;    check = \*(long \*)(in\_FS\_OFFSET + 0x28);  fgets(input,0x80,(FILE \*)stdin);  strtol(input,(char \*\*)0x0,10);  if (check != \*(long \*)(in\_FS\_OFFSET + 0x28)) {  /\* WARNING: Subroutine does not return \*/  \_\_stack\_chk\_fail();  }  return;  } |

Get\_number

Although get\_number appears to be a void function that does not return any data, it essentially returns the result of strtol. When a function runs, the RAX register holds its return data. When get\_number returns, the data returned by strtol remains in the RAX register, which, in turn, is saved as a pointer in the val variable. If the user enters a base-ten number, it will be stored in RAX as hexadecimal. Otherwise, RAX will equal 0, the strtol error code.

In the following example, the user entered the number 4200836. The value of RAX will change before and after the call to get\_number.

|  |  |
| --- | --- |
| A screenshot of a computer code  Description automatically generated RAX Before Call | A screenshot of a computer  Description automatically generated RAX After Call |

A screen shot of a computer

Description automatically generated  
Program Output

RAX holds a value of 0000000000401984 after get\_number runs. This number is the Hexadecimal notation of the user-entered decimal number, 4200836. The memory at 0x401984 is accessible to the program, so it runs without a segmentation error, as illustrated below:

Although the program ran without error, the value the user entered was not correct. To successfully complete the challenge, the player must enter a decimal number corresponding to a program-accessible memory address that stores the "secret" value 0xD00DFACEEE.

### Challenge Attempt

Completing the challenge requires the user to enter an address of memory that is not only accessible to the postage program but also contains specific data. An attacker has two options: finding a memory location containing the target data or bypassing the comparison entirely.

#### Bypass Comparison

In the decompiled main method, postage sets the variable pointer to the user-entered address using get\_number. This call is also visible at line 0x00401949 of the assembly code. Then, it saves the data stored at that address in the val variable before the if statement. These operations are performed by lines 0x0040194E through 0x0040195E of the assembly code.

|  |
| --- |
| puts("Can you tell me where to mail this postage?");  pointer = (long \*)get\_number();  val = \*pointer;  if (val != 0xd000dfaceee) {  puts("That doesn\'t look right... try again later, friend!");  }  else {  puts("Got it! That\'s the right number!");  print\_flag();  } |

Main Method

|  |
| --- |
| 00401949 e8 69 ff CALL get\_number  ff ff  0040194e 48 89 45 f0 MOV qword ptr [RBP + local\_18],RAX  00401952 48 8b 45 f0 MOV RAX,qword ptr [RBP + local\_18]  00401956 48 89 45 f8 MOV qword ptr [RBP + local\_10],RAX  0040195a 48 8b 45 f8 MOV RAX,qword ptr [RBP + local\_10]  0040195e 48 8b 00 MOV RAX,qword ptr [pointer]  **00401961 48 ba ee MOV RDX,0xd000dfaceee**  **ce fa 0d**  **00 0d 00 00**  **0040196b 48 39 d0 CMP RAX,RDX**  0040196e 75 20 **JNZ LAB\_00401990**  00401970 48 8d 05 LEA RAX,[s\_Got\_it!\_That's\_the\_right\_number!\_00 =  d1 67 09 00  00401977 48 89 c7 MOV RDI=>s\_Got\_it!\_That's\_the\_right\_number!\_  0040197a e8 e1 12 CALL puts int puts(char \* \_\_s)  01 00  0040197f b8 00 00 MOV RAX,0x0  00 00  00401984 e8 5c fe CALL print\_flag undefined print\_flag()  ff ff  00401989 b8 00 00 MOV RAX,0x0  00 00  0040198e eb 14 JMP LAB\_004019a4  **LAB\_00401990** XREF[1]: **0040196e(j)**  00401990 48 8d 05 LEA RAX,[s\_That\_doesn't\_look\_right...\_  d9 67 09 00 |

Main Method Assembly

Once the values are set, the MOV command at 0x00401961 places the hexadecimal data 0xD00DFACEEE in the RDX register in preparation for the comparison (CMP) at 0x0040196B. If the CMP operation returns True, the program will continue into 0x00401970 to print the success message before calling print\_flag at 0x00401984. Otherwise, it will jump to LAB\_00401990 and begin the "incorrect" response at 0x00401990.

By copying the data stored in RDX at 0x00401961 into RAX before the CMP at 0x0040196B, an attacker can get the "success" message without entering a correct answer. Using a debugger, they can set a breakpoint at 0x0040196B and copy the data from RDX into RAX before the CMP operation runs. The following example uses *GDB*:

The attacker must use an input value consistent with a decimal notation of the address space postage can access.

|  |
| --- |
| gdb ./postage  ...omitted for brevity...  Reading symbols from ./postage...  (No debugging symbols found in ./postage)  (gdb) break \_start  Breakpoint 1 at 0x4016c0  (gdb) r  Starting program: /home/kali/Desktop/1-Week/postage  Breakpoint 1, 0x00000000004016c0 in \_start ()  (gdb) disas main  Dump of assembler code for function main:  ...omitted for brevity...  0x000000000040195e <+63>: mov (%rax),%rax  0x0000000000401961 <+66>: movabs $0xd000dfaceee,%rdx  0x000000000040196b <+76>: cmp %rdx,%rax  0x000000000040196e <+79>: jne 0x401990 <main+113>  ...omitted for brevity...  End of assembler dump.  (gdb) break \*0x000000000040196b  Breakpoint 2 at 0x40196b  (gdb) c  Continuing.  Can you tell me where to mail this postage?  4200836  Breakpoint 2, 0x000000000040196b in main ()  (gdb) info registers rax  rax 0xb8fffffe5ce8 203409651031272  (gdb) info registers rdx  rdx 0xd000dfaceee 14293885701870  (gdb) set $rax = $rdx  (gdb) info registers rax  rax 0xd000dfaceee 14293885701870  (gdb) info registers rax  rax 0xd000dfaceee 14293885701870  (gdb) c  Continuing.  Got it! That's the right number!  ERROR: no flag found. |

Successful Bypass

Although this example successfully bypasses the program secret, it did not reveal the flag because the necessary debugging was done using a local copy of postage. To get the flag, the attacker must enter the correct value to attack the remote program.

#### The Right Answer

To get the flag, an attacker must input a memory address in decimal notation that holds the data 0xD00DFACEEE. As shown in the disassembled main method, postage does not store the secret string in a variable. The hardcoded value is only stored in RDX at line 0x00401961, right before the CMP at line 0x0040196B, as shown below:

|  |
| --- |
| 00401961 48 ba ee MOV RDX,0xd000dfaceee  ce fa 0d  00 0d 00 00  0040196b 48 39 d0 CMP RAX,RDX  0040196e 75 20 JNZ LAB\_00401990  00401970 48 8d 05 LEA RAX,[s\_Got\_it!\_That's\_the\_right\_number!\_00 =  d1 67 09 00 |

Secret Stored

The program itself stores the secret value. The line starts at 0x00401961, the first two bytes of data detail the operation, and then the secret value is stored at 0x00401963. The following table shows each byte in memory and the corresponding address.

|  |  |
| --- | --- |
| **Address** | **Value** |
| 0x00401961 | 48 |
| 0x00401962 | ba |
| 0x00401963 | ee |
| 0x00401964 | ce |
| 0x00401965 | fa |
| 0x00401966 | 0d |
| 0x00401967 | 00 |
| 0x00401968 | 0d |
| 0x00401969 | 00 |
| 0x0040196A | 00 |

Secret in Memory

The address where the secret data begins, 0x00401963, can be written as 4200803 in decimal notation. This is the correct address to enter, as shown below:

|  |
| --- |
| $ nc offsec-chalbroker.osiris.cyber.nyu.edu 1247  Can you tell me where to mail this postage?  4200803  Got it! That's the right number!  Here's your flag, friend: flag{i\_hope\_ur\_ready\_4\_some\_pwning\_in\_a\_few\_weeks} |

Su**ccess**

# Appendix D: BoD\_Remote.py

|  |
| --- |
| from pwn import \*  import re  # Array with data from forestsOfEwing  f0e = […omitted for brevity…]  # Host and port for the remote challenge  HOST = 'offsec-chalbroker.osiris.cyber.nyu.edu'  PORT = 8005  # A function to send a name to the remote challenge  # Input: Connection  # Output: Message  def question1(p):  msg = 'Juneau'  p.sendline(msg)  return msg  # A function to send a the answer to question 2  # Input: Connection  # Output: Array with both answers  def question2(p):  msg = '10'  p.sendline(msg.encode())  p.sendline(msg.encode())  ans = [msg, msg]  return ans  # A function to send a the answer to question 3  # Input: Connection  # Output: Array with all nine answers  def question3(p):  i = 0  index = [[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]]  for value in f0e:  if value == 0x1:  index[0][0] = i//256  index[0][1] = i%256  elif value == 0x2:  index[1][0] = i//256  index[1][1] = i%256  elif value == 0x3:  index[2][0] = i//256  index[2][1] = i%256  elif value == 0x4:  index[3][0] = i//256  index[3][1] = i%256  elif value == 0x5:  index[4][0] = i//256  index[4][1] = i%256  elif value == 0x6:  index[5][0] = i//256  index[5][1] = i%256  elif value == 0x7:  index[6][0] = i//256  index[6][1] = i%256  elif value == 0x8:  index[7][0] = i//256  index[7][1] = i%256  elif value == 0x9:  index[8][0] = i//256  index[8][1] = i%256  i +=1  i = 0  while i < 9:  msg = str(index[i][0]).encode()  p.sendline(msg)  msg = str(index[i][1]).encode()  p.sendline(msg)  i += 1  return index  def main():  p = remote(HOST, PORT)  p.recvuntil(b'What is your name?')  print("Answering Question 1")  print(question1(p))  p.recvuntil(b'What is your quest?')  print("Answering Question 2")  print(question2(p))  p.recvuntil(b'What is the air-speed velocity of an unladen swallow?')  print("Answering Question 3")  print(question3(p))  p.recvuntil(b'flag{')  print(p.recvline())  p.close()  return 0    if \_\_name\_\_ == "\_\_main\_\_":  main() |

# Appendix E: Dora\_BF.py

|  |
| --- |
| # Characters taken from the Dora memory map  test\_chars = bytes([...omitted for brevity...])  # A function to xor all 255 characters with every byte in the array to see if we get usable data  # Input: N/A  # Output: The value that creates data with the word "flag" in it  def bruteForceMagic():  for i in range(256):  data = bytes(c ^ i for c in test\_chars)  if 'flag'.encode() in data:  return i  return -13  print(bruteForceMagic()) |