Week 2 Challenge Writeup

Reverse Engineering Part 1

Lindsay Von Tish
lmv9443@nyu.edu
02/07/2024

Table of Contents

Challenge Details

Numerix

Overview

Are You Alive		
100 Points	Flag Value	flag{gl4d_you_d1dnt_n33d_to_p4rs3_w3ird_f0rmats_huh}
	Location	offsec-chalbroker.osiris.cyber.nyu.edu 1246

Details

The challenge begins with a prompt offering a binary download and a remote connection.

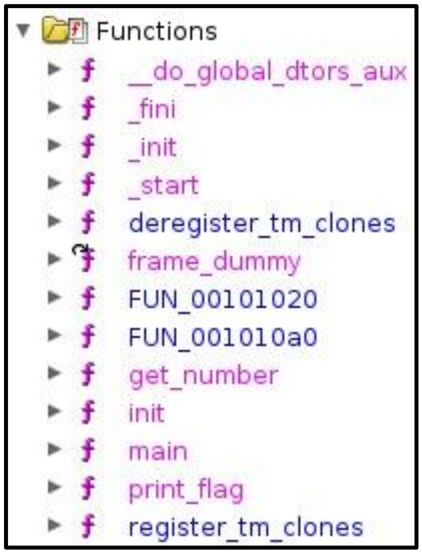
Challenge Prompt

A brief analysis with *strings* revealed some of the text used in the executable, but no sensitive information was stored.

```
flag.txt
Here's your flag, friend: %s
ERROR: no flag found. If you're getting this error on the remote system, please
message the admins. If you're seeing this locally, run it on the remote system! You
solved the challenge, and need to get the flag from there!
HEY!! I forgot my favorite numbers...
Can you get them from my diary?
What's my favoritest number?
No! No! No! That's not right!
What's my second most favorite number?
What? NO! Try again!!
Ok, you're pretty smart! What's the next one?
Ugh, ok, listen, you really need to hit the books...
YEAAAAAAAAAH you're doing GREAT! One more!
Darn, so close too...
Awwwwww yeah! You did it!
```

Binary Strings Example

After opening the binary in *Ghidra*, the player will discover three important functions in the program: `main`, `get_number`, and `print_flag`.



Functions

The program starts at `main`. The `main` method decompiled code is shown below. The variable names have been changed, and each corresponding if and else statement has been highlighted for clarity.

```
undefined8 main(EVP_PKEY_CTX *param_1)

{
  int Guess2;
  uint Guess3;
  long Guess1;
  undefined8 Success;

  init(param_1);
  puts("HEY!! I forgot my favorite numbers...");
  puts("Can you get them from my diary?");
  puts("What\'s my favoritest number?");
  Guess1 = get_number();
  if (Guess1 == 0xdeadbeef) {
    puts("What\'s my second most favorite number?");
    Guess2 = get_number();
    if (Guess2 == 0x539) {
      puts("Ok, you\'re pretty smart! What\'s the next one?");
      Guess1 = get_number();
      if (Guess1 == 0xc0def001337beef) {
        puts("YEAAAAAAAAAH you\'re doing GREAT! One more!");
        Guess3 = get_number();
        if ((Guess3 & 0xf0f0f0f0) == 0xd0d0f0c0) {
          puts("Awwwwww yeah! You did it!");
          print_flag();
          Success = 0;
        }
        else {
```

```
          puts("Darn, so close too...");
          Success = 1;
        }
      }
      else {
        puts("Ugh, ok, listen, you really need to hit the books...");
        Success = 1;
      }
    }
    else {
      puts("What? NO! Try again!!");
      Success = 1;
    }
  }
  else {
    puts("No! No! No! That\'s not right!");
    Success = 1;
  }
  return Success;
}
```

Main Method Decompiled Code

The program offers a greeting before using the `get_number` method to obtain a number entered by the user. It then compares the user-entered number to a hardcoded value using the `if` statement highlighted in yellow. If the values do not match, the program admonishes the player and sets the `Success` flag to `1`, indicating that the player has failed. If the values do match, the program accepts a new guess and compares it to a new hardcoded value, shown in the if and else statements highlighted in green. The program takes in another guess and makes one more comparison to a hardcoded value before asking for the last number.

For the last guess, the program performs a bitwise **AND** function between the guessed value and another hardcoded value. These values are hardcoded in red and blue, respectively. This portion of the main method is shown below:

```
if ((Guess3 & 0xf0f0f0f0) == 0xd0d0f0c0) {
  puts("Awwwwww yeah! You did it!");
  print_flag();
  Success = 0;
}
else {
  puts("Darn, so close too...");
  Success = 1;
}
```

Final Guess Code

If the result of the **AND** operation is equal to the hardcoded value highlighted above in purple, the player has successfully beaten the game. After a successful final guess, the program prints a happy message, calls the `print_flag` function, and sets the `Success` flag to `0`.

The `print_flag` function, shown below, is not complex. However, the function text reveals an additional challenge.

```
__stream = fopen("flag.txt","r")
  if (__stream == (FILE *)0x0) {
    puts(
        "ERROR: no flag found. If you\'re getting this error on the remote system,
please message th e admins. If you\'re seeing this locally, run it on the remote
system! You solved the challe nge, and need to get the flag from there!"
        );
  }
  else {
    fgets(local_98,0x80,__stream);
    printf("Here\'s your flag, friend: %s\n",local_98);
  }
```

Print Flag Decompiled Code

The function attempts to open `flag.txt`. If the file is not found, it prints a message directing the player to run the program on the remote system.

Challenge Attempt

Before attempting to connect to the remote system using the URL specified in the challenge prompt, the assessor attempted the game locally to verify their answers. Each correct guess, shown in the following table, was revealed in the program's main method.

Guess	Correct Answer
1	3735928559
2	1337
3	868613086753832000
4	3503354048

Correct Answers

The values of the first three guesses, `3735928559`, `1337`, and `868613086753832687`, are equal to the decimal value of each of the hexadecimal numbers in the first three if statements of the main method. The value of the final guess, `3503354048`, is the decimal value of a hexadecimal number that, when used in a bitwise `AND` operation with the hexadecimal value `0xf0f0f0f0`, will equal `0xd0d0f0c0`. There are multiple correct answers for the fourth question.

The following figure shows the results of a successful game played on a local system:

```
$ ./numerix
HEY!! I forgot my favorite numbers...
Can you get them from my diary?
What's my favoritest number?
3735928559
What's my second most favorite number?
1337
Ok, you're pretty smart! What's the next one?
868613086753832687
YEAAAAAAAAAH you're doing GREAT! One more!
```

```
3503354048
Awwwwww yeah! You did it!
ERROR: no flag found. If you're getting this error on the remote system, please
message the admins. If you're seeing this locally, run it on the remote system! You
solved the challenge, and need to get the flag from there!
```

Local Game

The player does not get the challenge flag when they win on a locally-ran game, which is consistent with the errors shown in the **`print_flag`** function. However, after connecting to the remote system, the assessor was able to get the challenge flag.

```
$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1246
HEY!! I forgot my favorite numbers...
Can you get them from my diary?
What's my favoritest number?
3735928559
What's my second most favorite number?
1337
Ok, you're pretty smart! What's the next one?
868613086753832687
YEAAAAAAAAAH you're doing GREAT! One more!
3503354048
Awwwwww yeah! You did it!
Here's your flag, friend: flag{gl4d_you_d1dnt_n33d_to_p4rs3_w3ird_f0rmats_huh}
```

Remote Game

An attacker with access to the game binary would be able to exploit the game and obtain the challenge flag. Decompiling the program would grant the attacker access to all of the information used to determine whether a guess was correct, allowing them to reverse-engineer the correct values to guess.

Strops

Overview

Are You Alive		
100 Points	Flag Value	flag{l00ps_and_x0rs_and_reads_o_my}
	Location	local

Details

The challenge prompt includes a binary file but not a remote location, implying that the flag can be obtained through local execution of the program. On the first run, the program asked the user to enter a flag before evaluating the input and exiting.

```
./strops.bin
Enter your flag: TheTorturedPoetsDepartment
Nope.
```

First Run

The strops `main` method, shown below after decompilation using *Ghidra*, takes in the user input and stores it in the `flagGuess` array. Then, the program iterates character by character through the guess to compare it to each character of the flag value. The loop will continue, and the counter will increase if the characters match. If the counter reaches thirty-five, `strops` prints "Correct!" and the program exits.

```
undefined8 main(void)
{
  ...omitted for brevity...
  printf("Enter your flag: ");
  read(1,flagGuess,0x40);
  counter = 0;
  do {
    if (0x22 < counter) {
      puts("Correct!");
LAB_001012c6:
      ...omitted for brevity...
      return 0;
    }
    if ((byte)~flag[(int)counter] != flagGuess[(int)counter]) {
      puts("Nope.");
      goto LAB_001012c6;
    }
    counter = counter + 1;
  } while( true );
}
```

Strops Main Method

The if statement in the main method controls most of the functionality important to an attacker. In the `main` method assembly code, as seen below, `strops` places each character from the flag in `EDX` and each character from `flagGuess` in `EAX` before comparing the values using the `CMP` operation at `0x000055555555528c`.

```
(gdb) disas main
Dump of assembler code for function main:
      ...omitted for brevity...
      0x0000555555555268 <+88>:    mov    -0x54(%rbp),%eax
      0x000055555555526b <+91>:    cltq
      0x000055555555526d <+93>:    lea    0x2dac(%rip),%rdx        #
0x555555558020 <flag>
      0x0000555555555274 <+100>:   movzbl (%rax,%rdx,1),%eax
      0x0000555555555278 <+104>:   movsbl %al,%eax
      0x000055555555527b <+107>:   not    %eax
      0x000055555555527d <+109>:   mov    %eax,%edx
      0x000055555555527f <+111>:   mov    -0x54(%rbp),%eax
      0x0000555555555282 <+114>:   cltq
      0x0000555555555284 <+116>:   movzbl -0x50(%rbp,%rax,1),%eax
      0x0000555555555289 <+121>:   movsbl %al,%eax
      0x000055555555528c <+124>:   cmp    %eax,%edx
      0x000055555555528e <+126>:   je     0x5555555552a6 <main+150>
       ...omitted for brevity...
   0x00005555555552da <+202>:   leave
   0x00005555555552db <+203>:   ret
```

Strops Main Method Assembler Code

To beat the challenge, an attacker can use a debugger to set `EAX` to the same value as `EDX` before each `CMP`.

Challenge Attempt

Manual Debugger

Using a debugger to manually set the value of `EAX` each time the program calls `CMP` at `0x000055555555528c` is the simplest way to get the challenge flag. The following example uses *gdb*.

```
gdb ./strops.bin
...omitted for brevity...
(gdb) disas main
Dump of assembler code for function main:
...omitted for brevity...
   0x0000555555555289 <+121>:   movsbl %al,%eax
   0x000055555555528c <+124>:   cmp    %eax,%edx
   0x000055555555528e <+126>:   je     0x5555555552a6 <main+150>
...omitted for brevity...
End of assembler dump.
(gdb) break *0x000055555555528c
Breakpoint 2 at 0x55555555528c
(gdb) c
Continuing.
Enter your flag:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaa
```

Setting Breakpoint to 0x55555555528c

Set the breakpoint to the location of the `CMP`; in this example, the breakpoint will be at `0x000055555555528c`. After the breakpoint is set, continue and enter a guess at the prompt. This example uses a string of "a" to make it clear which register holds characters from the guess when comparing the register information at each breakpoint. Register EAX holds the characters from the user-entered guess, while EDX holds the flag characters.

```
Breakpoint 2, 0x000055555555528c in main ()
(gdb) info registers edx
edx            0x66                102
(gdb) info registers eax
eax            0x61                97
(gdb) set $eax = $edx
(gdb) info registers edx
edx            0x66                102
(gdb) info registers eax
eax            0x66                102
(gdb) c
Continuing.

Breakpoint 2, 0x000055555555528c in main ()
(gdb) info registers edx
edx            0x6c                108
(gdb) info registers eax
eax            0x61                97
(gdb) set $eax = $edx
(gdb) info registers edx
edx            0x6c                108
(gdb) info registers eax
eax            0x6c                108
(gdb) c
Continuing.
...omitted for brevity...
Breakpoint 2, 0x000055555555528c in main ()
(gdb) info registers edx
edx            0x7d                125
(gdb) info registers eax
eax            0x61                97
(gdb) set $eax = $edx
(gdb) info registers edx
edx            0x7d                125
(gdb) info registers eax
eax            0x7d                125
(gdb) c
Continuing.
Correct!
[Inferior 1 (process 2849417) exited normally]
(gdb) q
```

EAX and EDX Values at Breakpoints

Setting the value of EAX to EDX ensures that the program passes the comparison check before continuing to compare the next characters, where it will stop at the breakpoint again. After thirty-six iterations, `strops` will output "Correct!" and close.

Break	Value	Break	Value
1	0x66	19	0x73
2	0x6c	20	0x5f
3	0x61	21	0x61
4	0x67	22	0x6e
5	0x7b	23	0x64
6	0x6c	24	0x5f
7	0x30	25	0x72
8	0x30	26	0x65
9	0x70	27	0x61
10	0x73	28	0x64
11	0x5f	29	0x73
12	0x61	30	0x5f
13	0x6e	31	0x6f
14	0x64	32	0x5f
15	0x5f	33	0x6d
16	0x78	34	0x79
17	0x30	35	0x7d
18	0x72		

Flag Hex

Each hex value in the **RDX** register is one ASCII character of the flag. Decoding those values will reveal the flag.

While this method is one way to get the flag, it is rather tedious and can be automated.

Automated Attack: stropsploit.py

The following example is one attempt to automate the **strops** exploitation using Python and the *PwnTools* library. The code shown below and in Appendix C has been edited for readability, but the entire script, titled **stropsploit.p**y, is included in the materials accompanying this document.

The **stropsploit main** method contains most of the script's functionality. The script runs **strops** using *gdb* and finds the **CMP** operation address before setting a breakpoint. It then sends debugger commands every time **strop** reaches the breakpoint to retrieve the flag character by character.

```
def main():
       # Start gdb session, set breakpoint at start, and then run strops
       p =  process("/bin/bash")
       p.sendline("gdb ./strops.bin -q")
       p.sendline("break _start")
       p.sendline("r")

       # Find location of cmp
       loc =findCMP(p)

       # Set breakpoint at cmp location and delete breakpoint at _start
       cmd = "break *" + loc
       p.sendline(cmd)
```

```
        p.sendline("clear _start")

        # Interact with strops and save debugger output
        getFlag(p)
        # Parse the flag from the log file
        print(parseFlag())
```

Stropsploit Main Method

After initiating the *gdb* session, `stropsploit` calls the `findCMP` function.

```
def findCMP(p):
        m = open("mainDisas.txt", "a")
        m.write("Main Method Disasembly:" + "\n")
        p.sendline("disas main")

        n = 0
        while True:
                ln = cleanLine(p.recvline())
                m.write(ln)
                if re.search("End of assembler dump.", ln):
                        break
                elif re.search("cmp.*eax.*edx", ln):
                        cline = ln
                elif(n == 20):
                        # Must page through disassembly for some reason
                        p.sendline("c")
                n+=1

        m.write("Found the memory location: [")
        c = re.split("\s+", cline)
        m.write(c[1])
        m.write("]")
        return c[1]
```

Stropsploit findCMP Method

The `findCMP` function uses *gdb* to get a disassembly of the `strops main` method. It loops through the disassembly line by line, writing them to an external file before checking to see if the current line matches the target `CMP` operation. The matching line is written to the cline variable, which `findCMP` parses before returning the address.

Next, the `main` method sets a breakpoint at the address and clears any other breakpoints before attempting to get the flag.

```
# Set breakpoint at cmp location and delete breakpoint at _start
cmd = "break *" + loc
p.sendline(cmd)
p.sendline("clear _start")

# Interact with strops and save debugger output
getFlag(p)
# Parse the flag from the log file
print(parseFlag())
```

Stropsploit Main Method After findCMP

The **getFlag** method handles the repeated debugger interaction required to retrieve each character of the flag. It sends strops the user input guess value, then lets the program run until the breakpoint. Every time **strops** hits the breakpoint, **getFlag** sends two *gdb* commands: The first copies the value stored in RDX to RAX, and the second gets the current state of RAX. The script saves all of the debugger output to the log.

```
def getFlag(p):
	log = open("Strop.txt", "a")
	p.sendline("c")

	# Wait for the enter flag prompt and send a guess
	while True:
		r = cleanLine(p.recvline())
		if re.search("Enter your flag:", r):
			guess =
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
			p.sendline(guess.encode())
			break

	# Loops through as strops reaches the breakpoint at CMP
	for i in range(40):
		# Save debugger response in log
		r = cleanLine(p.recv())
		log.write(r)
		# Save EDX value in EAX value then write EAX information to log
		p.sendline("set $eax = $edx")
		p.sendline("info registers eax")
		r = cleanLine(p.recv())
		log.write(r)

		# Break once we get "correct" response
		if re.search("Correct", r):
			break
		# Send debugger continue command
		p.sendline("c")
	log.close()
	return 0
```

Stropsploit getFlag Method

The loop runs until it passes the expected length of the flag, but it breaks if **strops** returns the message "Correct!" The interaction loop allows the script to essentially spam the *gdb* commands “**set $eax = $edx**” and "**info registers eax**" every time **strops** hits the breakpoint. The **getFlag** function does not return useful data. After it returns, the **main** method calls **parseFlag** to retrieve the flag from the **getFlag** log data.

```
def parseFlag():
	log = open("Strop.txt", "r")
	f = ""
	i = 0
	for line in log:
		if re.search("eax.*0x.*", line):
			l = re.split("\s+", line)
			n = re.split("x", l[3])
			f += n[1]
	return bytes.fromhex(f).decode('ascii')
```

Stropsploit parseFlag Method

`Parseflag` opens the text file containing the debugger and iterates through each line. If the line's format is consistent with that of the `EAX` information *gdb* output, the function splits out the stored hexadecimal value and saves it to a string. After iterating through every line in the file, `parseFlag` returns the ASCII text of the hex values, which should contain the challenge flag.

```
python3 stropsploit.py
[+] Starting local process '/bin/bash': pid 2895274
...omitted for brevity...
@@flag{l00ps_and_x0rs_and_reads_o_my}
[*] Stopped process '/bin/bash' (pid 2895274)
```

Successful Flag Retrieval Using Stropsploit

The `stropsploit` code is still under testing and has some known errors. In some cases, the script will stop early or cause strops to crash, as shown in the following example:

```
python3 stropsploit.py
[+] Starting local process '/bin/bash': pid 2895204
/home/kali/Desktop/1-Week/stropsploit.py:107: BytesWarning: Text is not bytes;
assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline("gdb ./strops.bin -q")
...omitted for brevity...
@@flag{l00
[*] Stopped process '/bin/bash' (pid 2895204)
```

Partial Flag Retrieval

Additionally, the *PwnTools* `sendline` command `stropsploit` uses to communicate with the debugger causes warnings when the script runs.

The entire `stropsploit` code is available in Appendix C, where it has been edited for readability, and the last known functioning version of the code is saved as `stropsploit.py`.

Feedback is welcome. Please run at your own risk.

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 ()
```

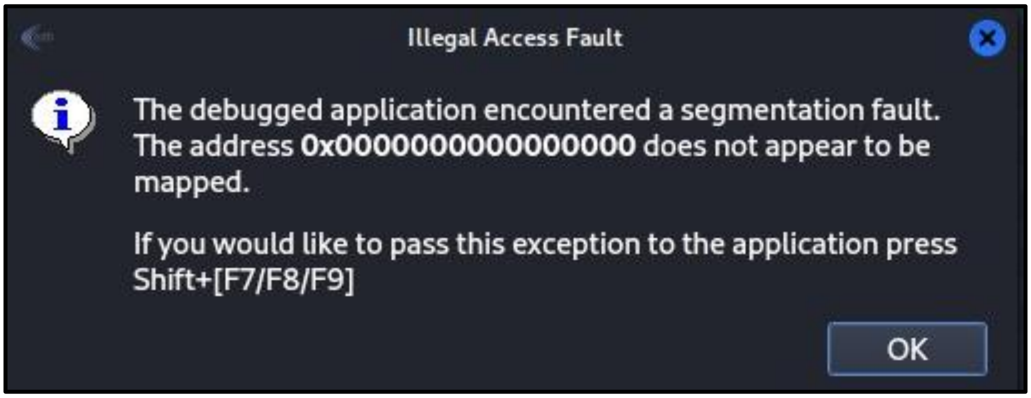
Segmentation Fault

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**.



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.

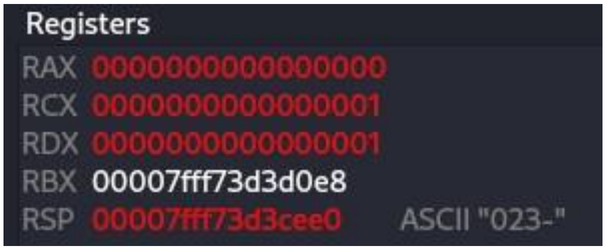
The **get_number** function, shown decompiled below, gives more insight into acceptable user input. The function uses **fgets** to save the user input as a string. Then it calls **strtol**, a C function that converts that user input string to a base ten long. If the string data cannot be converted, like 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`.



RAX Before Call

RAX After Call

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}
```

Success

This technique allows an attacker to bypass both local and remote versions of `postage` successfully.

Appendix A: Student Information

Lindsay Von Tish	
Email	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: Stropsploit.py

Python code used to automate the solution for Strops.

```
from pwn import *
import re

#################################################################
#      stropsploit.py                                           #
#      Lindsay Von Tish (lmv9443@nyu.edu)                       #
#      Reverse Engineering 1: Strops Challenge Solver Script    #
#      02/07/2024                                               #
#################################################################

# A function to send a line and receive the response
#      Input: Message String, Connection
#      Output: Recieved message
def sendRecv(msg, dst):
       dst.sendline()
       r = dst.recv()
       return r

# A function to convert encoded input to a string and remove text format characters
#      Input: Encoded string
#      Output: Unencoded string
def cleanLine(ln):
       ansi_escape = re.compile(r'\x1B(?:[@-Z\\-_]|\[[0-?]*[ -/]*[@-~])')
       l = ansi_escape.sub('', str(ln, encoding='utf-8'))
       return l

# A function to find the memory location of the CMP operation
#      Input: Connection
#      Output: Memory location in hex string

def findCMP(p):
       m = open("mainDisas.txt", "a")
       m.write("Main Method Disasembly:" + "\n")
```

```
	p.sendline("disas main")

	n = 0
	while True:
		ln = cleanLine(p.recvline())
		m.write(ln)
		if re.search("End of assembler dump.", ln):
			break
		elif re.search("cmp.*eax.*edx", ln):
			cline = ln
		elif(n == 20):
			# Must page through disassembly for some reason
			p.sendline("c")
		n+=1

	m.write("Found the memory location: [")
	c = re.split("\s+", cline)
	m.write(c[1])
	m.write("]")
	return c[1]

# A function to iterate through interactions with the strops binary
#	Sends a guess to the program
#	Waits until strops reaches the set breakpoint
#		Sends debug command to set the value of EAX to that of EDX
#		Saves current state of EAX register
#	Input: Connection
#	Output: None
def getFlag(p):
	log = open("Strop.txt", "a")
	p.sendline("c")

	# Wait for the enter flag prompt and send a guess
	while True:
		r = cleanLine(p.recvline())
		if re.search("Enter your flag:", r):
			guess =
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
			p.sendline(guess.encode())
			break

	# Loops through as strops reaches the breakpoint at CMP
	for i in range(40):
		# Save debugger response in log
		r = cleanLine(p.recv())
		log.write(r)
		# Save EDX value in EAX value then write EAX information to log
		p.sendline("set $eax = $edx")
		p.sendline("info registers eax")
		r = cleanLine(p.recv())
		log.write(r)
```

```
            # Break once we get "correct" response
            if re.search("Correct", r):
                  break
            # Send debugger continue command
            p.sendline("c")
      log.close()
      return 0

# A function to retreive the flag data from the log file
#      Input: None
#      Output: Decoded Flag
def parseFlag():
      log = open("Strop.txt", "r")
      f = ""
      i = 0
      for line in log:
            if re.search("eax.*0x.*", line):
                  l = re.split("\s+", line)
                  n = re.split("x", l[3])
                  f += n[1]
      return bytes.fromhex(f).decode('ascii')

def main():
      # Start gdb session
      p =  process("/bin/bash")
      p.sendline("gdb ./strops.bin -q")
      p.recv()
      p.sendline("break _start")
      p.recv() # GDB response with one line indicating that the breakpoint is set
      p.sendline("r")
      print(p.recv())

      # Find location of cmp
      loc =findCMP(p)
      # Set breakpoint at cmp location and delete breakpoint at _start
      cmd = "break *" + loc
      #print(cmd)
      p.sendline(cmd)
      print(p.recv())
      p.sendline("clear _start")
      print(p.recv)

      # Interact with strops and save debugger output
      getFlag(p)
      # Parse the flag from the log file
      print(parseFlag())

if __name__=="__main__":
      main()
```