# ROPME Buffer Overflow Attack Analysis Report

1. Vulnerability Analysis

1) Identified Vulnerability

The program contains a stack buffer overflow vulnerability in the `func()` function:

void func(){

char overflowme[32];

read(0, overflowme, 0x200);

}

The vulnerability exists because:

- The buffer size is only 32 bytes

- The read function can accept up to 0x200 (512) bytes

- No bounds checking is performed

2) How to Fix

To fix this vulnerability, several approaches could be taken:

1. Use bounded input functions like `fgets()` instead of `read()`

2. Add buffer size checking

3. Enable stack canaries

4. Implement input validation

2. Finding Return Address Offset

The offset to the return address was determined through the following stack layout analysis:

- Buffer size: 32 bytes

- Saved RBP: 8 bytes

- Return address: 8 bytes

- Total offset: 40 bytes (32 + 8)

This was verified through controlled buffer overflow testing using pattern generation and crash analysis.

3. Finding Libc Base Address

The libc base address can be calculated using the leaked setvbuf address:

1. Program provides setvbuf address: `printf("The address of setvbuf : %16p\n", setvbuf);`

2. Calculate base address: `libc\_base = setvbuf\_addr - setvbuf\_offset`

3. Verification through objdump and runtime analysis confirms the calculation

4. ROP Chain Construction

1) Gadget Analysis

Required gadgets for the exploit:

- `pop rdi ; ret` at 0x4012a3 (for function argument setup)

- `ret` gadget for stack alignment

2) ROP Chain Workflow

1. Overflow buffer with 40 bytes of padding

2. Use ret gadget for stack alignment

3. Use pop rdi gadget to load /bin/sh string address

4. Call system() with /bin/sh

5. Clean exit using exit()

The complete chain:

```python

payload = b'A' \* 40 # Buffer overflow padding

payload += p64(RET) # Stack alignment

payload += p64(POP\_RDI) # pop rdi ; ret

payload += p64(binsh\_addr) # /bin/sh address

payload += p64(system\_addr) # system() call

payload += p64(POP\_RDI) # pop rdi ; ret

payload += p64(0) # exit status

payload += p64(exit\_addr) # exit() call

```

5. Clean Program Termination

The exploit ensures clean program termination by:

1. Properly aligning the stack before system() call

2. Using exit() instead of letting the program crash

3. Maintaining proper stack frame integrity

This approach prevents crashes and ensures the program exits gracefully after shell access is obtained.

6. Testing and Verification

The exploit was tested in a controlled environment:

1. Initial offset verification

2. Address leak confirmation

3. ROP chain execution testing

4. Shell access verification

5. Clean exit confirmation

All test cases demonstrated successful exploitation while maintaining system stability.

7. Security Implications

This vulnerability demonstrates the importance of:

1. Proper input validation

2. Buffer size management

3. Modern security protections (NX, ASLR, canaries)

4. Secure coding practices

8. Conclusion

The successful exploitation of this vulnerability shows the critical nature of buffer overflow protections. While this was an educational exercise, in real-world applications such vulnerabilities could lead to serious security breaches. Proper security measures and coding practices are essential to prevent such vulnerabilities.