Week1-Refresher Write-ups

Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 350/300 pts solved !!!

GDB 0 (50 pts)

```
❯ nc offsec-chalbroker.osiris.cyber.nyu.edu 1241
......
     Your mission, should you choose to accept it,
     is to step through the code, find the place
     where the password is hiding and use that
     password to rescue the flag.
......
         ------   Welcome to GDB 0   ------


     This time you will have access to the source code!
......
    HEEEELP! My password is somewhere around here, but I can't find it.
    Can you tell me my password?
    >
......
```

In this challenge, utilizing GDB, we must find the hidden password to rescue the flag. This time we have access to the source code.

Firstly, we set a breakpoint at the entry to the `main` function using `b main` or `breakpoint main` before running. After that, we run the binary with `r` or `run` to hit the breakpoint.

```
pwndbg> b main
Breakpoint 1 at 0x562e6667e245: file gdb0.c, line 19.
pwndbg> r
Starting program: /home/ctf/gdb0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7ffd544cc878) at gdb0.c:19
19      set_buffering_mode();
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
......
─────────────────────────────────[ SOURCE (CODE) ]─────────────────────────────────
In file: /home/ctf/gdb0.c:19
   15 int main(int argc, char** argv) {
   16     char flag[0x80];
   17     char buffer[0x20];
   18
 ▶ 19     set_buffering_mode();
   20     puts("\n\n\tHEEEELP! My password is somewhere around here, but I can't find it.");
   21     puts("\tCan you tell me my password?");
   22
   23     printf("\t> ");
──────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────
......
```

We can observe from the `SOURCE (CODE)` part that the breakpoint is hit at line 19 (source line).

Then, we use `n` or `next` to go to the next instruction (but not dive into any function) until we reach the line where the hidden password is loaded into a register.

```
......
pwndbg> n
27      if (strcmp(buffer, get_password()) == 0) {
......
──────────────────────────────[ SOURCE (CODE) ]──────────────────────────────
In file: /home/ctf/gdb0.c:27
   23     printf("\t> ");
   24     fgets(buffer, sizeof(buffer), stdin);
   25     buffer[strcspn(buffer, "\n")] = '\0';
   26
 ► 27     if (strcmp(buffer, get_password()) == 0) {
   28         puts("\tYou did it! You found my password!");
   29         get_flag(flag);
   30         printf("\nHere's your flag, friend: %s\n\n\n", flag);
──────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────
......
pwndbg> n
35      puts("\n\tThat's nice, but it doesn't look like my password!\n\tTry again friend!\n\n");
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
─────────────[ REGISTERS / show-flags off / show-compact-regs off ]─────────────
*RAX  0xffffffcc
 RBX  0
 RCX  0
*RDX  0x34
 RDI  0x7ffd544cc6c0 ◂— 0
*RSI  0x560dbec56010 (password) ◂— '4_v3ry_1337_s3cr3t_p4ssw0rd'
 ......
 ──────────────────────────────[ SOURCE (CODE) ]──────────────────────────────
In file: /home/ctf/gdb0.c:35
   31
   32         return EXIT_SUCCESS;
   33     }
   34
 ► 35     puts("\n\tThat's nice, but it doesn't look like my password!\n\tTry again friend!\n\n");
   36     return EXIT_FAILURE;
   37 }
   38
   39
──────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────
......
```

We can observe from the `REGISTERS` part that after executing line 27 in the source code, by calling the `strcmp(buffer, get_password())`, the password `4_v3ry_1337_s3cr3t_p4ssw0rd` is loaded into the memory by the `get_password` and passed to the `strcmp` function as an argument in the register `RSI`.

Finally, we use `c` or `continue` to continue and finish normal execution, `delete` to delete all breakpoints, and `r` or `run` to run the binary again, but this time using the found password as the input to get the flag.

The captured flag is `flag{34sy_3n0ugh_wh3n_y0u_g3t_d3bug_symb0ls!_a1f7d5b5f08e4bba}`.

GDB 1 (50 pts)

```
❯ nc offsec-chalbroker.osiris.cyber.nyu.edu 1242
......
     Your mission, should you choose to accept it,
     is to step through the code, find the place where
     the flag is hiding and input the address to get
     the flag.
......
          ------   Welcome to GDB 1   ------
......
pwndbg> r
Starting program: /home/ctf/gdb1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
    What is the address of the buffer the flag is read into?
    (hint: it is zeroed out at the beginning of the function)
    >
......
```

In this challenge, utilizing GDB, we must step through the code, find the address of the buffer the flag is read into, and input the address to get the flag.

We use `disass main` or `disassemble main` to display the assembly code of the `main` function.

```
pwndbg> disass main
Dump of assembler code for function main:
   0x00000000000012a9 <+0>: endbr64
   0x00000000000012ad <+4>: push   rbp
   0x00000000000012ae <+5>: mov    rbp,rsp
   0x00000000000012b1 <+8>: add    rsp,0xffffffffffffff80
   0x00000000000012b5 <+12>:    mov    DWORD PTR [rbp-0x74],edi
   0x00000000000012b8 <+15>:    mov    QWORD PTR [rbp-0x80],rsi
   0x00000000000012bc <+19>:    mov    rax,QWORD PTR fs:0x28
   0x00000000000012c5 <+28>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000000012c9 <+32>:    xor    eax,eax
   0x00000000000012cb <+34>:    lea    rax,[rbp-0x60]
   0x00000000000012cf <+38>:    mov    edx,0x50
   0x00000000000012d4 <+43>:    mov    esi,0x0
   0x00000000000012d9 <+48>:    mov    rdi,rax
   0x00000000000012dc <+51>:    call   0x1140 <memset@plt>
   0x00000000000012e1 <+56>:    mov    eax,0x0
   0x00000000000012e6 <+61>:    call   0x14ce <set_buffering_mode>
   0x00000000000012eb <+66>:    lea    rax,[rip+0xd16]        # 0x2008
   0x00000000000012f2 <+73>:    mov    rdi,rax
   0x00000000000012f5 <+76>:    call   0x1100 <puts@plt>
   0x00000000000012fa <+81>:    lea    rax,[rip+0xd47]        # 0x2048
   0x0000000000001301 <+88>:    mov    rdi,rax
   0x0000000000001304 <+91>:    call   0x1100 <puts@plt>
   0x0000000000001309 <+96>:    lea    rax,[rip+0xd73]        # 0x2083
   0x0000000000001310 <+103>:   mov    rdi,rax
   0x0000000000001313 <+106>:   mov    eax,0x0
   0x0000000000001318 <+111>:   call   0x1130 <printf@plt>
   0x000000000000131d <+116>:   mov    eax,0x0
   0x0000000000001322 <+121>:   call   0x1381 <read_input>
   0x0000000000001327 <+126>:   mov    QWORD PTR [rbp-0x68],rax
   0x000000000000132b <+130>:   lea    rax,[rbp-0x60]
   0x000000000000132f <+134>:   cmp    QWORD PTR [rbp-0x68],rax
   0x0000000000001333 <+138>:   jne    0x1357 <main+174>
   0x0000000000001335 <+140>:   lea    rax,[rip+0xd4b]        # 0x2087
   0x000000000000133c <+147>:   mov    rdi,rax
   0x000000000000133f <+150>:   call   0x1100 <puts@plt>
   0x0000000000001344 <+155>:   lea    rax,[rbp-0x60]
   0x0000000000001348 <+159>:   mov    rdi,rax
   0x000000000000134b <+162>:   call   0x1443 <get_flag>
   0x0000000000001350 <+167>:   mov    eax,0x0
   0x0000000000001355 <+172>:   jmp    0x136b <main+194>
```

```
   0x0000000000001357 <+174>:   lea    rax,[rip+0xd4a]        # 0x20a8
   0x000000000000135e <+181>:   mov    rdi,rax
   0x0000000000001361 <+184>:   call   0x1100 <puts@plt>
   0x0000000000001366 <+189>:   mov    eax,0x1
   0x000000000000136b <+194>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x000000000000136f <+198>:   sub    rdx,QWORD PTR fs:0x28
   0x0000000000001378 <+207>:   je     0x137f <main+214>
   0x000000000000137a <+209>:   call   0x1120 <__stack_chk_fail@plt>
   0x000000000000137f <+214>:   leave
   0x0000000000001380 <+215>:   ret
End of assembler dump.
```

Here's what we can discover:

- At line `0x0000000000001322 <+121>`, the instruction calls the `read_input` function to take the input (expect the hexadecimal address of the buffer the flag is read into) from the user and return the result, which is stored in the `rax` register.
- At line `0x0000000000001327 <+126>`, the content of the `rax` register is moved to memory at `[rbp-0x68]`.
- At line `0x000000000000132b <+130>`, the instruction loads the effective address of the buffer `[rbp-0x60]` into the `rax` register. This buffer was set up and zeroed out from line `0x00000000000012cb <+34>` to `0x00000000000012dc <+51>` by calling the `memset` function.
- At line `0x000000000000132f <+134>`, the instruction compares the value stored at `[rbp-0x68]` (the result from read_input) with the address of the local buffer at `[rbp-0x60]`.
- At line `0x0000000000001333 <+138>`, if the value in `[rbp-0x68]` (the result of `read_input`) is not equal to the buffer's address at `[rbp-0x60]`, this instruction jumps to the code at `0x0000000000001357 <+174>`, meaning that the program displays a failing condition instead of proceeding correctly.

Therefore, through our analysis, we can conclude that the address of the buffer `[rbp-0x60]` is where the flag is read into because there is a comparison between it and the user's input. If they match in the comparison at line `0x000000000000132f <+134>`, the program will execute normally to call the `get_flag` function and load the flag into the memory.

Here's how we can verify our conclusion:

- At line `0x0000000000001344 <+155>`, the instruction loads the address of the local buffer located at `[rbp-0x60]` into the `rax` register.
- At line `0x0000000000001348 <+159>`, the instruction moves the content of the `rax` register (which now contains the address of the buffer `[rbp-0x60]`) into the `rdi` register, which is used to pass the first argument to the next called function.
- At line `0x000000000000134b <+162>`, the instruction calls the `get_flag` function with its first argument via the `rdi` register. According to the disassembly of the `get_flag` function (not presented here), the function opens the file and loads the flag into the buffer `[rbp-0x60]`.

To sum up, to get the flag, what we need to do is:

- Use `b *(main+121)` or `breakpoint *(main+121)` to set a breakpoint where the `read_input` function is called
- Use `b *(main+162)` or `breakpoint *(main+162)` to set a breakpoint where the `get_flag` function is called
- Use `r` or `run` to run the binary
- After hitting the first breakpoint at `main+121`, use `p/x $rbp - 0x60` or `print/x $rbp - 0x60` to print the address at buffer `rbp-0x60`, which is the address of the buffer the flag is read into.
- Use `n` or `next` to go to the next instruction (call the `read_input` function) and input the address we just found (in this case is `0x7fffe9d88d50`)
- Use `c` or `continue` to continue normal execution and hit the second breakpoint at `main+162`
- Use `n` or `next` to go to the next instruction (calling the `get_flag` function)
- Use `x/s $rbp - 0x60` to display the string (the flag we are looking for) stored at the memory address located at buffer `[rbp-0x60]`

```
pwndbg> b *(main+121)
```

```
Breakpoint 1 at 0x1322
pwndbg> b *(main+162)
Breakpoint 2 at 0x134b
pwndbg> r
Starting program: /home/ctf/gdb1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
    What is the address of the buffer the flag is read into?
    (hint: it is zeroed out at the beginning of the function)
    >
Breakpoint 1, 0x000055938f2e6322 in main ()
......
─────────────────────[ DISASM / x86-64 / set emulate on ]─────────────────────
 ► 0x55938f2e6322 <main+121>    call   read_input                  <read_input>
        rdi: 0x7fffe9d86af0 —▸ 0x7fb0e038a050 (funlockfile) ◂— endbr64
        rsi: 0x7fffe9d86c10 ◂— 0x203e09 /* '\t> ' */
        rdx: 0
        rcx: 0x7fb0e043c887 (write+23) ◂— cmp rax, -0x1000 /* 'H=' */
......
pwndbg> p/x $rbp - 0x60
$1 = 0x7fffe9d88d50
pwndbg> n
0x7fffe9d88d50
0x000055938f2e6327 in main ()
......
─────────────────────[ DISASM / x86-64 / set emulate on ]─────────────────────
   0x55938f2e6322 <main+121>    call   read_input                  <read_input>

 ► 0x55938f2e6327 <main+126>    mov    qword ptr [rbp - 0x68], rax     [0x7fffe9d88d48] => 0x7fffe9d88d50 ◂—
0
......
pwndbg> c
Continuing.
    That's the right address!

Breakpoint 2, 0x000055938f2e634b in main ()
......
 ► 0x55938f2e634b <main+162>    call   get_flag                    <get_flag>
        rdi: 0x7fffe9d88d50 ◂— 0
        rsi: 1
        rdx: 1
        rcx: 0x7fb0e043c887 (write+23) ◂— cmp rax, -0x1000 /* 'H=' */
......
pwndbg> n
0x000055938f2e6350 in main ()
......
──────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────
......
*RDX  0x7fffe9d88d50 ◂— 'flag{s331ng_wh4t_is_g01ng_0n_1ns1d3_4_pr0gr4m_1s_s00_1337!_96eca87b73a84b03}'
 RDI  0x7fffe9d88d50 ◂— 'flag{s331ng_wh4t_is_g01ng_0n_1ns1d3_4_pr0gr4m_1s_s00_1337!_96eca87b73a84b03}'
......
─────────────────────[ DISASM / x86-64 / set emulate on ]─────────────────────
......
   0x55938f2e634b <main+162>    call   get_flag                    <get_flag>

 ► 0x55938f2e6350 <main+167>    mov    eax, 0                          EAX => 0
......
pwndbg> x/s $rbp - 0x60
0x7fffe9d88d50: "flag{s331ng_wh4t_is_g01ng_0n_1ns1d3_4_pr0gr4m_1s_s00_1337!_96eca87b73a84b03}"
```

The captured flag is `flag{s331ng_wh4t_is_g01ng_0n_1ns1d3_4_pr0gr4m_1s_s00_1337!_96eca87b73a84b03}`.

GDB 2 (100 pts)

```
❯ nc offsec-chalbroker.osiris.cyber.nyu.edu 1243
......
     Your mission, should you choose to accept it,
     is to step through the code, find the place
     where the flag is stored and read the flag.
......
         ------   Welcome to GDB 2   ------
......
pwndbg> r
Starting program: /home/ctf/gdb2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".


        The flag is somewhere around here!
        Can you find it?
......
```

In this challenge, utilizing GDB, we must step through the code, find where the flag is stored, and read the flag.

We use `disass main` or `disassemble main` to display the assembly code of the `main` function.

```
pwndbg> disass main
Dump of assembler code for function main:
   0x0000000000001209 <+0>: endbr64
   0x000000000000120d <+4>: push   rbp
   0x000000000000120e <+5>: mov    rbp,rsp
   0x0000000000001211 <+8>: sub    rsp,0x10
   0x0000000000001215 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000000000001218 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x000000000000121c <+19>:    mov    eax,0x0
   0x0000000000001221 <+24>:    call   0x12c7 <set_buffering_mode>
   0x0000000000001226 <+29>:    mov    eax,0x0
   0x000000000000122b <+34>:    call   0x1250 <read_file>
   0x0000000000001230 <+39>:    lea    rax,[rip+0xdd1]        # 0x2008
   0x0000000000001237 <+46>:    mov    rdi,rax
   0x000000000000123a <+49>:    call   0x10b0 <puts@plt>
   0x000000000000123f <+54>:    mov    edi,0x3
   0x0000000000001244 <+59>:    call   0x1110 <sleep@plt>
   0x0000000000001249 <+64>:    mov    eax,0x0
   0x000000000000124e <+69>:    leave
   0x000000000000124f <+70>:    ret
End of assembler dump.
```

In the disassembly of the `main` function, the only information that might help is the instruction at line `0x000000000000122b <+34>`, calling the `read_file` function, which may contain something related to reading a flag from a file.

Thus, we use `disass read_file` or `disassemble read_file` to display the assembly code of the `read_file` function.

```
pwndbg> disass read_file
Dump of assembler code for function read_file:
   0x0000000000001250 <+0>: endbr64
   0x0000000000001254 <+4>: push   rbp
   0x0000000000001255 <+5>: mov    rbp,rsp
   0x0000000000001258 <+8>: sub    rsp,0x10
   0x000000000000125c <+12>:    mov    esi,0x0
   0x0000000000001261 <+17>:    lea    rax,[rip+0xddb]        # 0x2043
   0x0000000000001268 <+24>:    mov    rdi,rax
   0x000000000000126b <+27>:    mov    eax,0x0
   0x0000000000001270 <+32>:    call   0x10f0 <open@plt>
   0x0000000000001275 <+37>:    mov    DWORD PTR [rbp-0x4],eax
   0x0000000000001278 <+40>:    cmp    DWORD PTR [rbp-0x4],0xffffffff
```

```
   0x000000000000127c <+44>:    jne    0x129c <read_file+76>
   0x000000000000127e <+46>:    lea    rax,[rip+0xdcb]        # 0x2050
   0x0000000000001285 <+53>:    mov    rdi,rax
   0x0000000000001288 <+56>:    mov    eax,0x0
   0x000000000000128d <+61>:    call   0x10c0 <printf@plt>
   0x0000000000001292 <+66>:    mov    edi,0x1
   0x0000000000001297 <+71>:    call   0x1100 <exit@plt>
   0x000000000000129c <+76>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000000000000129f <+79>:    mov    edx,0x80
   0x00000000000012a4 <+84>:    lea    rcx,[rip+0x2d95]        # 0x4040 <flag>
   0x00000000000012ab <+91>:    mov    rsi,rcx
   0x00000000000012ae <+94>:    mov    edi,eax
   0x00000000000012b0 <+96>:    call   0x10d0 <read@plt>
   0x00000000000012b5 <+101>:   lea    rax,[rip+0xdc3]        # 0x207f
   0x00000000000012bc <+108>:   mov    rdi,rax
   0x00000000000012bf <+111>:   call   0x10b0 <puts@plt>
   0x00000000000012c4 <+116>:   nop
   0x00000000000012c5 <+117>:   leave
   0x00000000000012c6 <+118>:   ret
End of assembler dump.
```

Here's what we can discover:

- From line `0x0000000000001268 <+24>` to line `0x000000000000127c <+44>`: call the `open` function with `rdi` as the address of the file path and `eax` (set to 0) as the file descriptor will be returned; compare the returned file descriptor stored at `[rbp-0x4]` with `-1` to check if the `open` function failed; jump to `0x129c` if the file was successfully opened.
- From line `0x000000000000129c <+76>` to `0x00000000000012b0 <+96>`: call the `read` function, using the file descriptor `edi`, the address of the buffer to store the read flag `rsi`, and the number of bytes to read `edx`.

To sum up, to get the flag, what we need to do is:

- Use `b *(read_file+96)` or `breakpoint *(read_file+96)` to set a breakpoint where the `read` function is called
- Use `r` or `run` to run the binary
- After hitting the first breakpoint at `read_file+96`, use `n` or `next` to go to the next instruction (call the `read` function)
- Use `x/s $rsi` to display the string (the flag we are looking for) stored at the memory address located at `rsi`

```
pwndbg> b *(read_file+96)
Breakpoint 1 at 0x12b0
pwndbg> r
Starting program: /home/ctf/gdb2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x00005577fae002b0 in read_file ()
......
──────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────
 ▶ 0x5577fae002b0 <read_file+96>            call   read@plt                    <read@plt>
        fd: 6 (/home/ctf/flag.txt)
        buf: 0x5577fae03040 (flag) ◂— 0
        nbytes: 0x80
......
pwndbg> n
0x00005577fae002b5 in read_file ()
......
──────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────
......
 RSI  0x5577fae03040 (flag) ◂— 'flag{gl4d_y0u_f1gur3d_0ut_h0w_t0_f1nd_th3_fl4g!_e340d4ab18df0ecf}\n'
 ......
──────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────
   0x5577fae002b0 <read_file+96>            call   read@plt                    <read@plt>

 ▶ 0x5577fae002b5 <read_file+101>           lea    rax, [rip + 0xdc3]     RAX => 0x5577fae0107f ◂—
0x443b031b0100
......
pwndbg> x/s $rsi
```

```
0x5577fae03040 <flag>:  "flag{gl4d_y0u_f1gur3d_0ut_h0w_t0_f1nd_th3_fl4g!_e340d4ab18df0ecf}\n"
```

The captured flag is `flag{gl4d_y0u_f1gur3d_0ut_h0w_t0_f1nd_th3_fl4g!_e340d4ab18df0ecf}`.

Directions (50 pts)

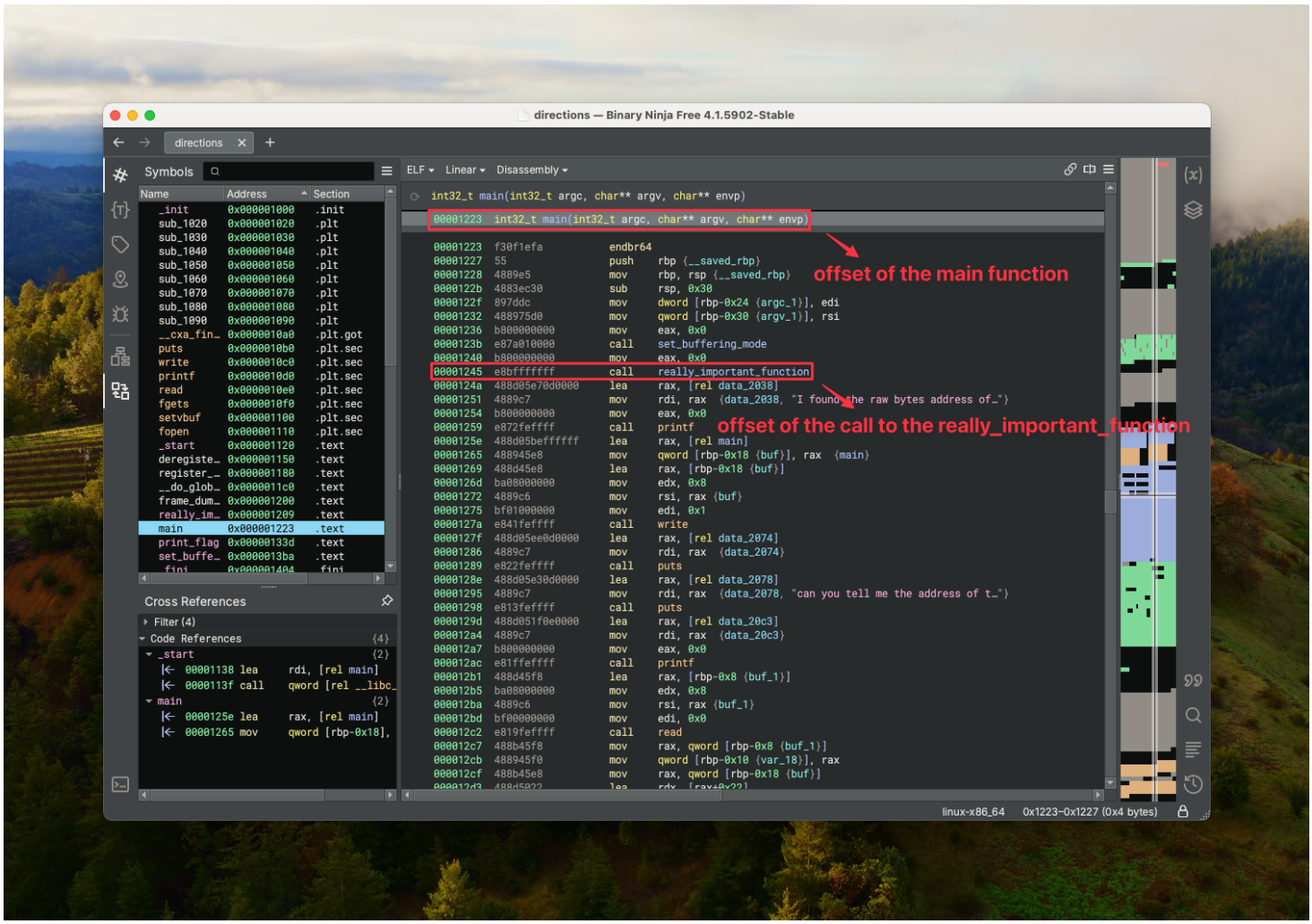
```
root@17b95fe8a8e6:~/wk1/directions# nc offsec-chalbroker.osiris.cyber.nyu.edu 1244
......
Let's practice finding addresses again!

I found the raw bytes address of main() written somewhere: #D\)V
can you tell me the address of the call to the really_important_function?
>
......
```

In this challenge, with the raw bytes address of the `main` function, we need to find the address of the `CALL` to the `really_important_function`.

Firstly, we learn that PIE is on for the binary `directions` using `readelf` with `-h` flag, showing `DYN` in the `Type` field.

```
root@17b95fe8a8e6:~/wk1/directions# readelf -h directions
ELF Header:
......
  Type:                              DYN (Position-Independent Executable file)
......
```

Then, we open the binary `directions` with Binary Ninja to inspect the disassembly of the `main` function.



We can discover that the offsets of the `main` function and the `CALL` to the `really_important_function` in hexadecimal are `0x1223` and `0x1245`.

Finally, we write a script using `pwntools` to obtain the raw bytes address of the `main` function, convert it to an unsigned integer, calculate the address of the `CALL` to the `really_important_function` by `really_important_function_CALL_addr = main_addr - main_offset + really_important_function_CALL_offset`, and

send it to the server in raw bytes.

Part of the script is as follows.

```
from pwn import *
......
p = remote(URL, PORT)
......
print(p.recvuntil(b": ").decode())
main_addr = p.recv(8)
log.info(f"Receiving address in raw bytes: {main_addr}")
print(p.recvuntil(b"> ").decode())
main_addr = u64(main_addr)
main_offset = 0x1223
really_important_function_CALL_offset = 0x1245
really_important_function_CALL_addr = main_addr - main_offset + really_important_function_CALL_offset
p.sendline(p64(really_important_function_CALL_addr))
log.info(f"Sending address in raw bytes: {p64(really_important_function_CALL_addr)}")

p.interactive()
```

The captured flag is `flag{st4t1c_4n4lys1s_g1v3s_us_s0_much_1nf0_4b0ut_4_b1n4ry!_33af9f51381f50c1}`.

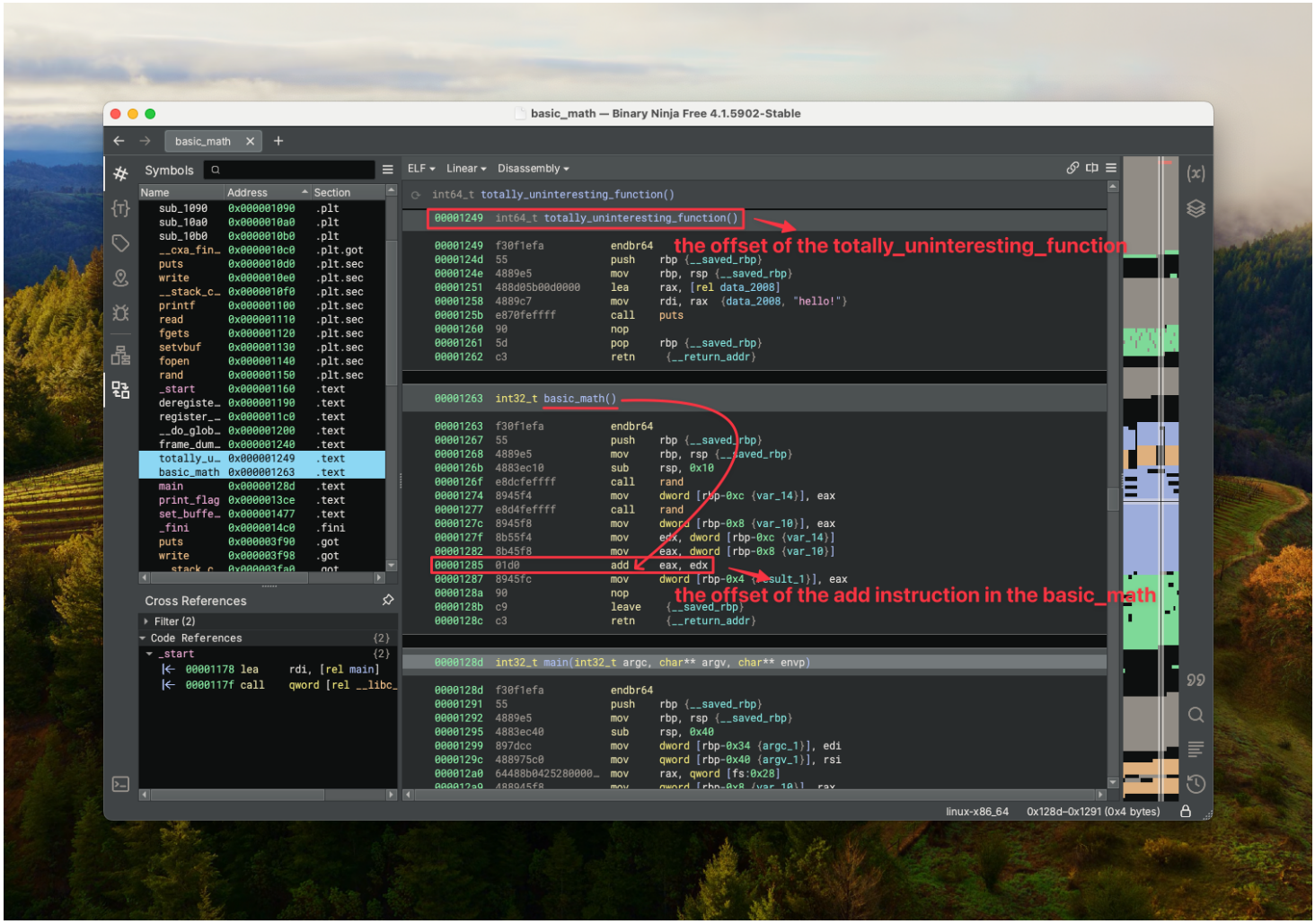
Basic Math (100 pts)

```
root@17b95fe8a8e6:~/wk1/directions# nc offsec-chalbroker.osiris.cyber.nyu.edu 1245
......
I found the raw bytes address of `totally_uninteresting_function` written somewhere: IR=@V
can you tell me the address of the ADD instruction in basic_math?
>
......
```

In this challenge, with the raw bytes address of the `totally_uninteresting_function`, we need to find the address of the `ADD` instruction in the `basic_math` function.

Firstly and similarly, we learn that PIE is on for the binary `basic_math`.

Then, we open the binary `basic_math` with Binary Ninja to inspect the disassembly of the two functions, `totally_uninteresting_function` and `basic_math`.



We can discover that the offsets of the `totally_uninteresting_function` and the `ADD` instruction in the the `basic_math` in hexadecimal are `0x1249` and `0x1285`.

Finally, we write a script using `pwntools` to obtain the raw bytes address of the `totally_uninteresting_function`, convert it to an unsigned integer, calculate the address of the `ADD` instruction in the the `basic_math` by `basic_math_ADD_addr = totally_uninteresting_function_addr - totally_uninteresting_function_offset + basic_math_ADD_offset`, and send it to the server in raw bytes.

Part of the script is as follows.

```
from pwn import *
......
```

```
p = remote(URL, PORT)
......
print(p.recvuntil(b": ").decode())
totally_uninteresting_function_addr = p.recv(8)
log.info(f"Receiving address in raw bytes: {totally_uninteresting_function_addr}")
print(p.recvuntil(b"> ").decode())
totally_uninteresting_function_addr = u64(totally_uninteresting_function_addr)
totally_uninteresting_function_offset = 0x1249
basic_math_ADD_offset = 0x1285
basic_math_ADD_addr = totally_uninteresting_function_addr - totally_uninteresting_function_offset +
basic_math_ADD_offset
p.sendline(p64(basic_math_ADD_addr))
log.info(f"Sending address in raw bytes: {p64(basic_math_ADD_addr)}")

p.interactive()
```

The captured flag is `flag{R34d1ng_4ss3mbly_l4ngu4ge_w4snt_th4t_h4rd!_1e237660c13c7494}`.