lab3

杨乙　　21307130076　信息安全

Task 1

经过反汇编的 task1 文件中，`main` 函数对应的汇编代码如下：

```
0000000000000e64 <main>:
 e64:   a9be7bfd    stp x29, x30, [sp, #-32]!
 e68:   910003fd    mov x29, sp
 e6c:   b9001fe0    str w0, [sp, #28]
 e70:   f9000be1    str x1, [sp, #16]
 e74:   97ffff85    bl  c88 <dummy_func>
 e78:   d28005e1    mov x1, #0x2f                   // #47
 e7c:   90000100    adrp    x0, 20000 <memcpy@GLIBC_2.17>
 e80:   9102e000    add x0, x0, #0xb8
 e84:   97ffff54    bl  bd4 <rand_str>
 e88:   f9400be0    ldr x0, [sp, #16]
 e8c:   91002000    add x0, x0, #0x8
 e90:   f9400000    ldr x0, [x0]
 e94:   d28005e2    mov x2, #0x2f                   // #47
 e98:   aa0003e1    mov x1, x0
 e9c:   90000100    adrp    x0, 20000 <memcpy@GLIBC_2.17>
 ea0:   9102e000    add x0, x0, #0xb8
 ea4:   97fffedf    bl  a20 <memcmp@plt>
 ea8:   7100001f    cmp w0, #0x0
 eac:   540000a1    b.ne    ec0 <main+0x5c>  // b.any
 eb0:   90000100    adrp    x0, 20000 <memcpy@GLIBC_2.17>
 eb4:   9102e000    add x0, x0, #0xb8
 eb8:   97fffed6    bl  a10 <puts@plt>
 ebc:   14000004    b   ecc <main+0x68>
 ec0:   90000000    adrp    x0, 0 <__abi_tag-0x278>
 ec4:   913d6000    add x0, x0, #0xf58
 ec8:   97fffed2    bl  a10 <puts@plt>
 ecc:   52800000    mov w0, #0x0                    // #0
 ed0:   a8c27bfd    ldp x29, x30, [sp], #32
 ed4:   d65f03c0    ret
```

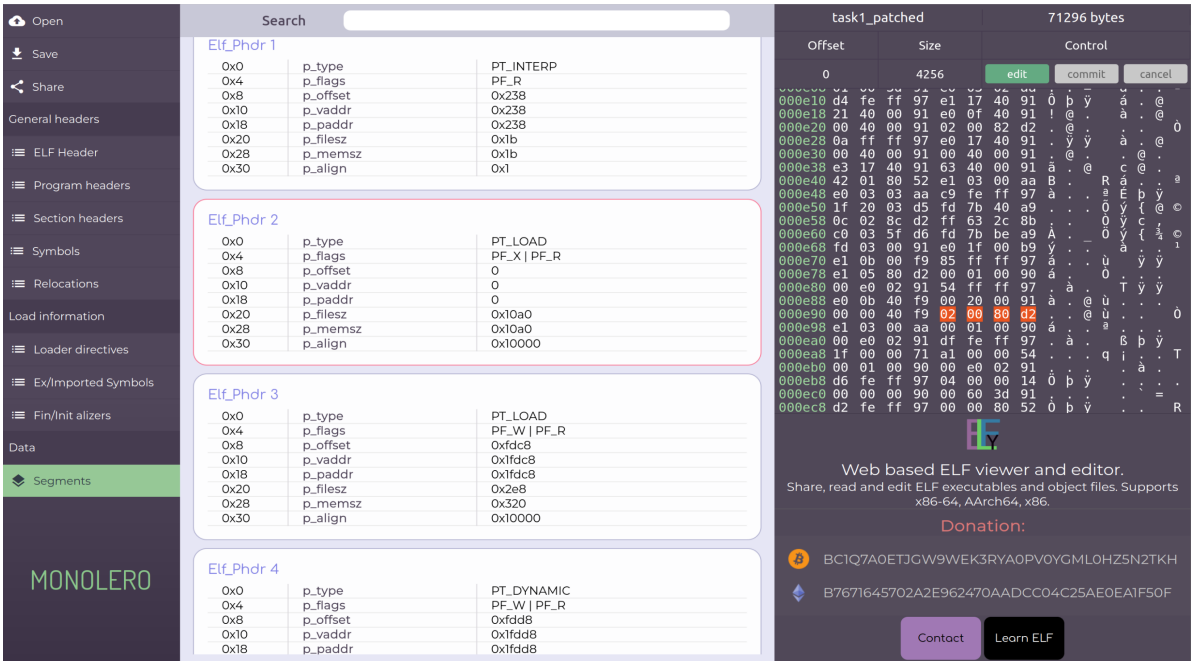
`0x2f` 作为 `memcmp` 函数的第二个参数，通过 `x2` 寄存器来传递。因此可以定位到需要修改的指令：

```
e94:   d28005e2    mov x2, #0x2f
```

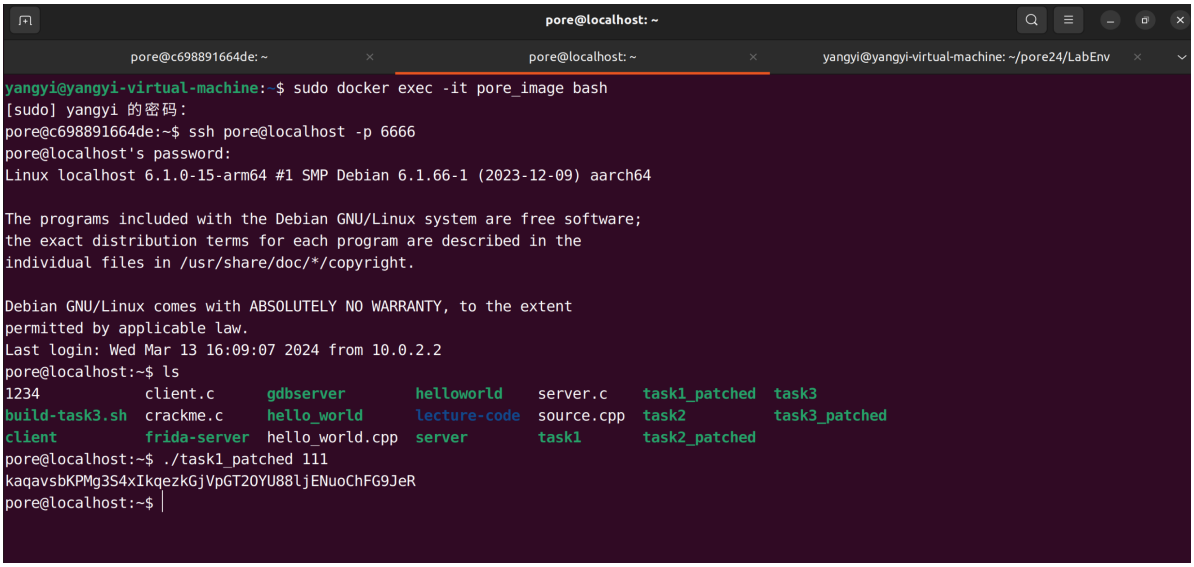
需要修改为：

```
e94:   d2800002    mov x2, #0x0
```

在 ELF 编辑器中找到地址 `0xe94`，对指令内容进行如下修改：



运行修改后的文件，结果如下：



Task 2

经过反汇编的 task1 文件中 `main` 函数对应的汇编代码同 Task1

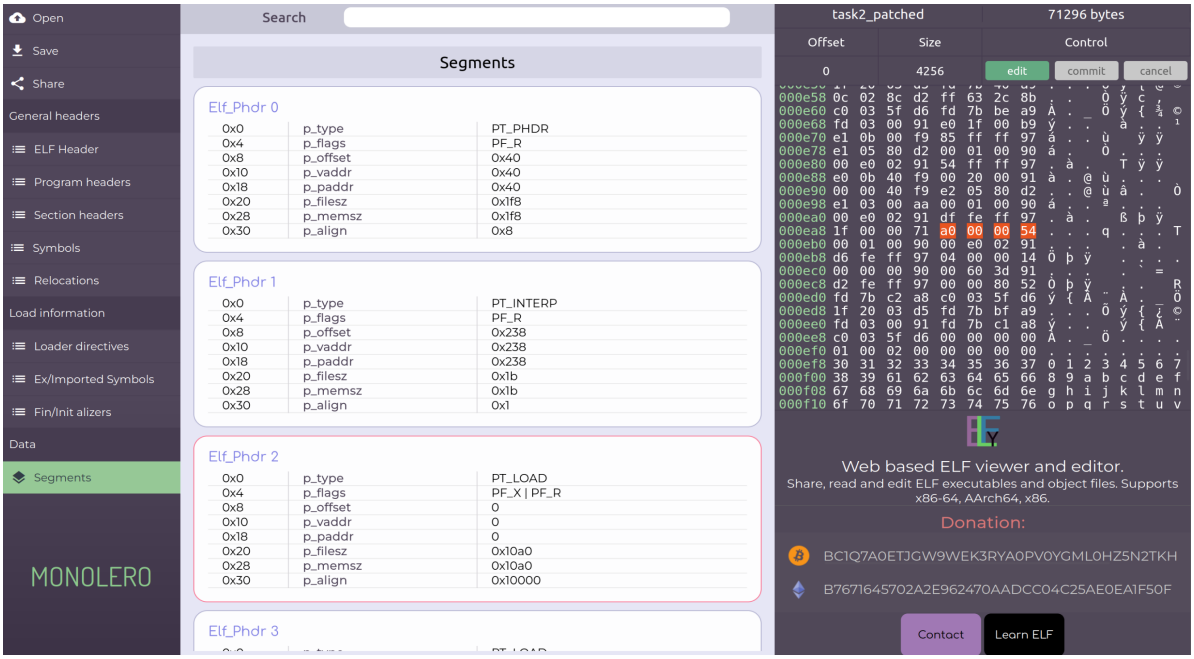
`main` 函数中唯一一条分支指令如下：

```
eac:    540000a1    b.ne    ec0 <main+0x5c>  // b.any
```

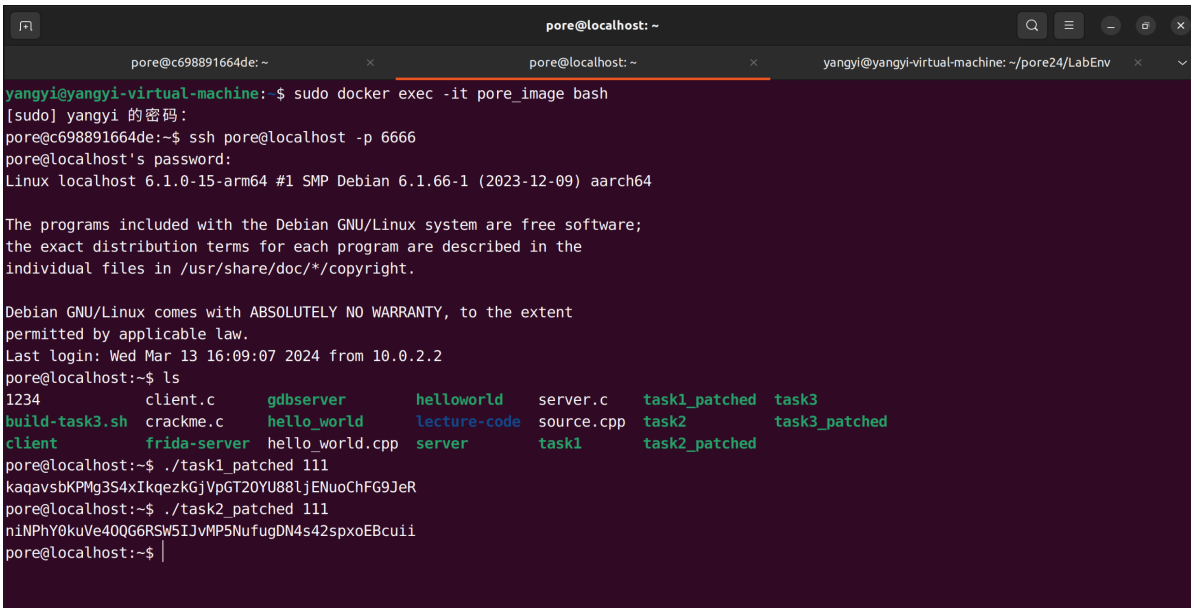
若 `memcmp` 函数返回值不为 0（用户输入和 `flag` 不相等），则 `w0` 寄存器值和 `0x0` 不相等，跳转到 `0xec0`，不执行 `puts` 函数。通过反转分支，可以避免跳转：

```
eac:    540000a0    b.eq    ec0 <main+0x5c>
```

在 ELF 编辑器中找到地址 `0xeac`，对指令内容进行如下修改：



运行修改后的文件，结果如下：



Task3

经过反汇编的 server 文件中， `plt` 表对应的汇编代码如下：

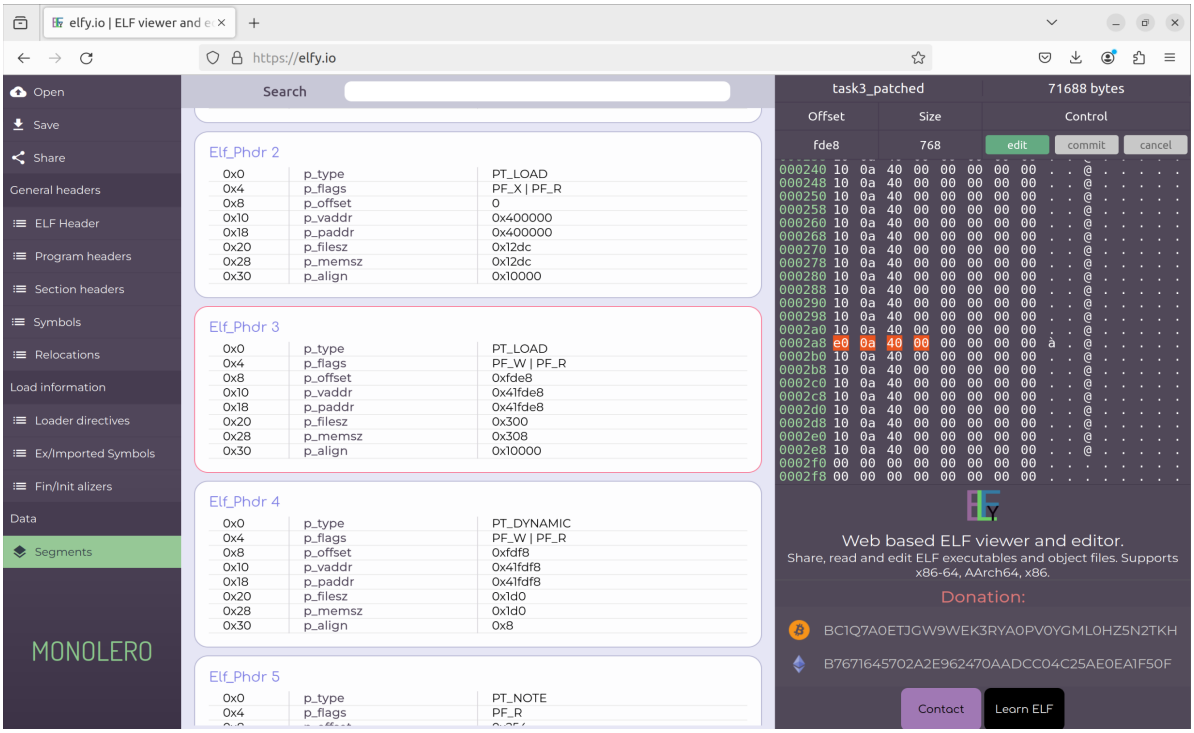
```
./server:     file format elf64-littleaarch64

Disassembly of section .plt:

0000000000400a10 <.plt>:
  400a10:   a9bf7bf0    stp x16, x30, [sp, #-16]!
  400a14:   f00000f0    adrp    x16, 41f000 <__FRAME_END__+0x1dd28>
  400a18:   f947fe11    ldr x17, [x16, #4088]
  400a1c:   913fe210    add x16, x16, #0xff8
```

```
  400a20:   d61f0220    br  x17
  400a24:   d503201f    nop
  400a28:   d503201f    nop
  400a2c:   d503201f    nop

// ......

0000000000400ae0 <system@plt>:
  400ae0:   90000110    adrp    x16, 420000 <memcpy@GLIBC_2.17>             //
0x420058
  400ae4:   f9402e11    ldr x17, [x16, #88]
  400ae8:   91016210    add x16, x16, #0x58
  400aec:   d61f0220    br  x17

// ......

0000000000400b50 <puts@plt>:
  400b50:   90000110    adrp    x16, 420000 <memcpy@GLIBC_2.17>         //
0x420090
  400b54:   f9404a11    ldr x17, [x16, #144]
  400b58:   91024210    add x16, x16, #0x90
  400b5c:   d61f0220    br  x17

// ......
```

根据指令得到，`system` 函数在 got 表中地址为 `0x420058`，`puts` 函数在 got 表中地址为 `0x420090`；**如果想把 `puts` 函数劫持为 `system` 函数，可以将 `puts` 函数在 got 表中的地址改写为 `system` 函数在 plt 表中的地址。**修改后程序运行流程如下（因为 `system` 函数之前调用过，所以它的 got 表项中已经是 `system` 函数的地址）：

```
    main 函数 ->
    puts 函数的 plt 表项 ->
    puts 函数的 got 表项 ->
    system 函数的 plt 表项 ->
    system 函数的 got 表项 ->
    system 函数地址
```

此时 plt0 处理逻辑解析得到的函数地址变成了 `system` 函数的地址。将地址 `0x420090` 处内容修改为 `400ae0`：



运行修改后的程序，结果如下：

客户端：

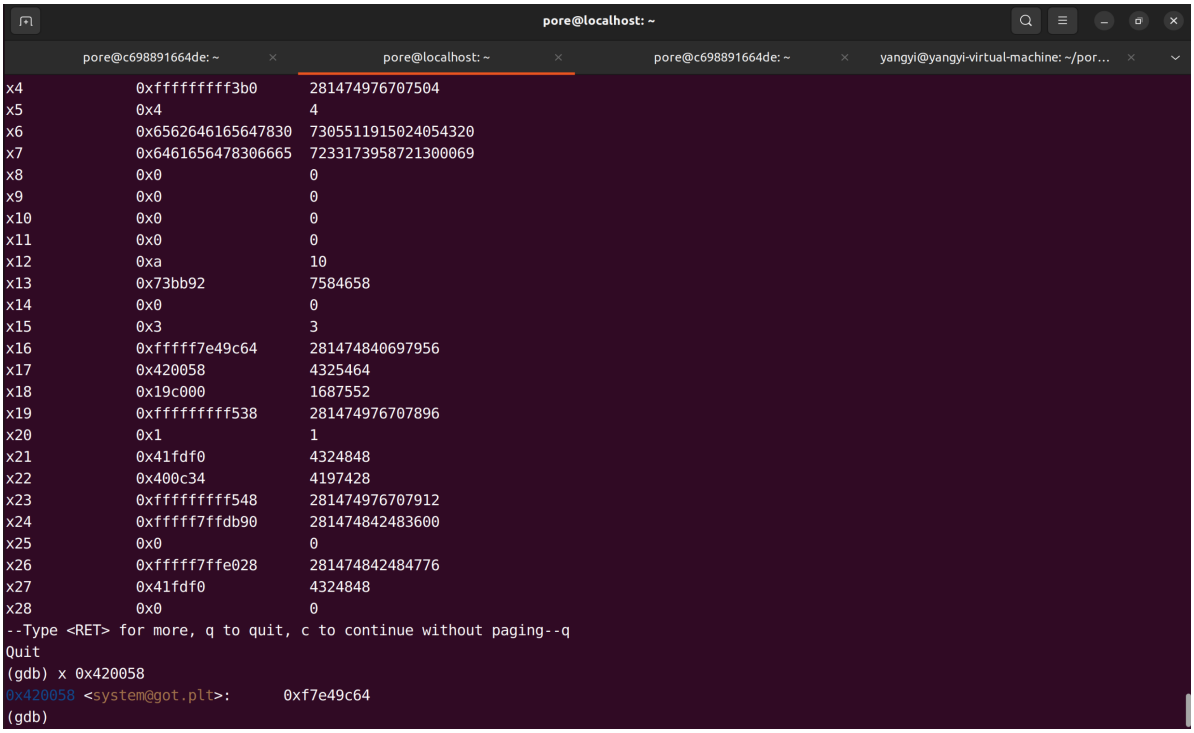
```
pore@localhost:~$ ./client
连接成功!
touch 1234
Send msg:touch 1234
```

服务器（不再输出内容，文件目录下多了 1234）：

```
pore@localhost:~$ ./task3_patched
PoRE Server
接收成功! pore@localhost:~$ ls
1234            client.c      gdbserver        helloworld    server.c    task1_patched  task3
build-task3.sh  crackme.c     hello_world      lecture-code  source.cpp  task2          task3_patched
client          frida-server  hello_world.cpp  server        task1       task2_patched
pore@localhost:~$
```

P.S.

遇到的问题：最初我采用的方法是通过 GDB 调试，在 `system` 函数处打断点，获取 `system` 函数的 got 表项，即 `system` 函数的地址（如下图）。再将这个地址填入 `puts` 函数的 got 表项



设想因为之前已经调用过 `system` 函数，则函数已经链接并且加载到内存中。但是结果服务端会报错 Segmentation fault（下图），推测可能是因为某些原因 `system` 函数的地址发生了变化（如内存交换等）