Andrew login ID: ________________________________
Full Name: ________________________________
Recitation Section: ________________________________

CS 15-213 / ECE 18-243, Spring 2010
Exam 1
Version 1100101
Tuesday, March 2nd, 2010

Instructions:

• Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front. Read all instructions and sign the statement below.

• Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.

• The exam has a maximum score of 100 points.

• The problems are of varying difficulty. The point value of each problem is indicated (instructors reserve the right to change these values). Pile up the easy points quickly and then come back to the harder problems.

• You may not use any books or notes on this exam. Reference material is located at the end of this exam. No calculators or other electronic devices are allowed.

• Good luck!

I understand the CMU policy on cheating applies in full to this exam. ________________________________
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1- Multiple Choice (14):</td>
<td></td>
</tr>
<tr>
<td>2- Peephole (16):</td>
<td></td>
</tr>
<tr>
<td>3- Floating Point (14):</td>
<td></td>
</tr>
<tr>
<td>4- Structs (14):</td>
<td></td>
</tr>
<tr>
<td>5- Stacks (15):</td>
<td></td>
</tr>
<tr>
<td>6- Buffer Overflow (17):</td>
<td></td>
</tr>
<tr>
<td>7- Assembly (10):</td>
<td></td>
</tr>
<tr>
<td>TOTAL (100):</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. (14 points):

1. Which of the following lines of C code performs the same operation as the assembly statement 
   `lea 0xffffffff(%esi),%eax`.
   (a) `*(esi-1) = eax`
   (b) `esi = eax + 0xffffffff`
   (c) `eax = esi - 1`
   (d) `eax = *(esi -1)`

2. `test %eax, %eax`  
   `jne 3d<function+0x3d>`
   Which of the following values of `%eax` would cause the jump to be taken?
   (a) 1
   (b) 0
   (c) Any value of `%eax`
   (d) No value of `%eax` would cause the jump to be taken.

3. Which of the following are legitimate advantages of x86_64 over IA32? (Circle 0-3)
   (a) x86_64 is able to make use of a larger address space than IA32
   (b) x86_64 is able to make use of more registers than IA32
   (c) x86_64 is able to make use of larger registers than IA32

4. T/F: Any sequence of IA32 instructions can be executed on an x86_64 processor?
   (a) True
   (b) False

5. What sequence of operations does the `leave` instruction execute?
   (a) `mov %ebp,%esp
       pop %ebp`
   (b) `pop %ebp
       mov %ebp,%esp`
   (c) `pop %esp
       mov %ebp,%esp`
   (d) `push %ebp
       mov %esp,%ebp`
6. What is the difference between the %rbx and the %ebx register on an x86_64 machine?
   (a) nothing, they are the same register
   (b) %ebx refers to only the low order 32 bits of the %rbx register
   (c) they are totally different registers
   (d) %ebx refers to only the high order 32 bits of the %rbx register

7. On IA32 systems, where is the value of old %ebp saved in relation to the current value of %ebp?
   (a) there is no relation between where the current base pointer and old base pointer are saved.
   (b) old %ebp is stored at (%ebp - 4)
   (c) old %ebp is stored at (%ebp + 4)
   (d) old %ebp is stored at (%ebp)
Problem 2. (16 points):
Consider the following assembly code:

1. Fill in the blanks of the corresponding C function:

```c
int mystery(int i)
{
    if (______________) return ______________________;
    return _____________________;
}
```
2. Peephole optimizations are a kind of optimization which looks at a small number of assembly instructions and tries to optimize those instructions. Care must be taken to not affect the behavior of the rest of the program. Write an optimized version of the assembly instructions at addresses 0x804833d and 0x8048344.

3. If we look at the addresses 0x8048361 and 0x8048364 it seems like we can eliminate both instructions or replace the instructions with nops. Explain why we can’t implement this peephole optimization without affecting the behavior of the rest of the function.
Problem 3. (14 points):

Your friend, Harry Q. Bovik, encounters a function named mystery when running gdb on a 32-bit binary that was compiled on the fish machines. Use the gdb output below and the function prototype for mystery to complete this question.

```c
int mystery(float arg1, float arg2, float arg3, float arg4);
```

Breakpoint 1, 0x08048366 in mystery ()
(gdb) x/20 $esp
0xffd3d1e0: 0xf7f3fff4 0xf7f3e204 0xffd3d208 0x080483cd
0xffd3d1f0: 0x41700000 0x3de00000 0x7f800010 0x00000001
0xffd3d200: 0x7f7fffff 0xffd3d220 0xffd3d278 0xf7e13e9c
0xffd3d210: 0xf7f5fca0 0x080483f0 0xffd3d278 0xf7e13e9c
0xffd3d220: 0x00000001 0xffd3d2a4 0xffd3d2ac 0xf7f60810
(gdb) print $ebp
$1 = (void *) 0xffd3d1e8

1. What is on the stack where %ebp is pointing (in hex)?

2. What is the return address of the function mystery?

Fill in the below table. Hexadecimal may be used in the address column. The value column may not contain any binary. Instead of calculating large powers of two you may use exponentials in the value column but your answer must fit within the table boundaries.

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1</td>
<td></td>
</tr>
<tr>
<td>arg2</td>
<td></td>
</tr>
<tr>
<td>arg3</td>
<td></td>
</tr>
<tr>
<td>arg4</td>
<td></td>
</tr>
</tbody>
</table>
**Problem 4. (14 points):**

Take the struct below compiled on Linux 32-bit:

```c
struct my_struct {
    short b;
    int x;
    short s;
    long z;
    char c[5];
    long long a;
    char q;
}
```

1. Please lay out the struct in memory below (each cell is 1 byte). Please shade in boxes used for padding.

```
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
| | | | | | | | 
+----+----+----+----+----+----+----+----+
```

Page 8 of 14
Given the following gdb interaction (where ms is a struct my_struct).

```
(gdb) x/40b &ms
0xffffffffde0: 0xbb 0x00 0x86 0x47 0xf9 0xd9 0x01 0x00
0xffffffffde8: 0x6d 0x3b 0xff 0xff 0xbe 0xba 0xef 0xbe
0xffffffffdf0: 0x68 0x6c 0x70 0x6d 0x65 0x00 0x00 0x00
0xffffffffdf8: 0x1e 0xab 0xdf 0x1e 0xff 0xe1 0xaf 0xde
0xffffffffe00: 0x21 0x00 0x00 0x00 0xf4 0x7f 0x86 0x47
```

2. Label the fields above and fill in the values below.

- ms.b = 0x__________
- ms.x = 0x__________
- ms.s = 0x__________
- ms.z = 0x__________
- ms.c = ______, ______, ______, ______, ______
- ms.a = 0x__________
- ms.q = 0x__________

3. Define a struct with the same elements that has a total size of less than 30 bytes.

```c
struct my_compressed_struct {

}
```

4. What is the size of `my_compressed_struct` that you wrote above?
Problem 5. (15 points):
Below is the C code and assembly code for a simple function.

```c
int doSomething(int a, int b, int c){
    int d;
    if (a == 0){ return 1;}
    d = a/2;
    c = doSomething(d,a,c);
    return c;
}
```

Please draw a detailed stack diagram for this function in Figure 1 on the next page, starting with a function that calls this function and continuing for 2 recursive calls of this function. (That is, at least two stack frames that belong to this function). Please label everything you can.
Problem 6. (17 points):

As a security engineer for a software company it is your job to perform attacks against your company’s software and try to break it. One of your developers, Harry Q. Bovik, has written a password validator that he thinks is unbreakable! Below is the front-end to his system:

```c
int main(){
    char buffer[20];

    printf("Enter your password >");
    scanf("%s",buffer);
    if(validate(buffer)){
        getOnTheBoat();
        exit(0);
    }
    printf("Sorry, you do not have access :(");
    return 0;
}
```

Step 0: Briefly explain how you could attack this program with a buffer overflow. (25 words or less).

Harry then mentions that you actually cannot perform that attack because he runs this on a special system where the stack is not-executable. This means that it is impossible to execute any code on the stack, making the typical attack you performed in buffer-lab now impossible.

You can still do this though! You are going to perform a RETURN TO LIBC attack! This attack relies on pre-existing code in the program that will allow you to execute arbitrary instructions. There are a few important things you need to know about first:

The C function `system(char * command)` will execute the string `command` as if you had typed it into a shell prompt.

Using GDB you discover:

```
(gdb) print system
$1 = {<text variable, no debug info>} 0xf7e263a0 <system>
```

In every program executable, your environment variables are loaded at runtime. And part of your environment variables is your current SHELL:

```
(gdb) print (char *) 0xff89d957
$2 = 0xff89d957 "SHELL=/bin/bash"
```
Using this information, you can successfully launch a shell from Harry’s program, proving that you can execute arbitrary code with his program’s privilege level!

Step 1:

• What is the address of the system() function?

• What is the address of the string ”/bin/bash”?

Step 2:  Design your exploit string (keep in mind where arguments go for IA32 ). We’re looking for an drawing of what you can pass as input to this program causing it to launch a shell. Don’t worry about exact sizes/lengths.

Step 3: Explain how your exploit string will allow you to execute a shell on Harry’s program. This combined with your answer to Step 2 should be enough to prove Harry wrong. (This will be graded independently of your Step 2).
Problem 7. (10 points):

Use the x86_64 assembly to fill in the C function below

```c
void mystery(int (*funcP)(int), int a[], int n) {
```

```assembly
0x0000000000000400498 <mystery+0>: push %r13
0x000000000000040049a <mystery+2>: push %r12
0x000000000000040049c <mystery+4>: push %rbp
0x000000000000040049d <mystery+5>: push %rbx
0x000000000000040049e <mystery+6>: sub $0x8,%rsp
0x00000000000004004a2 <mystery+10>: mov %rdi,%r13
0x00000000000004004a5 <mystery+13>: mov %edx,%r12d
0x00000000000004004a8 <mystery+16>: test %edx,%edx
0x00000000000004004aa <mystery+18>: jle 0x4004c7 <mystery+47>
0x00000000000004004ac <mystery+20>: mov %rsi,%rbx
0x00000000000004004af <mystery+23>: mov $0x0,%ebp
0x00000000000004004b4 <mystery+28>: mov (%rbx),%edi
0x00000000000004004b6 <mystery+30>: callq *%r13
0x00000000000004004b9 <mystery+33>: mov %eax,(%ebx)
0x00000000000004004bb <mystery+35>: add $0x1,%ebp
0x00000000000004004be <mystery+38>: add $0x4,%rbx
0x00000000000004004c2 <mystery+42>: cmp %r12d,%ebp
0x00000000000004004c5 <mystery+45>: jne 0x4004b4 <mystery+28>
0x00000000000004004c7 <mystery+47>: add $0x8,%rsp
0x00000000000004004cb <mystery+51>: pop %rbx
0x00000000000004004cc <mystery+52>: pop %rbp
0x00000000000004004cd <mystery+53>: pop %r12
0x00000000000004004cf <mystery+55>: pop %r13
0x00000000000004004d1 <mystery+57>: retq
```