Full Name:____

Andrew ID (print clearly!):____

15-213/18-213, Spring 2013

SOLUTIONS TO Exam 1

Tuesday, March 5, 2013

Instructions:

- Make sure that your exam has 14 pages and is not missing any sheets, then write your full name and Andrew login ID on the front.
- This exam is closed book. You may not use **any** electronic devices. You may use one single-sided page of notes that you bring to the exam.
- Write your answers in the space provided below 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. Good luck!

Problem	Your Score	Possible Points
1		14
2		20
3		6
4		13
5		9
6		15
7		23
Total		100

Problem 1: (14 pts)

Please answer the following multiple choice questions by writing the correct number in the blank to the right of the question.

A) What is a linker? **1**

1. It combines object files into an executable.
2. It turns assembly code into machine code.
3. It translates source code to assembly.
4. It executes source code.

B) What is true about pending and/or blocked signals? **4**

1. User applications maintain the pending and blocked vectors for each process.
2. Blocked signals cannot be delivered to the blocking process.
3. Signals of the same type can be queued when they are pending.
4. Pending signals of the same type can only be received once.

C) Which of the following is the BEST example of spatial locality? **1**

1. Referencing array elements in succession.
2. Cycling through a loop repeatedly.
3. Allocating space for a struct or a union.
4. Continuously referencing the same local variable.

D) What is a fundamental idea of the memory hierarchy? **3**

1. To create a large amount of storage that is expensive and fast.
2. To create a small amount of storage that is expensive and slow.
3. Smaller, faster devices serve as caches for larger, slower devices.
4. Larger, slower devices serve as caches for smaller, faster devices.

E) In datalab, what was one thing you had to check to solve bang()? **4**

1. Whether the input's least significant byte was 0xFF.
2. Whether the input's most significant byte was 0xFF.
3. Whether the input's most significant bit was set to 1.
4. Whether the input's negation was equal to the input.

F) In bomblab, how were arguments to functions passed? **2**

1. Via memory.
2. Via registers.
3. Via the stack.
4. Via files.

G) In the Dynamite phase of buflab, what was one valid way to restore %ebp? **3**

1. By setting %ebp equal to %esp in your exploit code.
2. By using %ebp as is and not restoring it at all.
3. By finding %ebp with GDB and restoring it in your exploit.
4. By finding %ebp with the disassembly and restoring it in your exploit.

H) Which of the following is NOT true about the cachelab cache simulator? **2**

1. It used the B value to get the offset set bits.
2. It stored content from memory using the addresses.
3. It did not use "size" from the Valgrind traces.
4. It needed tag bits to simulate evictions.

I) The shark machines are which endianness? **1**

1. Little Endian
2. Big Endian

J) Which of the following is **NOT** a strong symbol? **2**

1. Procedures
2. Uninitialized globals
3. Initialized globals

K) In buflab, what was the fundamental problem that allowed for arbitrary code execution? **2**

1. The buffer was allocated with malloc.
2. The "Gets" function wrote outside of the buffer memory
3. The use of a 64-bit machine.

L) The difference between dynamite and nitro in buflab was: **5**

1. in nitro, %ebp could be set to an absolute value
2. in dynamite, the nop sled had a variable position
3. in nitro, the nop sled had a fixed position
4. all of the above
5. none of the above

M) A nop sled is ... **3**

1. a degradation of cache performance due to spillover from one level of cache to the next
2. a buffer overflow technique that uses assembly instruction *sled*
3. a collection of *nop* assembly instructions leading the %eip to the exploit
4. none of the above

Problem 2: Interpreting Assembly (20 pts)

Consider the following C code compiled for a 32-bit x86 machine:

```
struct point
{
    char x;
    double y;
    int z;
};

int el(int x);
int psy(int x);
int congroo(struct point *t);
```

```
int main()
{
    struct point p = {8, 5000, 103};
    int answer1 = el(-251);
    int answer2 = psy(56);
    int answer3 = congroo(&p);
    printf("answer1 = %d\n", answer1);
    printf("answer2 = %d\n", answer2);
    printf("answer3 = %d\n", answer3);
    return 0;
}
```

Using the following assembly code for `el()`, `psy()`, and `congroo()`, as well as the C code, answer the questions on the next page about each function's output and behavior:

```
080483c5 <el>:
 80483c5:             55              push   %ebp
 80483c6:             89  e5          mov    %esp,%ebp
 80483c8:             8b  55 08       mov    0x8(%ebp),%edx
 80483cb:             b8  00 00 00 00 mov    $0x0,%eax
 80483cd:             eb  06          jmp    80483d3 <el+0xe>
 80483ce:             83  c0 01       add    $0x1,%eax
 80483d0:             83  c2 01       add    $0x1,%edx
 80483d3:             85  d2          test   %edx,%edx
 80483d5:             78  f3          js     80483ce <el+0x9>
 80483d8:             c9              leave
 80483d9:             c3              ret

080483da <psy>:
 80483da:             55              push   %ebp
 80483db:             89  e5          mov    %esp,%ebp
 80483dd:             8b  55 08       mov    0x8(%ebp),%edx
 80483e0:             f7  da          neg    %edx
 80483e2:             b8  00 00 00 00 mov    $0x0,%eax
 80483e7:             83  fa f9       cmp    $0xfffffff9,%edx
 80483ea:             74  0a          je     80483f6 <psy+0x1c>
 80483ec:             d1  fa          sar    %edx
 80483ee:             83  c0 01       add    $0x1,%eax
 80483f1:             83  fa f9       cmp    $0xfffffff9,%edx
 80483f4:             75  f6          jne    80483ec <psy+0x12>
 80483f6:             c9              leave
 80483f7:             c3              ret

080483f8 <congroo>:
 80483f8:             55              push   %ebp
 80483f9:             89 e5           mov    %esp,%ebp
 80483fb:             8b 45  08       mov    0x8(%ebp),%eax
 80483fe:             0f be  08       movsbl (%eax),%ecx
 8048401:             8b 50  0c       mov    0xc(%eax),%edx
 8048404:             89 d0           mov    %edx,%eax
 8048406:             c1 fa  1f       sar    $0x1f,%edx
 8048409:             f7 f9           idiv   %ecx
 804840b:             89 d0           mov    %edx,%eax
 804840d:             c9              leave
 804840e:             c3              ret
```

A) [6 pts] Consider the function `el()`:

When the following values are in `%edx`, fill in the corresponding values of `%eax` when `%eip` is pointing to `0x80483ce`:

%edx	%eax
-251	**0**
-100	**151**
-1	**250**

What does `main()` print for answer1 when function `el()` returns?

answer1 = **251**

B) [6 pts] Consider the function `psy()`:

1) What value (signed base 10) is `%edx` being compared to when %eip points to `0x80483e7`? **-7**

2) How many times does the shift at the instruction at address `0x80483ec` occur? **3**

3) What does `main()` print for answer2 when function `psy()` returns?

answer2 = **3**

C) [8 pts] Consider the function `congroo()`:

1) What value (signed base 10) will be stored in `%ecx` after the instruction at `0x80483fe` is executed? **8**

2) What value (signed base 10) will be stored in `%edx` after the instruction at `0x8048401` is executed? **103**

3) What does `main()` print for answer3 when function `congroo()` returns?

answer3 = **7**

Problem 3: Bitey is a good snake name. (6 pts)

You are working on a machine with **16 bit** integers. The variables x, p and q are signed integers in two's complement. Match the expressions on the left to the code snippets on the right (by writing the letter in the blank space). *If an expression on the left does NOT have a corresponding code snippet, then leave the blank empty.*

1) $19 * x$ **F**

2) $x > 0$ **C**

3) Round x down to a multiple of 32. **D**

4) Absolute value of x. **A**

5) $p \oplus q$ (xor). **B**

A) $x * ((x >> 15)|1)$

B) $(p\&(\sim q))|((\sim p)\&q)$

C) $(x\& \; MIN\_INT) == 0$

D) $(x >> 5) << 5$

E) $(x\&(((unsigned) - 1) >> 11))$

F) $(x << 4) + (x << 1) + x$

Problem 4: Cache (13 pts)

Given an 32-bit Linux system, consider a 2-way associative cache of size 64 bytes with 16 bytes per block. The replacement policy that the cache adopt is LRU (Least Recent Used).

A) [2 pts] Warm up:

1) How many sets are there in the cache?______**2**______

2) How many cache lines are there in each set?______**2**______

B) [6 pts] Assume the following:

- one access to memory costs 100 ns
- one access to cache costs 1 ns
- ignore other time costs that might occur (eviction, store to cache, etc.)
- If cache is in use, we will always access the cache first (**Thus, in this case a cache miss is 101ns.**)

Given an integer array:

```
int Arr[6][4];
```

The Arr array starts at address 0x00000000.

If we were to access the following elements in the array one by one:

1) Fill in the blank slots with H or M, meaning cache hit and cache miss respectively.

Access	Cache Result
Arr[0][0]	**M**
Arr[0][2]	**H**
Arr[0][3]	**H**
Arr[1][1]	**M**
Arr[1][3]	**H**
Arr[2][1]	**M**

2) What is the time cost if cache is NOT used? ____**600ns**____

3) What is the time cost if cache is used? ____**306ns**____

C) [5 pts] If we were to access the array using the following program:

```
int i, j;
for ( i = 0; i < 4; i++) {
        for ( j = 0; j < 6; j++) {
                Arr[j][i]++;
        }
}
```

1) What is the time cost if cache is NOT used? ____**4800ns**____

2) What is the time cost if cache is used? ____**2448ns**____

Problem 5: Float On (9 pts)

Consider a 6-bit floating point representation based on the IEEE standard. This representation has no sign bit, it can only represent positive numbers.

- There are $k = 3$ exponent bits.
- There are $n = 3$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M$ x 2^E, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1-$ Bias for denormalized values and $E = e-$ Bias for normalized values, where e is the value of the exponent field exp interpreted as an unsigned number.

Given the table below, please show the corresponding floating point representation based on the modified 6-bit IEEE format described above for each decimal value. In addition, you should provide the rounded value of the encoded floating point number. To get full credit, you must give these as whole numbers (e.g.,17) or as fractions in reduced form (e.g.,$\frac{3}{4}$). Any rounding of the significand is based on **round-to-even**, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Write the floating point encoding of the following values and their rounded values. Remember that floats round-to-even.

Value	Floating Point Bits	Rounded Value
37/32	`011 001`	9/8
7/2	**100 110**	**7/2**
5/32	**000 101**	**5/32**
17/2	**110 000**	**8**
15213	**111 000**	**infinity**

Problem 6: Stack Discipline (15 pts)

Consider the following C code and its corresponding **32-bit** x86 machine code.

```
struct node_t {
    int data;
    struct node_t *next;
};

int fun (struct node_t *node) {
    if (!node)  return 0;
    else        return node->data + fun(node->next);
}
```

```
080483f4 <fun>:
 80483f4: 55                      push   %ebp
 80483f5: 89 e5                   mov    %esp,%ebp
 80483f7: 53                      push   %ebx
 80483f8: 83 ec 04                sub    $0x4,%esp
 80483fb: 83 7d 08 00             cmpl   $0x0,0x8(%ebp)
 80483ff: 75 07                   jne    8048408 <fun+0x14>
 8048401: b8 00 00 00 00          mov    $0x0,%eax
 8048406: eb 16                   jmp    804841e <fun+0x2a>
 8048408: 8b 45 08                mov    0x8(%ebp),%eax
 804840b: 8b 18                   mov    (%eax),%ebx
 804840d: 8b 45 08                mov    0x8(%ebp),%eax
 8048410: 8b 40 04                mov    0x4(%eax),%eax
 8048413: 89 04 24                mov    %eax,(%esp)
 8048416: e8 d9 ff ff ff          call   80483f4 <fun>
 804841b: 8d 04 03                lea    (%ebx,%eax,1),%eax
 804841e: 83 c4 04                add    $0x4,%esp
 8048421: 5b                      pop    %ebx
 8048422: 5d                      pop    %ebp
 8048423: c3                      ret
```

The program makes the following procedure call `fun(0x0804a008)`. Prior to the call (i.e., immediately BEFORE the execution of the `call` instruction) the `%esp=0xffffd400`, `%ebp=0xffffd428`, and the return address in the caller is `0x804847c`.

You are also given the following values in memory:

Address	Value
0x0804a008	0x0000000f
0x0804a00c	0x0804a010
0x0804a010	0x000000d5
0x0804a014	0x00000000

The call `fun(0x0804a008)` will result in the following function invocations: `fun(0x0804a008)`, `fun(0x0804a010)`, and `fun(0)`. Fill in the stack diagram with the values that would be present immediately BEFORE the call instruction that invokes `fun(0)` (i.e., AFTER the execution of the `mov %eax,(%esp)` instruction).

- Use the actual values whenever possible, rather than variable/register names.
- For a register whose value is unknown, simply note the register name.
- Cross out each empty box for which there is insufficient information to fill in its value.

Stack Address	Value
0xffffd404	//////////
0xffffd400	0x0804a008
0xffffd3fc	**0x0804847c**
0xffffd3f8	**0xffffd428**
0xffffd3f4	**old** `%ebx` **or** `%ebx`
0xffffd3f0	**0x0804a010**
0xffffd3ec	**0x0804841b**
0xffffd3e8	**0xffffd3f8**
0xffffd3e4	**0x0000000f or 15**
0xffffd3e0	**0x00000000 or 0**
0xffffd3dc	//////////

Problem 7: Structures and Alignment (23 pts)

Consider the following code defining a struct, for use on a **64-bit Linux** system.

```
struct stats {
    int num_views;
    short sum;
};

struct system_f {
    char a;
    int *b;
    int c[3];
    long d;
    struct stats e;
    short f;
};
```

A) [7 pts] Show how a `struct system_f` would be laid out in memory, given x86-64 alignment requirements in Linux. Fill in the block diagram by marking each box with the name of the structure member, and mark any wasted space with an X. Clearly mark the end of the struct.

```
0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| a | X | X | X | X | X | X | X | b | b | b | b | b | b | b | b |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| c | c | c | c | c | c | c | c | c | c | c | c | X | X | X | X |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| d | d | d | d | d | d | d | d | e | e | e | e | e | e | e | e |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| f | f | X | X | X | X | X | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Suppose that in order to stop Dr. Grave O'Dangeron, this struct needs to be used on an embedded system where memory is scarce.

B) [5 pts] Give an alternative definition of `struct system_f` that saves as much space as possible. Feel free to use the box diagram below as scratch space, but please remember to populate the struct to receive full credit.

```
struct system_f {
```

One possibility:

```
    int *b;
    long d;
    int c[3];
    strruct stats e;
    short f;
    char a;
```

```
};
```

```
 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| b | b | b | b | b | b | b | b | d | d | d | d | d | d | d | d |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| c | c | c | c | c | c | c | c | c | c | c | c | e | e | e | e |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| e | e | X | X | f |f  | a | X |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

C) [1 pts] How many bytes were wasted due to alignment conventions in the first, naive definition?

17[1]

D) [1 pts] How many bytes were wasted due to alignment conventions in the second, improved definition?

1[2]

E) [1 pts] How would you determine how large `struct system_f` would be at runtime in C (including padding requirements)? Write a simple C expression below:

`sizeof(struct system_f)`

[1] Or, 19: Depending on whether the struct stats is seen as opaque

[2] Or, 3: Depending on whether the struct stats is seen as opaque

F) [8 pts] Unfortunately, Dr. Grave O'Dangeron deleted your C file with handy functions that retrieved data from pointers to `struct system_f`, and all you have left is a series of disassembled functions. Drat!

Write in the assembly functions for the first, naive definition of `struct system_f` that correspond to the C functions in the blanks below (don't worry about the types). Fill in the blanks on the left with the corresponding assembly addresses (i.e. `a1-a7`) on the right.

C Code

```
______a2______(struct system_f *s) {
  return s->a;
}

______a6______(struct system_f *s) {
  return s->c[2];
}

______a1______(struct system_f *s) {
  return s->e.sum;
}

______a7______(struct system_f *s) {
  return (*s).f;
}

______a4______(struct system_f *s) {
  return *(s->b);
}
```

x86-64 Assembly

```
00000000004004d0 <a1>:
  4004d0: movzwl 0x2c(%rdi),%eax
  4004d4: retq

00000000004004e0 <a2>:
  4004e0: movzbl (%rdi),%eax
  4004e3: retq

00000000004004f0 <a3>:
  4004f0: mov    0x8(%rdi),%rax
  4004f4: retq

0000000000400500 <a4>:
  400500: mov    0x8(%rdi),%rax
  400504: mov    (%rax),%eax
  400506: retq

0000000000400510 <a5>:
  400510: lea    0x10(%rdi),%rax
  400514: retq

0000000000400520 <a6>:
  400520: mov    0x18(%rdi),%eax
  400523: retq

0000000000400530 <a7>:
  400530: movzwl 0x30(%rdi),%eax
  400534: retq
```