Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Fall 2005

Exam 1

Tuesday October 11, 2005

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- 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 58 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (10):	
2 (12):	
3 (04):	
4 (05):	
5 (06):	
6 (05):	
7 (08):	
8 (08):	
TOTAL (58):	

Problem 1. (10 points):

Assume we are running code on a 7-bit machine using two's complement arithmetic for signed integers. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
int x = -16;
unsigned uy = x;
```

- You need not fill in entries marked with "–".
- TMax denotes the largest positive two's complement number and TMin denotes the smallest negative two's complement number.
- Hint: Be careful with the promotion rules that C uses for signed and unsigned ints.

Expression	Decimal Representation	Binary Representation
–	-2	
–		`001 0011`
x		
uy		
$x - uy$		
TMax + 1		
TMin - 1		
-TMin		
TMin + TMin		
TMax + TMin		

Problem 2. (12 points):

Consider the following two 7-bit floating point representations based on the IEEE floating point format. Neither of them have sign bits—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 3$ exponent bits. The exponent bias is 3.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 3$ fraction bits.

Numeric values are encoded in both of these formats as a value of the form $V = M \times 2^E$, where E is 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).

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If rounding is necessary you should *round upward*. In addition, give the values of numbers given by the Format A and Format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., $17/64$).

Format A		Format B	
Bits	Value	Bits	Value
`011 0000`	1	`0111 000`	1
`101 1110`			
`010 1001`			
`110 1111`			
`000 0001`			

Problem 3. (4 points):

This problem will test your knowledge of buffer overflows. In Lab 3, you performed an overflow attack against a program that read user input. The input was read by `getbuf()` and your goal was to create an exploit string that called `smoke()`.

```
int getbuf()
{
    char buf[32];
    Gets(buf);
    return 1;
}

void smoke()
{
    printf(‘‘Smoke!: You called smoke()\n’’);
    validate(0);
    exit(0);
}
```

Creating a workable exploit string against a program like the `bufbomb` usually requires converting the executable file into human readable assembly (using `objdump`) and generating a sequence of raw, often unprintable, bytes (using a program like `hex2raw`).

However, with the `bufbomb`, you may have noticed that any 40 character string will result in `smoke()` being called.

```
unix> ./bufbomb -t ngm
Type string:It is easy to love 213 when you’re a TA.
Smoke!: You called smoke()
VALID
NICE JOB!
```

A. Why will any 40-character string result in `smoke()` being called?

The following information may help you in answering this question. Hints:

- Recall that `getbuf()` is called from `test()`.
- Also recall that C strings are always terminated by the NULL character.

```
0000000000400f66 <test>:
...
  400f72:       b8 00 00 00 00          mov    $0x0,%eax
  400f77:       e8 54 00 00 00          callq  400fd0 <getbuf>
  400f7c:       89 c2                   mov    %eax,%edx
...

0000000000400f00 <smoke>:
  400f00:       48 83 ec 08             sub    $0x8,%rsp
  400f04:       bf 1c 25 40 00          mov    $0x40251c,%edi
  400f09:       e8 fa fe ff ff          callq  400e08 <puts@plt>
  400f0e:       bf 00 00 00 00          mov    $0x0,%edi
  400f13:       e8 0c 07 00 00          callq  401624 <validate>
  400f18:       bf 00 00 00 00          mov    $0x0,%edi
  400f1d:       e8 76 fe ff ff          callq  400d98 <exit@plt>

0000000000400fd0 <getbuf>:
  400fd0:       48 83 ec 28             sub    $0x28,%rsp
  400fd4:       48 89 e7                mov    %rsp,%rdi
  400fd7:       e8 ff 00 00 00          callq  4010db <Gets>
  400fdc:       b8 01 00 00 00          mov    $0x1,%eax
  400fe1:       48 83 c4 28             add    $0x28,%rsp
  400fe5:       c3                      retq
```

Problem 4. (5 points):

Consider the code below, where L, M, and N are constants declared with `#define`.

```
int array1[L][M][N];
int array2[M][N][L];

int copy(int i, int j, int k)
{
        array1[i][j][k] = array2[j][k][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
        movslq  %edi,%rdi
        movslq  %esi,%rsi
        movslq  %edx,%rdx
        movq    %rdi, %rax
        salq    $5, %rax
        addq    %rdi, %rax
        addq    %rsi, %rax
        leaq    (%rsi,%rsi,8), %rsi
        leaq    (%rdx,%rax,2), %rax
        leaq    (%rdx,%rdx,8), %rdx
        leaq    (%rdx,%rsi,2), %rsi
        addq    %rdi, %rsi
        movl    array2(,%rsi,4), %edx
        movl    %edx, array1(,%rax,4)
        ret
```

What are the values of `L`, `M`, and `N`?

```
L =

M =

N =
```

Problem 5. (6 points):

Consider the following C function and its corresponding x86-64 assembly code:

```
int foo(int x, int i)
{
  switch(i)
  {
    case 1:
      x -= 10;
    case 2:
      x *= 8;
      break;
    case 3:
      x += 5;
    case 5:
      x /= 2;
      break;
    case 0:
      x &= 1;
    default:
      x += i;
  }
  return x;
}
```

```
00000000004004a8 <foo>:
  4004a8:       mov    %edi,%edx
  4004aa:       cmp    $0x5,%esi
  4004ad:       ja     4004d4 <foo+0x2c>
  4004af:       mov    %esi,%eax
  4004b1:       jmpq   *0x400690(,%rax,8)
  4004b8:       sub    $0xa,%edx
  4004bb:       shl    $0x3,%edx
  4004be:       jmp    4004d6 <foo+0x2e>
  4004c0:       add    $0x5,%edx
  4004c3:       mov    %edx,%eax
  4004c5:       shr    $0x1f,%eax
  4004c8:       lea    (%rdx,%rax,1),%eax
  4004cb:       mov    %eax,%edx
  4004cd:       sar    %edx
  4004cf:       jmp    4004d6 <foo+0x2e>
  4004d1:       and    $0x1,%edx
  4004d4:       add    %esi,%edx
  4004d6:       mov    %edx,%eax
  4004d8:       retq
```

Recall that the gdb command `x/g $rsp` will examine an 8-byte word starting at address in `$rsp`. Please fill in the switch jump table as printed out via the following gdb command:

```
>(gdb) x/6g 0x400690

0x400690:   0x____________________     0x____________________

0x4006a0:   0x____________________     0x____________________

0x4006b0:   0x____________________     0x____________________
```

Problem 6. (5 points):

Consider the following function's assembly code:

```
0040050a <bar>:
  40050d:       b9 00 00 00 00          mov    $0x0,%ecx
  400512:       8d 47 03                lea    0x3(%rdi),%eax
  400515:       83 ff ff                cmp    $0xffffffffffffffff,%edi
  400518:       0f 4e f8                cmovle %eax,%edi
  40051b:       89 fa                   mov    %edi,%edx
  40051d:       c1 fa 02                sar    $0x2,%edx
  400520:       85 d2                   test   %edx,%edx
  400522:       7e 14                   jle    400538 <bar2+0x2b>
  400524:       8d 42 03                lea    0x3(%rdx),%eax
  400527:       83 fa ff                cmp    $0xffffffffffffffff,%edx
  40052a:       0f 4f c2                cmovg  %edx,%eax
  40052d:       89 c2                   mov    %eax,%edx
  40052f:       c1 fa 02                sar    $0x2,%edx
  400532:       ff c1                   inc    %ecx
  400534:       85 d2                   test   %edx,%edx
  400536:       7f ec                   jg     400524 <bar2+0x17>
  400538:       89 c8                   mov    %ecx,%eax
  40053a:       c3                      retq
```

Please fill in the corresponding C code:

```
int bar(int x)
{
  int y = 0;
  int z = ________;

  for( ; ________ ; ________)
  {
    z = ________;
  }

  return ________;
}
```

Problem 7. (8 points):

Consider the following data structure declarations:

```
struct alpha {
    int array[3];
    int i;
};
```

Below are four C and four x86-64 functions. Next to each of the x86-64 functions, write the name of the C function that it implements.

```
int *jan(struct alpha *p)
{
    return &p->i;
}

int feb(struct alpha *p)
{
    return p->i;
}

int mar(struct alpha *p)
{
    return p->array[p->i];
}

int *apr(struct alpha *p)
{
    return &p->array[p->i];
}
```

```
movslq  12(%rdi),%rax
leaq    (%rdi,%rax,4), %rax
ret
```

```
leaq    12(%rdi), %rax
ret
```

```
movl    12(%rdi), %eax
ret
```

```
movslq  12(%rdi),%rax
movl    (%rdi,%rax,4), %eax
ret
```

Problem 8. (8 points):

Consider the following C declarations:

```
typedef struct Order {
  char id;
  short code;
  float amount;
  char name[3];
  long data;
  char initial;
  struct Order *next;
  char address[5];
} Order;

typedef union {
  unsigned int      value;
  char              buf[20];
  Order             new_order;
} Union_1;
```

A. Using the templates below (allowing a maximum of 64 bytes), indicate the allocation of data for the Order struct `Order`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used. Assume the 64 bit alignment rules discussed in class.**

`Order:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How many bytes are allocated for an object of type `Union_1`?

(a) `sizeof(Union_1) =` ________________

Now consider the following C code fragment:

```
void init(Union_1 *u)
{
    /* This will zero all the space allocated for *u */
    bzero((void *)u, sizeof(Union_1));

    strcpy(u->buf, "Hello World");
    strcpy(u->new_order.name, "SM");

    printf("Output #1 is u->value             = %x", u->value);
    printf("Output #2 is u->buf               = %s", u->buf);

    u->new_order.code = 256;

    printf("Output #3 is u->buf               = %s", u->buf);

    /* ’H’ = 0x48  ’e’ = 0x65  ’l’ = 0x6c   ’o’ = 0x6f
       ’W’ = 0x57  ’r’ = 0x72  ’d’ = 0x64   ’S’ = 0x63
       ’M’ = 0x4d  space = 0x20 */
```

After this code has run, please complete the output given below. Assume that this code is run on a Little-Endian machine such as a Linux/x86-64 machine. **Be careful about byte ordering!**

C. (a) `Output #1 is u->value =`

(b) `Output #2 is u->buf =`

(c) `Output #3 is u->buf =`