CMSC 414 Self-Test Exercise

About the self-test

This self-test is designed to help you think about which skills from CMSC 216 (or related work) you need to brush up on for 414. In particular, for Project 1 (Buffer overflows) you will need a good working knowledge of C, pointer arithmetic, memory layout, and stack for C function calls. Some familiarity with x86 assembly language and with debuggers (especially gdb) will also be useful.

Thus, the questions in this exercise focus on decoding simple assembly language sequences, C memory layout, and pointer arithmetic. We will not be grading this exercise; this is purely to give you a sense of where you might need to do some review before Project 1. Lectures for days 2-4 (or so) will discuss these topics as well, and you’ll get more from the lectures if you remind yourself about some relevant material ahead of time. We will review some of this material in class as well, but reviewing it ahead of time can only help you.

If you have a little trouble with these problems, it doesn’t mean that you should give up on 414, but it does mean that you should do some review and expect to spend some extra time on Project 1. If you have a lot of trouble with these problems, you might want to talk to your instructor and TAs sooner rather than later.

These problems are borrowed from Dave O’Hallaron and Dave Eckhart, who teach Intro to Systems and OS (15-213 and 15-410) at Carnegie Mellon University.
Problem 1.
In this problem you will be given the task of reconstructing C code based on the declarations of C structures and unions provided below and the assembly code on the next page generated from the C code by a Linux/IA32 C compiler.

```c
struct s1 {
    char a[3];
    union u1 *b;
    int c;
};

struct s2 {
    struct s1 d;
    struct s1 *e;
    struct s2 *f;
    double g;
    int h[4];
};

union u1 {
    int i;
    struct s2 j;
    struct s1 *k;
};
```
For each IA32 assembly code sequence below on the left, fill in the missing portion of the corresponding C source line on the right.

A  proc1:  
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 4(%eax),%eax
    movl 40(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret

   int procl(struct s1 *x)
    {
        return x->___________________ ;
    }

B  proc2:  
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 32(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret

   int proc2(struct s2 *x)
    {
        return x->___________________ ;
    }

C  proc3:  
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl (%eax),%eax
    movl 4(%eax),%eax
    movl (%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret

   int proc3(union u1 *x)
    {
        return x->___________________ ;
    }
Problem 2.

This problem tests your understanding of pointer arithmetic and pointer dereferencing.

Harry Q. Bovik, as part of his systems course work, was asked to write a dynamic memory allocator (malloc(), free(), etc.). Harry is having trouble figuring out how to write a C macro to access part of his data structure, and he is asking you for help.

The following is a description of Harry’s memory-block structure:

<table>
<thead>
<tr>
<th>HDR</th>
<th>ID_STRING</th>
<th>PAYLOAD</th>
<th>FTR</th>
</tr>
</thead>
</table>

- HDR - Header of the block (4 bytes)
- ID_STRING - Unique ID string (8 bytes)
- PAYLOAD - Payload of the block (arbitrary size)
- FTR - Footer of the block (4 bytes)

The size of the payload of each block is stored in the header and the footer of the block. Since there is an 8 byte alignment requirement, the least significant of the 3 unused bits is used to indicate whether the block is free (0) or allocated (1). Harry has also decided to uniquely label each block with a string stored right after the header of the block. The size of this ID field is 8 bytes.

For this problem, you can assume that:

- sizeof(int) == 4 bytes
- sizeof(char) == 1 bytes
- sizeof(short) == 2 bytes
- The size of any pointer (e.g. char*) is 4 bytes.
Your task is to help Harry figure out and circle clearly which of the following definitions of the macro GET_ID will cause print_block() to output the string that is stored in the ID_STRING field. There may be multiple macros that are correct, so be sure to circle all of them.

Also, assume that the block pointer bp points to the first byte of the payload.

```c
/* Harry Q. Bovik’s print_block() function
   Refer to this function in order to figure out the context in which the GET_ID macro is used.
*/
void print_block(void *bp){
    printf("Found block ID: %s\n", GET_ID(bp));
}

/* A. */
#define GET_ID(bp) ((char *)(((int)(bp)) - 8))

/* B. */
#define GET_ID(bp) ((char *)(((char)(bp)) - 8))

/* C. */
#define GET_ID(bp) ((char *)(((char *)(bp)) - 4))

/* D. */
#define GET_ID(bp) ((char *)(((char *)(bp)) - 8))

/* E. */
#define GET_ID(bp) ((char *)(((int *)(bp)) - 4))

/* F. */
#define GET_ID(bp) ((char *)(((int *)(bp)) - 8))

/* G. */
#define GET_ID(bp) ((char *)(((char**) (bp)) - 8))

/* H. */
#define GET_ID(bp) ((char *)(((short *) (bp)) - 4))

/* I. */
#define GET_ID(bp) ((char *)(((short *) (bp)) - 8))
```
**Problem 3.**

This problem tests your understanding of stack discipline.

Consider the following C code and its corresponding 32-bit x86 machine code. Please complete the stack diagram on the following page.

```c
int fact(int n) {
    if (n == 1)
        return n;
    else
        return n * fact(n-1);
}
```

```assembly
080483a4 <fact>:
  80483a4: 55  push %ebp
  80483a5: 89 e5  mov %esp,%ebp
  80483a7: 53  push %ebx
  80483a8: 83 ec 04  sub $0x4,%esp
  80483ab: 8b 5d 08  mov 0x8(%ebp),%ebx
  80483ae: 83 fb 01  cmp $0x1,%ebx
  80483b1: 74 0e  je 80483c1 <fact+0x1d>
  80483b3: 8d 43 ff  lea 0xffffffff(%ebx),%eax
  80483b6: 89 04 24  mov %eax,(%esp)
  80483be: 0f af d8  imul %eax,%ebx
  80483c1: 89 d8  mov %ebx,%eax
  80483c3: 83 c4 04  add $0x4,%esp
  80483c6: 5b  pop %ebx
  80483c7: 5d  pop %ebp
  80483c8: c3  ret
```
A. Draw a detailed picture of the stack, starting with the caller invoking \texttt{fact(4)}, and ending immediately \textbf{before} the call instruction that invokes \texttt{fact(2)}.

- The stack diagram should begin with the argument for \texttt{fact} that the caller has placed on the stack. To help you get started, we have given you the first one.
- Use the actual values for function arguments, rather than variable names. For example, use 3 or 2 instead of \texttt{n}.
- For callee-saved registers that are pushed to the stack, simply note the register name (e.g., \%ebx).
- Always label \%ebp and give its value when it is pushed to the stack, e.g., \texttt{old }\%ebp: 0xffff1400.

\textbf{Value of \%ebp when fact(4) is called: 0xffffd848}
\textbf{Return address in function that called fact(4): 0x080483e6}

\begin{verbatim}
Stack
+-----------------------------------+
| 0xffffd830 | 4 | +-----------------------------------+
| 0xffffd82c | | +-----------------------------------+
| 0xffffd828 | | +-----------------------------------+
| 0xffffd824 | | +-----------------------------------+
| 0xffffd820 | | +-----------------------------------+
| 0xffffd81c | | +-----------------------------------+
| 0xffffd818 | | +-----------------------------------+
| 0xffffd814 | | +-----------------------------------+
| 0xffffd810 | | +-----------------------------------+
+-----------------------------------+
\end{verbatim}

\textbf{B. What is the final value of \%ebp, immediately \textbf{before} execution of the instruction that calls \texttt{fact(2)}?}

\%ebp=0x_____________________

\textbf{C. What is the final value of \%esp, immediately \textbf{before} execution of the instruction that calls \texttt{fact(2)}?}

\%esp=0x_____________________