Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other electronic devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 6 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name Solution

printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed

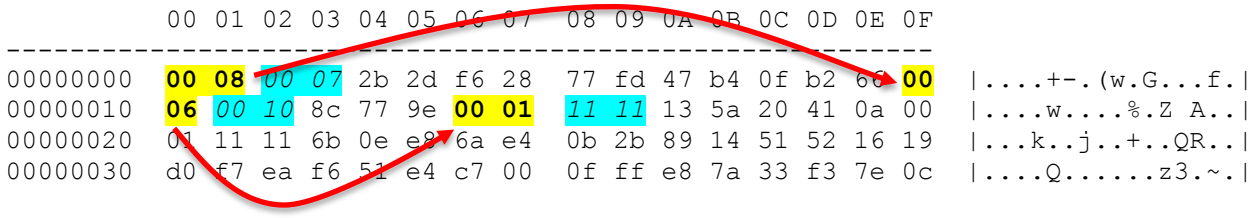

OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE', LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING.
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

xkcd.com

1. **[12 points]** Recall the `Untangle` assignment from earlier in the semester. In this question you will write a similar C function where you iterate through a series of records in a memory block.

Each record is made up of **two signed 16 bit integer values**. In the example memory dump below the first integer in each record is **bolded** and the second is *italicized*. The first record starts at the first byte of the memory region pointed to by `pBuffer`. To get to the next record, your function should **sum** the two signed integer values to obtain the next offset relative to `pBuffer`. You program should continue processing records until the two integers **sum to zero**, you may assume the two integers will never sum to a negative number.

As an additional complication, these numbers are **stored using big endian byte ordering** (so the first two bytes should be interpreted as `0x0008`, rather than `0x0800`). This means you'll have to manually reorder the bytes before summing the values.



```
          00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
-------------------------------------------------------------
00000000  00 08 00 07 2b 2d f6 28  77 fd 47 b4 0f b2 66 00  |....+-.(w.G...f.|
00000010  06 00 10 8c 77 9e 00 01  11 11 13 5a 20 41 0a 00  |....w....%.Z A..|
00000020  01 11 11 6b 0e e8 6a e4  0b 2b 89 14 51 52 16 19  |...k..j..+..QR..|
00000030  d0 f7 ea f6 51 e4 c7 00  0f ff e8 7a 33 f3 7e 0c  |....Q......z3.~.|
```

```
/**
 * Pre:  pBuffer points to a region of memory formatted as specified.
 *
 * Returns: The sum of all of the offsets parsed from the memory region.
 *
 * Restrictions: You must use pointer arithmetic in this function.
 *               Array brackets are not permitted.
 */
uint32_t miniUntangle(const uint8_t* pBuffer) {

        uint32_t total = 0;
        int16_t offset = 0;
        int16_t one, two, upper, lower;

        do
        {
                /* Get the first 16 bit integer */
                upper = *(pBuffer + offset) << 8;
                lower = *(pBuffer + offset + 1);

                one = upper + lower;

                /* Get the second 16 bit integer */
                upper = *(pBuffer + offset + 2) << 8;
                lower = *(pBuffer + offset + 3);

                two = upper + lower;

                offset = one + two;
                total += offset;

        }
        while (offset != 0);

        return total;
}
```

2. [12 points] Implement the C function described below. Be sure to note the pre and post conditions, failure to meet these conditions will result in a low score.

```
/*   Creates an array containing numCopies of C string src.
 *
 *   Pre:   *Copies is uninitialized or NULL.
 *          src points to a properly formatted C string.
 *          numCopies is greater than 0.
 *
 *   Post:  *Copies points to a newly allocated and appropriately sized array.
 *          In the event allocation fails *Copies should be set to NULL.
 *
 *          For example:
 *
 *          char * many;
 *          manyCopies(&many, "Hello", 2);
 *
 *          In the event many is not NULL it would point to an array containing:
 *          {'H','e','l','l','o','\0','H','e','l','l','o','\0'}
 *
 *   Additional Info:
 *          You may make use of any function declared in string.h
 */
void manyCopies(char ** Copies, const char * const src, uint32_t numCopies) {

   uint32_t offset = 0;
   *Copies = malloc((strlen(src) + 1) * numCopies);

   if (*Copies == NULL)
   {
          return;
   }

   for (int c = 0; c < numCopies; c++)
   {
          strcpy(*Copies + offset, src);
          offset += strlen(src) + 1;
   }

 }
```

3. Consider the following short C function.

```
#define DATAMAX 255

/* Process the data inside of a buffer.
 * Pre:
 *  Buffer is initialized.
 *
 * Post:
 *  A pointer to the result is returned.
 */
uint8_t * Q3(uint8_t Buffer[]) {
    uint32_t count = 0;
    uint8_t Result[DATAMAX];

    while((count < DATAMAX) && count < sizeof(Buffer))
    {
        /* Process Buffer in some way, storing the result in Result */
        Result[count] = Buffer[count] * Buffer[count];
    }

    return Result;
}
```

a) **[10 points]** There are several bugs in this function, analyze the code and determine the location of these errors. You may assume the program is syntactically correct. **Be specific, vague answers will receive no credit.**

There are three issues:

While you can't return arrays in C, you can return a pointer to the content. That's what's happening here, we are returning a pointer and it's the correct type (uint8_t *). The problem is it points to local data on the stack that is deallocated at the end of the function. Accessing this memory is logic error even if the code doesn't crash or produce immediately incorrect results.

sizeof() won't tell you the size of an array. Even in the scope where the array was declared it tells you the number of bytes in the array not the number of elements. In this case sizeof(Buffer) would be the size of a pointer type (4 bytes on x86, 8 bytes x86_64).

count is never incremented inside of the while leading to an infinite loop.

b) **[10 points]** How could you fix the bugs while keeping the intended functionality? You may make any required modifications to the function.

For issue 1, you could dynamically allocate the array and return a pointer to it, or you could pass an additional array as a parameter.

For issue 2, you'll need to pass the size of Buffer to the function and use that instead of sizeof().

Increment count in the loop.

4. Consider the x86-32 translation of a short C function shown at right.

Assume that all parameters and local variables are of type `int` or `int*`.

a) **[6 points]** State the stack address at which each parameter is stored (e.g., `%ebp - 48`).

%ebp + 8 # line 13
%ebp + 12 # line 5
%ebp + 16 # line 15
%ebp + 20 # line 20

b) **[6 points]** State the stack address at which each local variable is stored (e.g., `%ebp - 48`).

%ebp - 4 # 8
%ebp - 8 # 7
%ebp - 12 # 4

```
    . . .
Q4:
    pushl   %ebp                #  1
    movl    %esp, %ebp          #  2
    subl    $16, %esp           #  3
    movl    $0, -12(%ebp)       #  4
    movl    12(%ebp), %eax      #  5
    subl    $1, %eax            #  6
    movl    %eax, -8(%ebp)      #  7
    movl    $0, -4(%ebp)        #  8
    jmp     .L2                 #  9
.L4:                            # 10
    movl    -12(%ebp), %eax     # 11
    sall    $2, %eax            # 12
    addl    8(%ebp), %eax       # 13
    movl    (%eax), %eax        # 14
    cmpl    16(%ebp), %eax      # 15
    jle     .L3                 # 16
    movl    -12(%ebp), %eax     # 17
    sall    $2, %eax            # 18
    addl    8(%ebp), %eax       # 19
    movl    20(%ebp), %edx      # 20
    movl    %edx, (%eax)        # 21
    jmp     .L2                 # 22
.L3:                            # 24
    addl    $1, -4(%ebp)        # 25
.L2:                            # 26
    movl    -12(%ebp), %eax     # 27
    cmpl    -8(%ebp), %eax      # 28
    jle     .L4                 # 29
    movl    -4(%ebp), %eax      # 30
    leave                       # 31
    ret                         # 32
    . . .
```

c) **[6 points]** Examine the given x86-32 code. If there are any selection control structures in the code, for each such structure, state what the structure is (`if` or `if-else`) what range of statements (e.g., lines 17 through 23) belong to that structure (including any relevant labels and the Boolean test)?

The hallmark of an if or if-else is a forward branch; which we see in #16 and #22. Since the branch in #16 takes us just past the branch in #22, this is an if-else. The Boolean test for the if is in #15; the setup for the test is in lines #11 - #14. The jump in #22 takes us to #26, which marks the end of the structure.

So, there is an if-else in lines #11 - #26.

d) **[6 points]** Examine the given x86-32 code. If there are any loops in the code, for each loop, state what the structure is (`while` or `do-while`) what range of statements (e.g., lines 17 through 23) belongs to the body of the loop (including any relevant labels and the loop test)?

The hallmark of a do-while loop is a backward jump, which we find in #29; a while loop will also have a forward jump to the beginning of the loop test (which will be at the end of the loop body); we find that forward jump in #9. There is no setup for the forward jump since it is not a conditional jump.

e) **[4 points]** Examine the given x86-32 code. How many bytes are in the stack frame for `Q4`, including the backed up value for the frame pointer `ebp`? Justify your answer.

The frame pointer, %ebp, occupies 4 bytes, and is backed up in #1. The adjustment of the stack pointer, %esp, occurs in line #3; that adds 16 bytes to the frame.

So, the frame occupies 20 bytes.

5. A developer has an executable file that contains a C function and a `main()` function that calls it, but doesn't know much about the function except that it is named `mystery()`. So, she tries a gdb analysis. A partial transcript follows:

```
CentOS > gdb mdriver

(gdb) break mystery
Breakpoint 1 at 0x80483ca

(gdb) run
Breakpoint 1, 0x080483ca in mystery ()

(gdb) disassem
Dump of assembler code for function mystery:
   0x080483c4 <+0>:    push   %ebp
   0x080483c5 <+1>:    mov    %esp,%ebp
   0x080483c7 <+3>:    sub    $0x10,%esp
=> 0x080483ca <+6>:    mov    0xc(%ebp),%eax    eax = *(ebp + 12) = param2
   0x080483cd <+9>:    imul   0xc(%ebp),%eax    eax = param2 * param2
   0x080483d1 <+13>:   mov    %eax,-0xc(%ebp)   local1 = param2 * param2
   0x080483d4 <+16>:   mov    0x8(%ebp),%eax    eax = *(ebp + 8) = param1
   0x080483d7 <+19>:   imul   0x10(%ebp),%eax   eax = param1 * param3
   0x080483db <+23>:   mov    %eax,-0x8(%ebp)   local2 = param1 * param3
   0x080483de <+26>:   shll   $0x2,-0x8(%ebp)   local2 = 4 * param1 * param3
   0x080483e2 <+30>:   mov    -0x8(%ebp),%eax   eax = local2 = 4 * param1 * param3
   0x080483e5 <+33>:   mov    -0xc(%ebp),%edx   edx = param2 * param2
   0x080483e8 <+36>:   mov    %edx,%ecx         ecx = param2 * param2
   0x080483ea <+38>:   sub    %eax,%ecx         ecx = param2 * param2 -
                                                      4 * param1 * param3
   0x080483ec <+40>:   mov    %ecx,%eax         set return value
   0x080483ee <+42>:   mov    %eax,-0x4(%ebp)      rather oddly...
   0x080483f1 <+45>:   mov    -0x4(%ebp),%eax
   0x080483f4 <+48>:   leave
   0x080483f5 <+49>:   ret
End of assembler dump.
```

a)

```
(gdb) print *(int*)($ebp + 8)
$1 = 5
(gdb) print *(int*)($ebp + 12)
$2 = 2
(gdb) print *(int*)($ebp + 16)
$4 = 3

(gdb) ni
0x080483cd in mystery ()
...

(gdb) disassem
   ...
   0x080483d1 <+13>:   mov    %eax,-0xc(%ebp)
=> 0x080483d4 <+16>:   mov    0x8(%ebp),%eax
   0x080483d7 <+19>:   imul   0x10(%ebp),%eax
   ...
```

b)

```
(gdb) print *(int*)($ebp - 0xc)
$5 = 4

(gdb) ni
0x080483d4 in mystery ()
...
```

```
(gdb) disassem
   ...
   0x080483de <+26>:  shll   $0x2,-0x8(%ebp)
=> 0x080483e2 <+30>:  mov    -0x8(%ebp),%eax
   0x080483e5 <+33>:  mov    -0xc(%ebp),%edx
   ...

...
(gdb) print *(int*)($ebp - 0x8)
$7 = 60

(gdb) ni
0x080483e5 in mystery ()
...

(gdb) disassem
   ...
   0x080483ea <+38>:   sub    %eax,%ecx
=> 0x080483ec <+40>:   mov    %ecx,%eax
   0x080483ee <+42>:   mov    %eax,-0x4(%ebp)
   ...

(gdb) print $ecx
$8 = -56

(gdb) ni
...

(gdb) disassem
   ...
   0x080483f1 <+45>:   mov    -0x4(%ebp),%eax
=> 0x080483f4 <+48>:   leave
   0x080483f5 <+49>:   ret

(gdb) print $eax
$10 = -56
```

For the following questions, label the parameters to `mystery()` as `P1`, `P2`, etc., in the order they would be listed in the C call to the function, and label the local variables L1, L2, etc., in the order they occur in the stack frame (from the top down).

Each question that follows refers to the part of the gdb session above that is labeled to match the question, but you may consider other parts of the gdb session in answering each question.

a) **[4 points]** Explain what the three print commands tell us about the function. You might want to consider the `disassem` output in your explanation.

These show us the values of the three parameters that are passed into the function:

%ebp + 8 5
%ebp + 12 2
%ebp + 20 3

b) **[4 points]** What does the print command here tell us has just happened in the execution of the function? Consider the effect on and/or use of local variables and parameters.

Considering the preceding assembly code, this shows that a local variable (at %ebp – 0xC) has been set to the square of the second parameter, 4.

c) **[12 points]** Give an algebraic expression, in terms of the parameters to `mystery()`, for the value that is in `$eax` when the function reaches the `leave` instruction. Justify your answer by **showing work next to the disassembly on page 8**.

From the analysis on page 8:

ecx = param2 * param2 – 4 * param1 * param3

---

6. **[8 points]** The following x86 assembly code is part of a C function:

```
.L3:
     movl   -8(%ebp), %eax      load a local variable into eax
     xorl   %eax, %eax          xor it with itself
     movl   (%eax), %eax        load *eax into eax
     movl   %eax, -4(%ebp)
     movl   $0, %eax
     leave
     ret
```

Explain what would happen if this code were executed, and why.

From the analysis above, we see that eax is set to A xor A, for some unknown value A.

Now, A xor A equals 0, no matter what A is (see, Boolean algebra!).

Therefore, the third statement will dereference a NULL pointer, causing a segfault.