Final Exam Solutions

1. Floats (25)

A: 0 10011 00 in decimal

```
16 // 1.0 * 2^(19-15)
```

B: 0 10011 01 in decimal

```
20 // 1.25 * 2^(19-15)
```

C:

The difference would get larger as floats gets larger, because the increasingly large exponent multiplies the mantissa by a larger and larger amount each step.

D:

true // reasonable float arithmetic

true // extremely large number with extremely small number loses precision!

true // infinity plus itself by definition is itself

false // this float division is not the same when done in different orders because of precision loss

2. Generics (30)

Part 1: Generic Function

Sample Solution

```
bool insert_sorted(void *base, int nelems, int elem_size_bytes,
                   void *elem_to_insert, int (*cmp_fn)(void *, void *)) {

    // loop from left to right until we find the right position
    void *end = (char *)base + elem_size_bytes * nelems;
    for (int i = 0; i < nelems; i++) {
        void *elem = (char *)base + i * elem_size_bytes;
        if (cmp_fn(elem, elem_to_insert) > 0) {
            // insert the new element before this existing element
            memmove((char *)elem + elem_size_bytes, elem, (char *)end
                    - (char *)elem);
            memcpy(elem, elem_to_insert, elem_size_bytes);
            return true;
        } else if (cmp_fn(elem, elem_to_insert) == 0) {
            return false;
        }
    }

    // If we get here, we must insert the element at the very end
    memcpy(end, elem_to_insert, elem_size_bytes);
    return true;
}
```

Part 2: Comparison Function

Sample Solution

```
int compare_ints(void *a, void *b) {
    int ai = *(int *)a;
    int bi = *(int *)b;
    /* Note: while % behaves differently with negatives, in evaluating solutions
     * we allowed programs to assume that % 2 always returns 0 or +1.
     */
    if (ai % 2 == bi % 2) {
        return abs(ai) - abs(bi);
    } else {
        if (ai % 2 == 0) {
            return -1;
```

```
        } else {
            return 1;
        }
    }
}
```

3. Assembly (45)

A: Foo

Sample Solution

```
int foo(int n, char *input) {
    int x = strlen(input + 1);
    int sum = x + n / 4;
    for (int i = 0; i < x; i++) {
        sum += 2 * i;
    }

    if (strchr(input, 'a') == NULL) {
        input[2] = '\0';
        sum = 0;
    }

    return sum;
}
```

B: Caller-owned registers are being pushed and popped at the start and end of this function because they are guaranteed to the caller to be preserved across function calls so we must preserve them if we want to use them.

C: call is used to call functions, and jmp is used for control flow such as loops and conditional statements. To call functions, call stores the return address, the saved %rip, on the stack; jmp does not do this because it does not resume execution back where it originally jumped.

D: The C line that changes is the x + n / 4 line. It changes by removing the logic for the "fixup" for division, because the number is unsigned, so it no longer needs to handle the special case of adding for a negative number before dividing. Thus, the lea, test and cmov at 400608-400610 will go away, and the shift following it will become a logical shift.

4. Gaming The System (30)

A: The vulnerability is that the loop lowercasing each character iterates for the length of the *original* input string, rather than the shortened string. This means that if the inputted name is longer than the length limit (8 – we also accepted 7 because of the inconsistent code comment) then that loop will continue lowercasing characters beyond the string buffer. From the GDB run, we can see that the wallet amount lives 12 bytes above the short name buffer in memory. If we enter a name longer than 12 characters, we will thus "lowercase" the bytes of wallet. 65 is the ASCII representation for 'A', so when we loop over that 65 and overwrite with the lowercase version, we will write back 'a', which is 97!

Thus, the exploit is to enter any name longer than 12 characters. (Note that if you enter an extremely long name, you will likely corrupt vital stack data like the return address. We generally did not deduct points for not mentioning an upper limit to the length).

B: The fix is to change the lowercasing loop's condition to instead be dependent on the length of the short name rather than the length of the original name. This will prevent it from looping too far and going outside the bounds of the short name. For example, you could instead use i < strlen(short_name) or i < sizeof(short_name) – 1.

5. Heap Allocators (50)

A:

```
void set_header_status(block_header *header, int status) {
    *header &= (-1L << 2);    // must zero out first!
    *header |= status;
}
```

B:

```
void* get_payload(block_header *header) {
    return header + 1;
    // or return (char *)header + sizeof(header);
}
```

C:

```
void set_previous_free_ptr(block_header *header1, block_header *prev) {
    ((free_payload *)get_payload(header1))->prev = prev;
}

void set_next_free_ptr(block_header *header1, block_header *next) {
    ((free_payload *)get_payload(header1))->next = next;
}
```

D:

```
void add_temp_block_to_freelist(block_header *header) {
    // should point to the tail as its previous element
    set_previous_free_ptr(header, free_list_tail);

    // should point to NULL as its next element
    set_next_free_ptr(header, NULL);

    // If there is already a tail, have it point to this new element
    if (free_list_tail) {
        set_next_free_ptr(free_list_tail, header);
    } else {
        // Neither a head nor tail exist, so we must update the head
        free_list_head = header;
    }
```

```
    // This element is the new tail
    free_list_tail = header;
}
```

E:

```
void *mytalloc(size_t requested_size) {
    void *memory = mymalloc(requested_size);
    if (memory == NULL) return NULL;

    block_header *header = (block_header *)memory - 1;
    set_header_status(header, ALLOC_TEMP);

    add_temp_block_to_freelist(header);

    return memory;
}
```