

IMPORTANT: Explain your answer briefly. Do not just write a short answer or fill the assembly code.

1. [10pts] The following code fragment has a potential vulnerability.

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

- A. [4pts] Write possible values for `ele_cnt` and `ele_size` to crash the application/system. Explain the reason.

- B. [6pts] Write extra C codes to add before the `malloc` call to prevent the crash.

2. [10pts][floating point data representation]

- A. [4pts] Assume variables x, f, and d are of type int, float, and double, respectively. (Neither f nor d equals to +infinity, -infinity, or NaN). For each of the following expressions, either argue that it is always true or give a counterexample if it is not.

A-1) $x == (\text{int}) (\text{double}) x$

A-2) $f == -(-f)$

A-3) $1.0/2 == 1/2.0$

A-4) $d*d \geq 0.0$

A-5) $(f+d) - f == d$

- B. [3pts] Write the rounded binary numbers for the following values. They should be rounded to nearest 1/4 (2 bits right of binary point, and must use “round-to-even” rule.

Explain the advantage of such “round-to-even” rule, compared to round-down or round-up.

10.00 <u>011</u>	=>	_____
10.00 <u>110</u>	=>	_____
10.11 <u>100</u>	=>	_____
10.10 <u>100</u>	=>	_____

- C. [3pts] Explain how a floating point compare instruction (fcmp) can be implemented for the IEEE fp format. How will it be different from the integer compare instruction (cmp)?

3. [8pts] Answer the two questions for the following C function and the corresponding assembly code.

```
long rfun(unsinged long x) {  
    if ( _____ )  
        return _____;  
  
    unsigned long nx = _____;  
    long rv = rfun(nx);  
    return _____;  
}
```

```
rfun:  
    pushq    %rbx  
    moveq    %rdi, %rbx  
    movl     $0, %eax  
    testq    %rdi, %rdi  
    je       .L2  
    shrq     $2, %rdi  
    call     rfun  
    addq     %rbx, %rax  
.L2:  
    popq     %rbx  
    ret
```

A. [3pts] What value in the C code does rfun store in %rbx?

B. [5pts] Fill in the missing expressions in the C code.

4. [7pts] The following two C code fragments show two different ways of supporting 2D data structures.

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};

int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3

int *univ[UCOUNT] = {mit, cmu,
ucb};

int get_univ_digit (size_t index,
size_t digit){

    return univ[index][digit];

}
```

A. [4pts] Complete the following assembly codes for the two access functions. (explain your answer)

```
1) _____
addl %rax, %rsi
movl 2) _____, %eax
ret
```

```
salq    $2, %rsi
addq    3) _____, %rsi
4) _____
ret
```

B. [3pts] Discuss the pros and cons of the two approaches.

5. [10pts] Find the minimum number of operations to implement the given functions.

- Points will not be given if functions are implemented more than your minimum number of operations.
- You should justify your solution by implementing the functions.
- Assignment operator '=' is legal. But it will not be counted as an operator.

(1) addOK [5pts]

Description	Determine if can compute x+y without overflow
Examples	addOK(0x80000000,0x80000000) = 0 addOK(0x80000000,0x70000000) = 1
Legal Ops	! ~ & ^ + << >>

The minimum number of operations to implement this function is _____.
<pre>int addOK(int, int) { } }</pre>

6. [10pts] Solve the following problems with the given assembly code.

C code
<pre>void phase(char *input) { int i; int numbers[6]; read_six_numbers(input, numbers); for(i = 1; i < 6; i++) { if (<input type="text"/>) explode_bomb(); } }</pre>

Assembly
<p>Dump of assembler code for function phase:</p> <pre>0x00005555555522a <+0>: push %rbp 0x00005555555522b <+1>: push %rbx 0x00005555555522c <+2>: sub \$0x28,%rsp 0x000055555555230 <+6>: mov %fs:0x28,%rax 0x000055555555239 <+15>: mov %rax,0x18(%rsp) 0x00005555555523e <+20>: xor %eax,%eax 0x000055555555240 <+22>: mov %rsp,%rbp 0x000055555555243 <+25>: mov %rsp,%rsi 0x000055555555246 <+28>: callq 0x555555555820 <read_six_numbers> 0x00005555555524b <+33>: mov %rsp,%rbx 0x00005555555524e <+36>: add \$0x14,%rbp 0x000055555555252 <+40>: jmp 0x55555555525d <phase+51> 0x000055555555254 <+42>: add \$0x4,%rbx 0x000055555555258 <+46>: cmp %rbp,%rbx 0x00005555555525b <+49>: je 0x55555555526f <phase+69> 0x00005555555525d <+51>: mov (%rbx),%eax 0x00005555555525f <+53>: lea 0x2(%rax,%rax,2),%eax 0x000055555555263 <+57>: cmp %eax,0x4(%rbx) 0x000055555555266 <+60>: je 0x555555555254 <phase+42> 0x000055555555268 <+62>: callq 0x5555555557fa <explode_bomb> 0x00005555555526d <+67>: jmp 0x555555555254 <phase+42> 0x00005555555526f <+69>: mov 0x18(%rsp),%rax 0x000055555555274 <+74>: xor %fs:0x28,%rax 0x00005555555527d <+83>: jne 0x555555555286 <phase+92> 0x00005555555527f <+85>: add \$0x28,%rsp 0x000055555555283 <+89>: pop %rbx 0x000055555555284 <+90>: pop %rbp 0x000055555555285 <+91>: retq 0x000055555555286 <+92>: callq 0x5555555554ed8 End of assembler dump.</pre>

A. Fill the C statement in the blank, based on the corresponding assembly codes. [5pts]

B. What is the solution if the first number of the solution is "1"? [5pts]

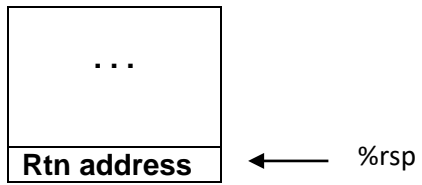
7. [7pts] Solve the following problems with the given C code and assembly code

C code
<pre>long mul(long *p,long val) { long x=*p; long y=x*val; *p=y; return x; } long call_mul(long x) { long v1=10; long v2=mul(&v1,300); return x+v2; }</pre>

Assembly
<pre>mul: movq (%rdi), %rax imul %rax, %rsi movq %rsi, (%rdi) ret call_mul: subq \$16, %rsp movq \$1000, 8(%rsp) <input type="text"/> <input type="text"/> call mul ← addq 8(%rsp), %rax addq \$16, %rsp ret</pre>

A. [3pts] Fill the assembly code in the blank(Assembly) [3pts]

- B. [4pts] Assume that the program starts with the `call_mul` function. Draw the stack frame when the program **finishes executing** `"call mul"` instructions.



Initial Stack Structure