This exam is based on the CS107 winter 2018 final exam by instructor Chris Gregg.

CS107 Practice Final Exam

This is a closed book, closed note, closed electronic device exam, except for one double-sided US-Letter-sized (8.5"x11") page of your own prepared notes which you may refer to during the exam. You have 180 minutes to complete all problems. You don't need to **`#include`** any header files. For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. Solutions that violate any specified restrictions may get partial credit. Style is secondary to correctness (e.g., there are no style deductions for using magic numbers). There is 1 point per minute of the exam.

Good luck!

SUNet ID (username): ________________**`@stanford.edu`**

First Name: ____________________

Last Name: ____________________

I accept the letter and spirit of the honor code.

[signed] ______________________________

Problems:

1. Generics	30 points
2. Reverse Engineering	36 points
3. Password Cracking	30 points
4. Heap Allocators	72 points
5. Compiler Optimizations	12 points
	180 points total

Problem 1: Generics [30 points]

Given a sorted `void *` array of elements, the **`remove_duplicates`** function will remove all duplicates in the array, returning the new size of the array. Your function should modify the array in-place (e.g., to remove a duplicate, move the rest of the array backwards over it):

```
int remove_duplicates(void *arr, size_t nelems,
                      int width, int (*cmp)(void *, void *));
```

For example, given the following code, and assuming a correct int comparison function:

```
int iarr[] = {1, 1, 1, 2, 2, 3, 3, 3, 4, 7, 8, 8};
int newsz = remove_duplicates(iarr,12,4,cmp_int);
```

`newsz` holds the value 6 and the first six elements of `iarr` are `{1, 2, 3, 4, 7, 8}`.

A) [20 points] Write the generic remove_duplicates function:

```
int remove_duplicates(void *arr, size_t nelems, int width,
    int (*cmp)(void *, void *)) {
    // YOUR CODE HERE
```

B) [5 points] Write a comparison function on longs that will work for **`remove_duplicates`** to sort in ascending order:

```
int cmp_long(void *p, void *q) {
    // YOUR CODE HERE
```

C) [5 points] Fill in the four blanks in main that will produce the following output when run from the command line:

```
$ ./remove_dup 1 1 1 2 2 3 3 3 4 7 8 8
1,1,1,2,2,3,3,3,4,7,8,8
1,2,3,4,7,8
```

Fill in the blanks below:

```
// function declaration for function written in 1b.
int cmp_long(void *p, void *q);

int main(int argc, char *argv[]) {
    int nelems = argc - 1;
    long arr[nelems];

    for (int i = 0; i < nelems; i++) {
        arr[i] = atol(argv[i+1]);
        printf("%ld",arr[i]);
        printf("%s", i == nelems - 1 ? "\n" : ",");
    }

    int newsz = remove_duplicates(_____________________________,  // 1

                                  _____________________________,  // 2

                                  _____________________________,  // 3

                                  _____________________________); // 4

    for (int i = 0; i < newsz; i++) {
        printf("%ld",arr[i]);
        printf("%s", i == newsz - 1 ? "\n" : ",");
    }
    return 0;
}
```

Problem 2: Reverse Engineering [36 points]

A) [14 points] Given the following assembly code, re-construct the C code that produced it.

```
0x400566 <+0>:   test   %rsi,%rsi
0x400569 <+3>:   je     0x400599 <mystery+51>
0x40056b <+5>:   push   %rbp
0x40056c <+6>:   push   %rbx
0x40056d <+7>:   sub    $0x8,%rsp
0x400571 <+11>:  mov    %rsi,%rbx
0x400574 <+14>:  mov    %rdi,%rbp
0x400577 <+17>:  shr    %rsi
0x40057a <+20>:  callq  0x400566 <mystery>
0x40057f <+25>:  mov    -0x8(%rbp,%rbx,8),%rsi
0x400584 <+30>:  mov    $0x400694,%edi
0x400589 <+35>:  mov    $0x0,%eax
0x40058e <+40>:  callq  0x400430 <printf@plt>
0x400593 <+45>:  add    $0x8,%rsp
0x400597 <+49>:  pop    %rbx
0x400598 <+50>:  pop    %rbp
0x400599 <+51>:  repz retq
```

```
void mystery(long *arr, size_t count) {

    if (___________________________) {  // line 1

        ______________________________; // line 2

        printf("%lu\n", _____________); // line 3

    }
}
```

B) [22 points] Given the following assembly code, re-construct the C code that produced it. Tip: the assembly looks long, but the pushing and popping is necessary due to the nature of the function; make sure you note where all of the arguments are moved to at the beginning of the assembly function.

```
0x40052f <+0>:   push   %r15
0x400531 <+2>:   push   %r14
0x400533 <+4>:   push   %r13
0x400535 <+6>:   push   %r12
0x400537 <+8>:   push   %rbp
0x400538 <+9>:   push   %rbx
0x400539 <+10>:  sub    $0x18,%rsp
0x40053d <+14>:  mov    %rdi,%r13
0x400540 <+17>:  mov    %rsi,%r14
0x400543 <+20>:  mov    %edx,0xc(%rsp)
0x400547 <+24>:  mov    %rcx,%r15
0x40054a <+27>:  mov    %rdi,%r12
0x40054d <+30>:  mov    $0x1,%ebp
0x400552 <+35>:  jmp    0x400574 <mystery+69>
0x400554 <+37>:  movslq 0xc(%rsp),%rbx
0x400559 <+42>:  imul   %rbp,%rbx
0x40055d <+46>:  add    %r13,%rbx
0x400560 <+49>:  mov    %rbx,%rsi
0x400563 <+52>:  mov    %r12,%rdi
0x400566 <+55>:  callq  *%r15
0x400569 <+58>:  test   %eax,%eax
0x40056b <+60>:  jns    0x400570 <mystery+65>
0x40056d <+62>:  mov    %rbx,%r12
0x400570 <+65>:  add    $0x1,%rbp
0x400574 <+69>:  cmp    %r14,%rbp
0x400577 <+72>:  jb     0x400554 <mystery+37>
0x400579 <+74>:  mov    %r12,%rax
0x40057c <+77>:  add    $0x18,%rsp
0x400580 <+81>:  pop    %rbx
0x400581 <+82>:  pop    %rbp
0x400582 <+83>:  pop    %r12
0x400584 <+85>:  pop    %r13
0x400586 <+87>:  pop    %r14
0x400588 <+89>:  pop    %r15
0x40058a <+91>:  retq
```

```
void *mystery(void *arr, size_t nelems, int width,
              int(*cmp)(void *, void *)) {

    void *x = ____________________________________________; // line 1

    for (_______________________________________________) { // line 2

        void *y = ________________________________________; // line 3

        if (____________________________________________) { // line 4

            ______________________________________________; // line 5

        }
    }

    return _______________________________________________; // line 6
}
```

Problem 3: Password Cracking [30 points]

The CIA has been told their system has a vulnerability, but they are not sure what it is, and have enlisted you to help find out. Specifically, they know of a case where someone with access to the system (who can run the program multiple times, but not in GDB) was able to get in without initially knowing the password. The CIA wants you to find out the steps they took to gain access.

You are given the following C code for the vulnerable program (see the exam reference sheet for prototypes of the four string library functions used). You can run the code on your local computer, but you don't have access to the `get_realpw` function, so you make that function up yourself for your testing. However, you do know that the password only changes once a day.

```c
// file: cia_login.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void get_realpw(char *pw) { // function made up by you
    strcpy(pw,"dummy password");
}

void authenticate(char *userpw) {
    char realpw[16];
    char userpwcopy[16];
    get_realpw(realpw);
    strncpy(userpwcopy, userpw, 16);

    if (strcmp(userpwcopy,realpw) == 0) {
        printf("Welcome to CIAnet!\n");
    } else {
        printf("Your password, %s, is incorrect.\n",userpwcopy);
    }
}

int main(int argc, char *argv[]) {
    char userpw[1024];
    printf("Password?\n");
    // read input from user
    fgets(userpw, 1024, stdin);
    // remove trailing newline
    userpw[strlen(userpw) - 1] = '\0';
    authenticate(userpw);
    return 0;
}
```

You run gdb on the binary file after you compile it. Here is the text of the gdb session:

```
$ gdb cia_login
The target architecture is assumed to be i386:x86-64
Reading symbols from cia_login...done.
(gdb) break 22
Breakpoint 1 at 0x40073a: file cia_login.c, line 22.
(gdb) run
Starting program: cia_login
Password?
abcdefg
Your password, abcdefg, is incorrect.

Breakpoint 1, authenticate (userpw=userpw@entry=0x7fffffffe580
"abcdefg") at cia_login.c:22
22 }
(gdb) p realpw
$1 = "dummy password\000"
(gdb) p userpw
$2 = 0x7fffffffe580 "abcdefg"
(gdb) p userpwcopy
$3 = "abcdefg\000\000\000\000\000\000\000\000"
(gdb) p &realpw
$4 = (char (*)[16]) 0x7fffffffe560
(gdb) p &userpw
Address requested for identifier "userpw" which is in register $rbx
(gdb) p/x $rbx
$7 = 0x7fffffffe580
(gdb) p &userpwcopy
$5 = (char (*)[16]) 0x7fffffffe550
(gdb) x/64bx &userpwcopy
0x7fffffffe550: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x00
0x7fffffffe558: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe560: 0x64 0x75 0x6d 0x6d 0x79 0x20 0x70 0x61
0x7fffffffe568: 0x73 0x73 0x77 0x6f 0x72 0x64 0x00 0x00
0x7fffffffe570: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe578: 0x7f 0x07 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe580: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x00
0x7fffffffe588: 0x00 0x0d 0xa1 0xf7 0xff 0x7f 0x00 0x00
```

Once you reach this point, you know how to break into the CIA computer!

A) [16 points] Write instructions to the CIA that they can follow to break into the system. (you do not need to explain how/why your exploit works – that is the next question).

B) [7 points] Explain how and why your exploit works.

C) [7 points] Give a 1-line fix (or modification) that will make the CIA program more secure, and explain why your fix eliminates the vulnerability.

Problem 5: Heap Allocators [72 points]

Consider a heap allocator implementation designed as follows – your task is to implement various helper functions and components to implement this allocator design:

- All blocks must be aligned on 4-byte boundaries, and the max request size is `INT_MAX`.
- The minimum payload size is 4 bytes, and payloads are always given multiples of 4 bytes (e.g., a malloc of 10 bytes would be given a payload of 12 bytes).
- Each block has a **<u>20-byte header</u>**: 4 bytes (int) payload size, a "next" pointer (pointer to the header of the next block in the free list or NULL for last element of the list), and a "previous" pointer (pointer to the header of the previous block in the free list or NULL for first element in the list), where the pointers are for use in a doubly-linked free list (these pointers are always present in the header— not payload—but are simply not used if the block is allocated).
- Each block has a **<u>4-byte footer</u>**, which is like a header but goes immediately after the payload instead of before it: 4 bytes (int) payload size.
- Because block sizes must be multiples of 4 bytes, the **rightmost (least significant) two bits of the payload size** in the header and footer are not really needed (they would always be zero). So, we use the least significant bit to indicate allocated or free (1=allocated, 0=free). The next bit is only used for allocated blocks (must be 0 for free blocks), and it indicates whether this block has been the subject of a call to realloc after its malloc (1=has been reallocated, otherwise 0). Therefore, **to find the actual payload size of an allocated block, these two bits must be zeroed out**. Note: both the header and footer have allocated and realloc bits.
- There is a single explicit free list implemented as an **unsorted linked list** of free blocks.
- To maintain alignment, the header of the first block in the heap segment **starts 4 bytes after the start of the heap segment**. The entire heap after that is always part of one or more blocks (initialization of heap segment starts with one giant free block that is split over time).

<u>Example 1</u>: An allocated block that was first malloced as 25 bytes, then subsequently realloced as 50 bytes would have an actual payload size of 52 bytes (closest multiple of 4 greater than or equal to 50), and the header and footer would record the payload size as 55 (two rightmost bits set to 1). The header would include unused next/prev pointers.

Example 2: For a free block with an actual payload size of 16 bytes, the header and footer would record the payload size as 16 (two rightmost bits set to 0). The header would include valid next/prev pointers.

Assume the following global typedefs, constants, and variables have already been set up – for this problem, you may assume that the memory layout for the **headerT** struct is in the order given here, with no padding:

```
typedef struct headerT {
   int payloadsz;
   struct headerT *next;
   struct headerT *prev;
} headerT;

#define MIN_SIZE 4
int roundup(int size, int mult); // rounds size up to multiple of mult

static void *heapStart; /* base address of entire heap segment */
static size_t heapSize; /* number of bytes in heap segment */
headerT *free_list;     /* front of the free list */
```

A) [6 points] Write a helper function that, given a pointer to a header or footer, reads and returns the actual payload size of that block, in bytes (i.e., returns the size with the two least significant bits zeroed out).

```
int get_size(void *curr) {
    // YOUR CODE HERE
```

B) [6 points] Write a helper function to identify when a block is allocated. Given a pointer to a header or footer, return true if the block is allocated, otherwise return false.

```
bool is_allocated(void *curr) {
    // YOUR CODE HERE
```

C) [6 points] Write a helper function to identify when a block has been reallocated since the time of its malloc. Given a pointer to a header or footer, return true if the block has been reallocated, otherwise return false.

```
bool is_reallocated(void *curr) {
    // YOUR CODE HERE
```

D) [8 points] Write a helper function that, given a pointer to the header of a block, returns the address of the header of the block to its right in memory. If there is none (i.e., at the boundary of the heap), return NULL.

```
headerT *right_block(headerT *curr) {
    // YOUR CODE HERE
```

E) [23 points] Write a implementation of **`myfree`** given this design. Your **`myfree`** should do the following:

- If **ptr** is outside the range of addresses of the heap, return without doing anything. Otherwise assume **ptr** points to a valid payload.
- Check that the block is currently allocated (if not, return without doing anything).
- Mark the header and footer values appropriately for a free block (don't forget to set the realloc bit!)
- Insert the block at the front of the free list.
- This version does not attempt to do coalesce or other advanced features.

```
void myfree(void *ptr) {
    // YOUR CODE HERE
```

F) [23 points] Write an implementation of **`myrealloc`** given this design. Your **`myrealloc`** should do the following:

- Assume **ptr** points to a valid, allocated block's payload. If the realloc size is less than or equal to the current payload size, do nothing and return **ptr**. (you do not need to change the value of the reallocated bit in this case)
- If the block has previously been reallocated, proceed as if the request is for double the actual request size, in anticipation that there may be future myrealloc calls that will now be very fast so long as they fit within this preemptive doubling.
- Perform the realloc by calling myfree and mymalloc (do not manually do any of the work for this). Be sure to copy the caller's data for them. Even though you are the heap allocator, order these operations in a way that respects the usual convention that one does not access freed memory.
- Be sure that the allocated and reallocated flag bits are set appropriately to show this was reallocated.
- This version does not attempt to look for free blocks to the right to expand into, nor does coalesce or other advanced features.

```
void *myrealloc(void *ptr, size_t size) {
    // YOUR CODE HERE
```

Problem 6: Compiler Optimizations [12 points]

A) [4 points] Name one type of optimization done by GCC, and what that optimization is.

B) [4 points] When you compile a program using optimizations, oftentimes it becomes difficult to step through and debug using GDB. Why is this?

C) [4 points] What is the difference between the static instruction count and the dynamic instruction count? Give an example of when they might differ for a block of code.