Andrew ID: _____

Full Name: _____

Recitation Section: _____

CS 15-213, Fall 2008

Exam 2

Thurs. Oct 30, 2008

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- Write your answers in the space provided for the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 60 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. No calculators or other electronic devices are allowed.
- Good luck!

1 (6):	
2 (9):	
3 (6):	
4 (8):	
5 (10):	
6 (6):	
7 (7):	
8 (8):	
TOTAL (60):	

Problem 1. (6 points):

In **buflab**, you performed various buffer overflow attacks against a vulnerable function `gets` that writes into a small buffer. However, in practice, a decent compiler (such as `gcc`) warns about the vulnerabilities of `gets`, and most programmers tend to take the advice.

Harry Q. Bovik thinks that his code is invulnerable against buffer overflow attacks as long as he stays away from unsafe functions such as `gets`.

Here is a piece of code Bovik wrote; it compiled without warnings under a 32-bit little-endian machine:

```
// str.c (headers omitted)

int main()
{
    char buf[23];
    scanf("%s", buf);
    return 0;
}

void remove_later()
{
    printf("You have found my weakness!!!\n");
}
```

Your goal is to prove Bovik wrong by jumping to the `remove_later` function. Do not worry about how the program would behave upon exiting the function.

Relevant assembly output from `objdump` of the `str` program:

```
080483c0 <main>:
 80483c0:   55                      push   %ebp
 80483c1:   89 e5                   mov    %esp,%ebp
 80483c3:   83 ec 38                sub    $0x38,%esp
 80483c6:   83 e4 f0                and    $0xfffffff0,%esp
 80483c9:   8d 45 d8                lea    0xffffffd8(%ebp),%eax
 80483cc:   83 ec 10                sub    $0x10,%esp
 80483cf:   89 44 24 04             mov    %eax,0x4(%esp)
 80483d3:   c7 04 24 e8 84 04 08    movl   $0x80484e8,(%esp)
 80483da:   e8 f5 fe ff ff          call   80482d4 <scanf@plt>
 80483df:   c9                      leave
 80483e0:   31 c0                   xor    %eax,%eax
 80483e2:   c3                      ret

080483f0 <remove_later>:
 80483f0:   55                      push   %ebp
 80483f1:   89 e5                   mov    %esp,%ebp
 80483f3:   83 ec 08                sub    $0x8,%esp
 80483f6:   c7 04 24 eb 84 04 08    movl   $0x80484eb,(%esp)
 80483fd:   e8 c2 fe ff ff          call   80482c4 <puts@plt>
 8048402:   c9                      leave
 8048403:   c3                      ret
```

Assume that you are allowed to work under the same directory where Bovik created `str`, and you are executing `./hex2raw < exploit | ./str`, where `exploit` contains your attack code in hexadecimal.

Write down the contents of your `exploit`, and use `[n]` to denote `n` consecutive arbitrary bytes:

Problem 2. (9 points):

Consider the following C function to sum all the elements of a 5×5 matrix. Note that it is iterating over the matrix **column-wise**, and iterating over the columns **in reverse order**.

```
char sum_matrix(char matrix[5][5]) {
  int row, col;
  char sum = 0;
  for (col = 4; col >= 0; col--) {
    for (row = 0; row < 5; row++) {
      sum += matrix[row][col];
    }
  }
  return sum;
}
```

Suppose we run this code on a machine whose memory system has the following characteristics:

- Memory is byte-addressable.
- There are registers, an L1 cache, and main memory.
- A char is stored as a single byte.
- The cache is direct-mapped, with 4 sets and 2-byte blocks.

You should also assume:

- `matrix` begins at address 0.
- `sum`, `row` and `col` are in registers; that is, the only memory accesses during the execution of this function are to `matrix`.
- The cache is initially cold and the array has been initialized elsewhere.

Fill in the table below. In each cell, write "**h**" if there is a cache hit when accessing the corresponding element of the matrix, or "**m**" if there is a cache miss.

	0	1	2	3	4
0					
1					
2					
3					
4					

Problem 3. (6 points):

Using pointers

Give the output for the following code snippet, assuming that it was compiled on an IA-32 machine. Variable i, j, and k have memory addresses 600, 700 and 800, respectively.

```
#include <stdio.h>

int main() {
  // Assume that i is stored at memory address 600
  int i = 50;
  // Assume that j is stored at memory address 700
  int *j = &i;
  // Assume that k is stored at memory address 800
  int *k = (int *) i;

  printf("%d,%d,%d", (int) i, (int) &i, (int) (i+1));
  printf("\n");

  printf("%d,%d,%d", (int) j, (int) &j, (int) (j+1));
  printf("\n");

  printf("%d,%d,%d", (int) k, (int) &k, (int) (k+1));
  printf("\n");
  return 0;
}
```

This program prints out three lines. Each line has three values that are separated by a comma. What is the output?

Problem 4. (8 points):

Consider the following C program, with line numbers:

```
1   int main() {
2     int counter = 0;
3     int pid;
4
5     while (counter < 4 && !(pid = fork())) {
6       counter += 2;
7       printf("%d", counter);
8     }
9
10    if (counter > 0) {
11      printf("%d", counter);
12    }
13
14    if (pid) {
15      waitpid(pid, NULL, 0);
16      counter += 3;
17      printf("%d", counter);
18    }
29  }
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.
- Logical operators such as && evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result.

A. List all possible outputs of the program in the following blanks.

(You might not use all the blanks.)

_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

B. If we modified line 10 of the code to change the `>` comparison to `>=`, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there?

(Just give a number, you do not need to list them all.)

```
NEW NUMBER OF POSSIBLE OUTPUTS = ___________
```

Problem 5. (10 points):

Consider the following C program:

```
void handler1(int sig) {
    printf("Phantom\n");
    exit(0);
}

int main()
{
    pid_t pid1;

    signal(SIGUSR1, handler1);

    if((pid1 = fork()) == 0) {
        printf("Ghost\n");
        exit(0);
    }
    kill(pid1, SIGUSR1);
    printf("Ninja\n");
    return 0;
}
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.

Mark each column that represents a valid possible output of this program with ‘Yes’ and each column which is impossible with ‘No’.

`Phantom`
`Ninja`
	`Ninja`
	`Phantom`
	`Ghost`
		`Ghost`
		`Ninja`
		`Phantom`
			`Ninja`
			`Ghost`
				`Ninja`
				`Phantom`
				`Ninja`

Problem 6. (6 points):

Consider a system with 10 GB of physical memory (with a 4 KB page size) and a 50 GB disk drive with the following characteristics:

- 512-byte sectors
- 800 sectors/track
- 15,000 RPM (i.e., 4ms to complete one full revolution)
- 8ms average seek time

Imagine an application that MALLOC()s nearly 50 GB of space, initializes it to all zeros, and then randomly selects integers from across the full space and increments them. (Assume that there are no other processes.)

A. What percentage of the integers selected would result in page faults? _____

B. What is the average time to service a page fault? (round to the nearest millisecond) _____

C. Approximately how many integers can be incremented per second? (again, rounding is fine) _____

D. If an additional 15 GB of physical memory were available, how many integers could be incremented incremented per second? (again, rounding is fine) _____

Problem 7. (7 points):

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    char c;
    int file1 = open("buffer.txt", O_RDONLY);
    int file2;
    int file3 = open("buffer.txt", O_RDONLY);

    read(file1, &c, 1);
    file2 = dup(file1);
    read(file1, &c, 1);
    read(file2, &c, 1);

    printf("1 = %c\n", c);

    int pid = fork();
    if(pid == 0) {
        read(file3, &c, 1);
        printf("2 = %c\n", c);

        dup2(file2, file3);
        close(file1);
        read(file3, &c, 1);
        printf("3 = %c\n", c);

        file1 = open("buffer.txt", O_RDONLY);
        read(file1, &c, 1);
        printf("4 = %c\n", c);
    } else {
        waitpid(pid, NULL, 0);
        printf("5 = %c\n", c);

        read(file3, &c, 1);
        printf("6 = %c\n", c);
        close(file2);
        dup2(file1, file2);
        read(file1, &c, 1);
        printf("7 = %c\n", c);
    }
    return 0;
}
```

Assume that the disk file `buffer.txt` contains the string of bytes `PRECOUNT` . Also assume that all system calls succeed. What will be output when this code is compiled and run? You may not need all the lines in the table given below.

Output Line Number	**Output**
1^{st} line of output	
2^{nd} line of output	
3^{rd} line of output	
4^{th} line of output	
5^{th} line of output	
6^{th} line of output	
7^{th} line of output	

Problem 8. (8 points):

Imagine a system with the following attributes:

- The system has 1MB of virtual memory
- The system has 256KB of physical memory
- The page size is 4KB
- The TLB is 2-way set associative with 8 total entries.

The contents of the TLB and the first 32 entries of the page table are given below. **All numbers are in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	1B	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

A. Warmup Questions

(a) How many bits are needed to represent the virtual address space? _____

(b) How many bits are needed to represent the physical address space? _____

(c) How many bits are needed to represent a page table offset? _____

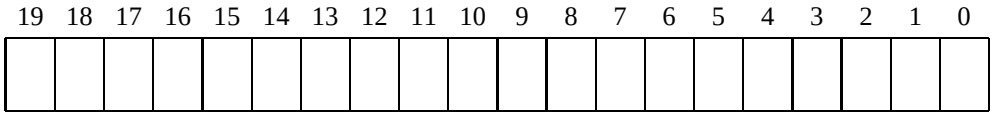
B. Virtual Address Translation I

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x1F213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).



(Please go to the next page for part C)

C. Virtual Address Translation II

Please step through the following address translation. Indicate a page fault by entering '-' for Physical Address.

Virtual address: `0x14213`

Parameter	Value	Parameter	Value
VPN	0x	TLB Hit? (Y/N)	
TLB Index	0x	Page Fault? (Y/N)	
TLB Tag	0x	Physical Address	0x

Use the layout below as scratch space for the virtual address bits. To allow us to give you partial credit, clearly mark the bits that correspond to the VPN, TLB index (TLBI), and TLB tag (TLBT).

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0