Full Name:________________________

Final Exam, Spring 2012 Date: May 14th, 2012

Instructions:

• This final exam takes 1 hour and 30 minutes. Read through all the problems and complete the easy ones first.

• This exam is OPEN BOOK. You may use any books or notes you like. However, laptop usage is limited to browsing the e-book only.

<table>
<thead>
<tr>
<th>1 (xx/20)</th>
<th>2 (xx/20)</th>
<th>3 (xx/20)</th>
<th>4 (xx/20)</th>
<th>5 (xx/20)</th>
<th>Total (xx/100)</th>
</tr>
</thead>
</table>
**Problem 1 (20 pts)**

Ben Bitdiddle is working on a machine that only supports 8-bit integer computation. Since his computation requires using bigger integers, Ben decides to emulate 16-bit integer arithmetic in software.

Ben’s code skeleton is shown below. Ben uses a struct of two 8-bit integers to represent a 16-bit integer in 2’s complement representation. The high field of BigInt is a signed 8-bit integer which contains the high-order 8 bits in the 2’s complement representation including the 2’s complement sign bit. The low field of BigInt is an unsigned 8-bit integer which contains the low-order 8 bits in the 2’s complement representation. For example, suppose BigInt x represents decimal 260 (0x0104), then x.high = 0x01 and x.low = 0x04. As another example, suppose x represents decimal -1 (0xffff), then x.high = 0xff and x.low = 0xff.

```c
typedef struct BigInt {
    char high; /* high-order 8 bits including 2’s complement sign bit */
    unsigned char low; /* low-order 8 bits of 2’s complement */
} BigInt;

BigInt add(BigInt a, BigInt b) {
    BigInt c;
    /* compute c = a + b */

    return c;
}

BigInt sub(BigInt a, BigInt b) {
    BigInt c;
    /* compute c = a - b */

    return c;
}
```

(a) (2 pts) Given BigInt x such that x.high = 0x80 and x.low = 0x00, what is x’s value in decimal?

(b) (10 pts) Please complete add and sub functions for Ben Bitdiddle.
(c) (8 pts) Alyssa P. Hacker has convinced Ben to test the correctness of his code on x86 which does support 16-bit integers. Alyssa wrote the following test code skeleton for Ben. Please help Ben complete the two functions BigInt_to_short and short_to_BigInt.

```c
short
BigInt_to_short(BigInt a)
{
    short x;
    /* convert BigInt a to a short (16-bit) integer*/

    return x;
}

BigInt
short_to_BigInt(short x)
{
    BigInt a;
    /* convert short integer x to BigInt representation*/

    return a;
}

void test(short x, short y) {
    short z;
    BigInt a,b,c;

    a = short_to_BigInt(x);
    b = short_to_BigInt(y);

    c = add(a,b);
    z = BigInt_to_short(c);
    assert(z == (short)(x+y));

    c = sub(a,b);
    z = BigInt_to_short(c);
    assert(z == (short)(x-y));
}

void main() {
    int i;
    for (i = 0; i <10000000; i++) {
        test(random(), random());
    }
}
```
Problem 2 (20 pts)

Ben Bitdiddle has written the following program to print a string of character 'a' (the number of 'a's is given by the command line argument). For example, when typing $./a.out 5, the program is supposed to output the string "content of buf is aaaaa".

```c
char init_char = 'a';
char *buf;

void
create_string(char *b, int len)
{
    int i;
    b = (char *) malloc(len + 1);
    for (i = 0; i < len; i++) {
        b[i] = init_char;
    }
    b[len] = '\0';
}

void main()
{
    create_string(buf, atoi(argv[1]));
    printf("content of buf is %s\n", buf);
}
```

(a) (12 pts) For each of the following variable, please indicate the size of the memory storage it occupies AND which region of the program it occupies. This program runs on a 32-bit machines and the valid program regions are .text/.rodata/.data/.bss/stack/heap.

<table>
<thead>
<tr>
<th>variable</th>
<th>size</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) (8 pts) Does Ben’s program do what he intends it to do? If not, please correct his error. (You can ignore the error of not freeing the malloc-ed buffer.)
Problem 3 (20 pts)

Ben Bitdiddle is trying to figure out what the following x86-32 assembly does. He has so far written a C function skeleton to describe what the assembly does.

```assembly
<foo>:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl $0, -8(%ebp)
    movl $0, -4(%ebp)
    jmp .L2
.L3:
    addl $1, -8(%ebp)
    movl 12(%ebp), %eax
    addl %eax, -4(%ebp)
.L2:
    movl -4(%ebp), %eax
    cmpl 8(%ebp), %eax
    jl .L3
    movl -8(%ebp), %eax
    leave
    ret

<main>:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl $8, 4(%esp)
    movl $16, (%esp)
    call foo
    movl %eax, -4(%ebp)
    leave
    ret

int foo(int x, int y)
{
    int i;
    int n;

    __________;

    for (__________; ____________; ____________) {
        __________;
    }

    __________;
}

void main()
{
    int n = foo(16, 8);
}
```
(a) (12 pts) Please help Ben complete the C code equivalent of the assembly.

(b) (8 pts) In function foo’s assembly code, what is being stored at following memory locations?

<table>
<thead>
<tr>
<th>Memory location</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12(%ebp)</td>
<td></td>
</tr>
<tr>
<td>8(%ebp)</td>
<td></td>
</tr>
<tr>
<td>-8(%ebp)</td>
<td></td>
</tr>
<tr>
<td>-4(%ebp)</td>
<td></td>
</tr>
</tbody>
</table>
Problem 4 (20 pts)

Ben Bitdiddle has to use the library function `unsafe_getinput` which has a buffer overrun vulnerability. Ben’s code using `unsafe_getinput` is shown below.

```c
#define LEN 100

void f1()
{
    L1:
    f2();
    L2:
}

void f2()
{
    L3:
    char buf[LEN]; /* the largest legitimate input is LEN bytes long*/
    L4:
    f3(buf);
    L5:
}

void f3(char *buf)
{
    L6:
    unsafe_getinput(buf); /* this function might overrun buf */
    L7:
}
```

(a) (4 pts) When the attacker exploits the buffer overrun bug in `unsafe_getinput` to execute malicious code by overwriting a return address. Which function would that return address have been pointing to if the attack had not occurred?

(b) (8 pts) Ben asks Alyssa to think of a way to detect the buffer overrun. Alyssa suggests Ben allocate a variable on the stack next to the buffer and to initialize it with a random value. If the buffer is overrun, then the initialized value in the variable will be destroyed. By checking if the content of the variable has changed, one can detect whether `unsafe_getinput` has resulted an overflow.

Suppose the code to allocate and initialized the variable is “`int canary = 0xdeadbeef;`” Using the labels given in the code, please indicate where Ben should put this code.

Suppose the code to check whether the canary value is destroyed is “`assert(canary==0xdeadbeef);`” Using the labels given in the code, please indicate where Ben should put this code.
(c) (8 pts) After Googling the literature on stack overflow defenses, Alyssa realizes that her solution based on stack canary might not work in all cases. Alyssa brainstormed with her fellow CSO students and found another solution based on virtual memory protection. This time, Ben is to allocate an entire memory page next to the buffer and makes it not writable. Thus, when the buffer is about to be overrun, a page fault will occur and cause the program to be aborted.

Please help Ben write the code to implement Alyssa’s idea. You can either write your code directly on top of the original code, or write your code in the space below and indicate where it belongs using the labels given above. (Hint: you need to use the mprotect syscall and its man page is shown in the next page)
NAME
mprotect - set protection on a region of memory

SYNOPSIS
#include <sys/mman.h>
int mprotect(const void *addr, size_t len, int prot);

DESCRIPTION
mprotect() changes protection for the calling process’s memory page(s)
containing any part of the address range in the interval
[addr, addr+len-1]. addr must be aligned to a page boundary. If the
calling process tries to access memory in a manner that violates the
protection, then the kernel generates a SIGSEGV signal for the process.
prot is either PROT_NONE or a bitwise-or of the other values in the following list:

PROT_NONE The memory cannot be accessed at all.
PROT_READ The memory can be read.
PROT_WRITE The memory can be modified.
PROT_EXEC The memory can be executed.

RETURN VALUE
On success, mprotect() returns zero. On error, -1 is returned, and
errno is set appropriately.
Problem 5 (20 pts)

Courtney E. Compileur is writing a program that will scan through source code looking for mistakes. She wants to write the part of the program that will count and log three types of messages: errors, warnings, and infos. She writes a function that her program will use, which is called once for each message. As its arguments it takes the type of message (error, warning, or info) and a string holding the current message. The function is responsible for updating the global num_errors, num_warnings, and num_infos variables and writing the message to a log file. When it writes the message to the log, it prepends it with the correct current count of errors, warnings, and info. She flexes her fingers and gets to work, composing this function:

```c
#define MSG_ERROR 0
#define MSG_WARN 1
#define MSG_INFO 2

int num_errors = 0;
int num_warnings = 0;
int num_infos = 0;
FILE *logfile; = 0;

void do_log(int type_of_msg, char *msg) {
    // Update message counts
    switch(type_of_msg) {
        case MSG_ERROR:
            num_errors++;
            break;
        case MSG_WARN:
            num_warnings++;
            break;
        case MSG_INFO:
            num_infos++;
            break;
    }

    if (!logfile_open) {
        logfile = fopen("somefile.log","w"); //assume this succeeds
        logfile_open = 1;
    }

    //Write something like "e1 w40 i84 Message goes here" to log
    fprintf(logfile,"e%d w%d i%d\t%s\n",num_errors,num_warnings,num_infos,msg);
}
```

// caller function eventually closes logfile properly.
Courtney is working with an admirably helpful system on which every system call succeeds, so she doesn’t need to check the return value of system calls. The above function works nicely in her single-threaded program. Unfortunately, she wants to scan through the 15 million lines of code in the 3.2 Linux kernel, and it’s taking way too long. She has a quad-core machine, and figures that she can scan at least four files at a time by threading her program. However, when she starts testing it, she finds that the messages don’t look right, and some of the error/warning/info counts are wrong.

(a) (10 pts) What’s wrong with her implementation when used in a threaded program? What specific bugs will she see?

(b) (10 pts) Courtney realizes that she should be using some sort of mutices to synchronize accesses to global state to make her function thread-safe. Help her add the global mutices and use the \( \text{P}() \) and \( \text{V}() \) primitives to perform locking by annotating the code above. She wants to write optimized code, so help her use as few mutices as possible and keep locks held for as brief a time as possible. Watch out for deadlocks! Write the code as if the counts should match the messages, but it’s ok if the full lines written to the logfile get reordered. What would change in her creation and use of the mutices if she wanted to keep the lines in order?
Solution

P1 (a) $-2^{15}$

P1 (b)

BigInt

add(BigInt a, BigInt b)
{
    BigInt c;
    c.high = a.high + b.high;
    c.low = a.low + b.low;
    return c;
}

BigInt

sub(BigInt a, BigInt b)
{
    BigInt c;
    c.high = a.high - b.high;
    c.low = a.low - b.low;
    return c;
}

P1(c)

short

BigInt_to_short(BigInt a)
{
    short x;
    x = a.high << 8;
    x |= a.low;
    return x;
}

BigInt

short_to_BigInt(short x)
{
    BigInt a;
    a.high = x >> 8;
    a.low = x;
    return a;
}
P2 (a)

<table>
<thead>
<tr>
<th>variable</th>
<th>size (in bytes)</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_char</td>
<td>1</td>
<td>.data</td>
</tr>
<tr>
<td>buf</td>
<td>4</td>
<td>.bss</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td>stack</td>
</tr>
<tr>
<td>b[0]</td>
<td>1</td>
<td>heap</td>
</tr>
<tr>
<td>len</td>
<td>4</td>
<td>stack</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>stack</td>
</tr>
</tbody>
</table>
P2 (b) The `create_string` function is not passing back the malloc-ed buffer to the caller. One way to fix this is as follows:

```c
void create_string(char **b, int len)
{
    int i;
    char *new_buf = (char *)malloc(len+1);
    for (i = 0; i < len; i++) {
        new_buf[i] = init_char;
    }
    new_buf[len]= '\0';
    *b = new_buf;
}

void main()
{
    create_string(&buf, atoi(argv[1]));
    printf("content of buf is \%s\n", buf);
}
```

P3 (a)

```c
int foo(int x, int y)
{
    int i;
    int n;
    n = 0;
    for ( i = 0; i < x ; i+=y) {
        n++;
    }
    return n;
}
```

P3 (b)

<table>
<thead>
<tr>
<th>Memory location</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12(%ebp)</td>
<td>y</td>
</tr>
<tr>
<td>8(%ebp)</td>
<td>x</td>
</tr>
<tr>
<td>-8(%ebp)</td>
<td>n</td>
</tr>
<tr>
<td>-4(%ebp)</td>
<td>i</td>
</tr>
</tbody>
</table>
void f2()
{
    int canary=0xdeadbeef;
    char buf[LEN]; /* the largest legitimate input is LEN bytes long*/
    f3(buf);
    assert(canary==0xdeadbeef);
}

#define PAGE_SIZE (1<<12)
void f2()
{
    /* allocate two pages on stack just above buf*/
    /* we need two pages for alignment purpose*/
    char guard[2*PAGE_SIZE];
    char buf[LEN];
    /* do mprotect by aligning guard to page boundary. */
    assert(mprotect(guard+PAGE_SIZE-((int)guard&(PAGE_SIZE-1)), PAGE_SIZE, PROT_NONE)==0);
    f3(buf);
}