CSE 351 Final Solutions - Winter 2015

March 18, 2015

Please read through the entire examination first! We designed this exam so that it can be completed in 110 minutes and, hopefully, this estimate will prove to be reasonable.

There are 10 problems for a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space, you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply, and that you write your name on all pages. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones, no laptops). Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good luck and have fun!

Name: ______________________

Student ID: ____________________

Problem	Max Score	Score
1	10	
2	10	
3	15	
4	10	
5	15	
6	10	
7	10	
8	5	
9	5	
10	10	
TOTAL	100	

1 Buffer overflow (10 points)

The following code runs on a 32-bit x86 Linux machine. The figure below depicts the stack at point A before the function `baz()` returns. The stack grows downwards towards lower addresses.

```
int foo(char *buf);
void bar(void);

void baz() {
    char buf[4];
    gets(buf);
    if (foo(buf))
        bar();
A:
    return 0;
}
```

Higher Addresses	...
	Return Address
	Old `%ebp`
Lower Addresses	`buf`

Recall that `gets()` is a `libc` function that reads characters from standard input until the newline ('\n') character is encountered. The resulting characters are stored in the buffer that's given to `gets()` as a parameter. If any characters are read, `gets()` appends a null-terminating character ('\0') to the end of the string.

(a) Explain why the use of the `gets()` function introduces a security vulnerability in the program.

gets() reads from input until a newline character is reached. This allows an arbitrarily large string to be read from input which can be larger than the size of the buffer passed to gets(). An attacker could potentially overwrite information on the stack.

(b) Given the following input strings, indicate whether the string has the potential to cause a segmentation fault. For any string that can cause a segmentation fault, explain why.

String	**Seg fault?**	**Explanation**
`a`	N	
`hi`	N	
`beef`	Y	null terminator ('\0') overwrites old %ebp
`steak`	Y	'k' and null terminator ('\0') overwrites old %ebp
`fried chicken`	Y	overwrites both old %ebp and the return address

(c) The function `bar()` is located at address `0x00000351`. Construct a string that can be given to this program that will cause the function `baz()` to unconditionally transfer control to the function `bar()`. Provide a hexademical representation of your attack string.

00 00 00 00 00 00 00 00 51 03 00 00

The first 8 bytes are just padding so they can be anything.

(d) How should the program be modified in order to eliminate the vulnerability the function `gets()` introduces?

Use fgets() instead. We gave full points for anything that mentioned restricting the size of the input.

(e) Describe two types of protection operating systems and compilers can provide against buffer overflow attacks. Briefly explain how each protection mechanism works.

We accepted three options: Making the stack non-executable, adding stack canaries, or stack address randomization.

2 Mystery cache (10 points)

Let mystery4.o define a cache with block size B, cache size C and associativity A.

(a) Getting cache associativity

Billy writes the following code to compute the associativity:

```
int cache_associativity(int cache_size) {
   access_cache(0);
   int i = 1;
   while (access_cache(0)) {
      access_cache(i * cache_size);
      i++;
   }
   return i - 1;
}
```

His code never seems to return the correct value. Without rewriting his code, explain the flaw in his method.

access_cache(0) in the while loop always makes address 0 the LRU line in the cache, thus it is never kicked out by successive accesses. This will result in an infinite loop. (Originally there was a typo on the exam where we returned i instead of i - 1, we gave credit for that answer as well).

(b) Suppose some person in your class named Susie is only given B and C. Give a sequence of cache queries (each of which return HIT or MISS) whose responses would allow Susie to determine that the associativity of the cache is 2.

access_cache(0) = MISS, access_cache(C) = MISS, access_cache(0) = HIT, access_cache(C) = HIT, access_cache(2*C) = MISS, access_cache(0) = MISS

(c) Can we still address a cache using the same methods presented in class if A is not a power of 2? Explain why/why not?

Yes, the associativity does not need to be a power of 2 because it is irrelevant when addressing into the cache.

(d) Let B = 16, C = 1024, and A = 2. How many sets are in the cache?

32

3 Virtual Memory (15 points)

We have a system with the following properties:

- a virtual address of 14 bits
- a physical address of 10 bits
- pages are 64 bytes
- a TLB with 16 entries that is 4-way set associative
- a cache with 8 sets, a 4 byte block size, and is 2-way set associative

The current contents of the TLB, Page Table, and Cache are shown below:

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	17	0	1	06	-	0	3F	E	1
1	15	3	1	0A	-	0	00	B	1	01	F	1
2	07	-	0	2B	-	0	3F	2	1	2B	-	0
3	31	C	1	2C	1	1	02	0	0	1A	1	1

Page Table (Note: Not all entries are shown)

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	3	1	04	-	0	08	3	1	0C	F	1
01	6	1	05	-	0	09	-	0	0D	-	0
02	3	1	06	-	0	0A	1	1	0E	6	1
03	3	1	07	-	0	0B	3	1	0F	A	1

Cache

Set	Tag	V	B0	B1	B2	B3	Tag	V	B0	B1	B2	B3
0	1F	1	99	1F	34	56	11	1	DE	AD	BE	EF
1	0C	0	27	A4	C5	23	02	0	FF	FF	FF	FF
2	01	1	54	21	65	78	0F	0	FF	FF	FF	FF
3	1F	1	01	02	03	04	07	1	CA	FE	12	34
4	16	1	3E	DE	AD	0F	14	0	FF	FF	FF	FE
5	1D	0	7F	FF	FF	FF	03	1	1F	2E	11	09
6	03	1	12	5E	67	90	12	0	00	00	00	01
7	13	0	00	00	00	00	0F	1	12	34	56	78

(a) How many total entries are in the page table? (Note: the page table above does not show every PTE)

2^8

(b) Fill in the blanks with the number of bits needed for each component.

Virtual Page Number ___8___ Virtual Page Offset ___6___

TLB Tag ___6___ TLB Index ___2___

Physical Page Number ___4___ Physical Page Offset ___6___

Cache Tag ___5___ Cache Index ___3___ Cache Offset ___2___

(c) Complete the following memory accesses using the given virtual addresses and the information on the previous page. Use N/A if the column cannot be determined. Give your answers as hex numbers for Physical Address, and Data. Use Y/N for TLB Miss?, Page Fault?, and Cache Miss?

Virtual Address	Physical Address	Data	TLB Miss?	Page Fault?	Cache Miss?
0x3F36	0x3B6	N/A	N	N	Y
0x01BA	N/A	N/A	Y	Y	N/A
0x02EE	0x0EE	0x12	Y	N	N
0x2512	N/A	N/A	Y	N/A	N/A

4 Processes (10 points)

(a) (4 points) What are the **two key abstractions** that processes provide to programs? For each abstraction describe a mechanism that enables it to work. **Note:** Full sentences are not required.

Full control of the CPU: Context Switching Full control of memory space: Virtual Memory

(b) (6 points) Consider the following C program:

```
void forker() {
    int n = 1;

    if (fork() == 0) {
        printf("%d", n);
        n = n << 1
        if(fork() == 0) {
            n = n << 1;
            printf("%d", n);
            n = n << 1;
        }
    } else {
        n = 0;
    }
    printf("%d", n);
}
```

Which outputs are possible for this program? (circle your choices)

i) 02148
ii) 14028
iii) 01428
iv) 20418
v) 10482

ii, iii, and v

5 Assembly (15 points)

Suppose your CSE friend wants to send you an encrypted message. She's given you a 1 byte decryption key beforehand, and has made the message by XOR'ing this key with each byte in her original message. Unfortunately, you're working on a computer which only knows x86-64 assembly.

To decode her message, write an assembly function which XOR's each byte in the message with the decryption key. Suppose you have a pointer to the beginning of the message stored in `%rdi`, the size (in bytes) of the message stored in `%rsi`, and the decryption key stored in `%rdx`. You should overwrite the current contents of the message in memory. Two instructions have been added to get you started. Note that the first line suggests which registers you should use as variables. For reference, our solution added 5 lines.

```
decrypt_message:
                //Your code here

                movq $0, %rax ; Initialize an offset variable


     loop_start:
                movb (%rdi, %rax), %r10 ; Get the next byte
                xor %rdx, %r10 ; Do the XOR operation

                movb %r10, (%rdi, %rax) ; Put the byte back into memory

                add $1, %rax ; Increment the offset
                cmpl %rax, %rsi ; Compare the offset to the size of the buffer

                jne <loop_start>
                ret
```

6 Pointers, arrays and structs (10 points)

Consider the following variable declarations, assuming x86_64 architecture:

```
typedef struct {
        int a;
        char b;
        double c;
    } struct_type;

    struct_type* m;
    struct_type n[2];
```

Typo: the struct declaration should have been a typedef so that `struct_type`. As it was written, struct_type would be a variable of the unnamed struct type.

Fill in the following table:

C Expression	Evaluates to?	Resulting data type
m	0x10000000	struct_type*
n	0x20000000	struct_type*
&(m->a)	0x10000000	int*
&(m->b)	0x10000004	char*
&(m->c)	0x10000008	double*
sizeof(struct_type)	16	size_t (or int)
sizeof(*m)	16	size_t (or int)
sizeof(m)	8	size_t (or int)
&(n[0])	0x20000000	struct_type*
&(n[0].a)	0x20000000	int*
&(n[1].a)	0x20000010	int*

Some students answered ”pointer” or ”address” as the resulting data type. This was not specific enough to receive full credit.

7 C to assembly (10 points)

(a) Consider the following functions (*hint: don't forget to think about alignment in structs*):

```
struct building1 {
    char *name;
    int x;
    int y;
}
```

```
struct building2 {
    char *address;
    char state3;
    unsigned int zip;
}
```

```
int print_name_and_address(struct building1 *a, struct building2 *b) {
    int ret;
    ret = printf("%s, %s", a->name, b->address);
    return ret;
}

int print_address_and_coordinate(struct building1 *a, struct building2 *b) {
    int ret;
    ret = printf("%s, (%d, %d)", b->address, a->x, a->y);
    return ret;
}

int print_state_and_zip(struct building2 *b) {
    int ret;
    ret = printf("%s, %u", b->state, b->zip);
    return ret;
}
```

(b) Consider the following x86-64 assembly code

```
<func1>:
    subq    $0x8, %rsp
    movq    %rdi, %rdx
    movq    (%rdx), %rsi
    movl    $0x400704, %edi
    xorq    %eax, %eax
    call    printf
    addq    $0x8, %rsp
    retq

<func2>:
    subq    $0x8, %rsp
    movl    0xc(%rdi), %ecx
    movl    0x8(%rdi), %edx
    movq    (%rsi), %rsi
    movl    $0x4006f7, %edi
    movl    $0x0, %eax
    call    printf
    addq    $0x8, %rsp
    retq
```

```
<func3>:
    subq    $0x8, %rsp
    movl    0xc(%rdi), %edx
    leaq    0x8(%rdi), %rsi
    movl    $0x400704, %edi
    xorq    %eax, %eax
    call    printf
    addq    $0x8, %rsp
    retq

<func4>:
    push    %rbx
    push    %r12
    subq    $0x28, %rsp
    movq    %rdi, 0x8(%rsp)
    movq    %rsi, (%rsp)
    movq    (%rsp), %rbx
    movq    (%rbx), %rdx
    movq    0x8(%rsp), %r12
    movq    (%r12), %rsi
    movl    $0x4006eo, %edi
    movl    $0x0, %eax
    call    printf
    movl    %eax, 0x1c(%rsp)
    movl    0x1c(%rsp), %eax
    addq    $0x28, %rsp
    popq    %r12
    popq    %rbx
    retq
```

(c) Match each C function with its correct assembly version. Note: one assembly version will not match.

print_name_and_address -> func4
print_address_and_coordinate -> func2
print_state_and_zip -> func3

(d) Describe what the non-matching assembly code is printing assuming its argument is one of the building structs. (write both possibilities)

if struct building1 *a -> printf("%s, %u", a->name, a); (second argument could be &(a->name))
if struct building2 *b -> printf("%s, %u", b->address, b); (second argument could be &(b->address))

8 Memory bugs (5 points)

What is wrong with each of these three functions below?

```
void foo() {
    int val;
    ...
    scanf("%d", val);
}
```

Bug description:

scanf takes a pointer to an int, not an int. Replace val with &val in the function call.

```
int N = 20;
int M = 10
void bar() {
    int **p;

    p = (int **)malloc( N * sizeof(int) );

    for (i=0; i<N; i++) {
        p[i] = (int *)malloc( M * sizeof(int) );
    }
}
```

Bug description:

p should be pointing to a space big enough to hold N int pointers not N ints. Replace the first sizeof(int) with sizeof(int*).

```
int *bla () {
   int val;

   return &val;
}
```

Bug description:

The local variable val lives on the stack. Returning the address of a variable on the stack is undefined because when we leave the function local variables are no longer valid.

9 Java (5 points)

(a) Why can Java programs do array access out-of-bounds checks?

Java stores the length of the array at the front of the array and checks all accesses during runtime against this size.

(b) Do you have to explicitly free allocated memory in Java? Why yes or why not?

No, Java has automatic garbage collection.

10 Aligned malloc for matrices (10 points)

Allocating a matrix is common operation in numeric programs (e.g., machine learning). Aligning matrices to cache-line granularity is often beneficial for performance. Your goal is to write the code for a malloc wrapper that given the matrix parameters (nRows rows $\times$ nCols columns), allocates enough memory to return a pointer to a 64-byte aligned block. We provided a function prototype for you below with blanks to be filled. You do not need to worry about freeing the allocated block.

```
double* aligned_matrix_malloc(size_t nRows, size_t nCols) {
    double *aligned_ptr;


    void *m_ptr = malloc(  nRows * nCols * sizeof(double) + 64      );


    aligned_ptr = (double*) ((char*)m_ptr + (64 - ((char*)m_ptr % 64)) ;


    return aligned_ptr;
}
```

References

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	$2^{-6} = 0.015625$
$2^7 = 128$	$2^{-7} = 0.0078125$
$2^8 = 256$	$2^{-8} = 0.00390625$
$2^9 = 512$	$2^{-9} = 0.001953125$
$2^{10} = 1024$	$2^{-10} = 0.0009765625$

Hex help:

```
0x0 = 0 = 0b0000
0x1 = 1 = 0b0001
0x2 = 2 = 0b0010
0x3 = 3 = 0b0011
0x4 = 4 = 0b0100
0x5 = 5 = 0b0101
0x6 = 6 = 0b0110
0x7 = 7 = 0b0111
0x8 = 8 = 0b1000
0x9 = 9 = 0b1001
0xA = 10 = 0b1010
0xB = 11 = 0b1011
0xC = 12 = 0b1100
0xD = 13 = 0b1101
0xE = 14 = 0b1110
0xF = 15 = 0b1111
0x20 = 32
0x28 = 40
0x2A = 42
0x2F = 47
```

Assembly Code Instructions:

`push`	push a value onto the stack and decrement the stack pointer
`pop`	pop a value from the stack and increment the stack pointer
`call`	jump to a procedure after first pushing a return address onto the stack
`ret`	pop return address from stack and jump there
`mov`	move a value between registers and memory
`lea`	compute effective address and store in a register
`add`	add src (1st operand) to dst (2nd) with result stored in dst (2nd)
`sub`	subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
`and`	bit-wise AND of src and dst with result stored in dst
`or`	bit-wise OR of src and dst with result stored in dst
`sar`	shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1st operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`jns`	conditional jump to address if sign flag is not set
`cmp`	subtract src (1st operand) from dst (2nd) and set flags
`test`	bit-wise AND src and dst and set flags

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15`. `rsp` is a special register.

`%rax`	Return Value	`%r8`	Argument #5
`%rbx`	Callee Saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller Saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee Saved
`%rdi`	Argument #1	`%r13`	Callee Saved
`%rsp`	Stack Pointer	`%r14`	Callee Saved
`%rbp`	Callee Saved	`%r15`	Callee Saved