CSE 351 Final Exam - Winter 2017

March 15, 2017

Please read through the entire examination first, and make sure you **write your name and NetID on all pages!** We designed this exam so that it can be completed in 100 minutes.

There are 8 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. 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!

Name: _____

Student ID: _____

Section: _____

Problem	Max Score	Score
1. C and Assembly	15	
2. Buffer Overflow	15	
3. Caches	15	
4. Processes	10	
5. Virtual Memory	15	
6. Memory Allocation	10	
7. Memory Bugs/Java	10	
8. Pointers and Values	10	
TOTAL	100	
Extra Credit	10	

Name: NetID:

1. C and Assembly (15 points)

Consider the following (partially blank) x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit, and the machine is little-endian. All the values in memory are in hex, and the address of each cell is the sum of the row and column headers: for example, address `0x1019` contains the value `0x18`.

Assembly code:

```
foo:
  movl $0, %eax

L1:
  cmpq 0x0, %rdi
  je L2
  cmp 0x18, 0x1(%rdi)
  je L3
  mov 0x8(%rdi), %rdi
  jmp L1

L2:
  ret

L3:
  movzbl (%rdi), %eax
  jmp L2
```

C code:

```
typedef struct person {
  char height;
  char age;
  struct person* next_person;
} person;

int foo(person* p) {
    int answer = 0;
    while (p != NULL) {
        if (p->age == 24){
          answer = p->height;
          break;
        }
        p = p->next_person;
    }
    return answer;
}
```

Memory Listing
Bits not shown are 0.

	0x00	0x01	...	0x05	0x06	0x07
0x1000	80	1B	...	00	00	00
0x1008	80	1B	...	00	00	00
0x1010	3F	18	...	00	00	00
0x1018	3F	18	...	00	00	00
0x1020	00	00	...	00	00	00
0x1028	18	10	...	00	00	00
0x1030	18	10	...	00	00	00
0x1038	40	40	...	00	00	00
0x1040	40	40	...	00	00	00
0x1048	00	00	...	00	00	00

(a) Given the code provided, fill in the blanks in the C and assembly code.

(b) Trace the execution of the call to `foo((person*) 0x1028)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place **the assembly instruction** and the values of the appropriate registers **after that instruction executes**. *You may leave those spots blank when the value does not change.* You might not need all steps listed on the table.

Instruction	`%rdi` (hex)	`%eax` (decimal)
`movl`	`0x1028`	0
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`	`0x1018`	
`jmp`		
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`		63
`jmp`		
`ret`		

(c) Briefly describe the value that `foo` returns and how it is computed. Use only variable names from the C version in your answer.

foo traverses a linked list of person structs, and returns the height of the first person whose age == 24.

2. Buffer Overflow (15 points)

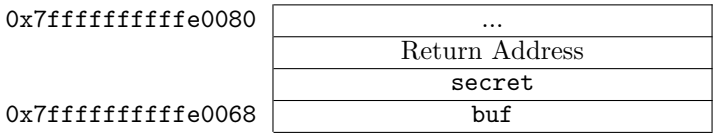
The following code runs on a 64-bit x86 Linux machine. The figure below depicts the stack at point `A` before the function `gatekeeper()` returns. The stack grows downwards towards lower addresses.

```
void get_secret(char*);
void unlock(void);
void backdoor(void);

void gatekeeper() {
    char secret[8];
    char buf[8];

    fill_secret(secret);
    gets(buf);

    if (strcmp(buf, secret) == 0)
        unlock();
A:
    return 0;
}
```



You are joining a legion of elite hackers, and your final test before induction into the group is gaining access to the CIA mainframe. `gatekeeper()` is a function on the mainframe that compares a password you provide with the system's password. If you try to brute force the password, you will be locked out, and your hacker reputation will be tarnished forever.

Assume that `fill_secret()` is a function that places the mainframe's password into the `secret` buffer so that it can be compared with the user-provided password stored in `buf`.

Recall that `gets()` is a `libc` function that reads characters from standard input until the newline ('\n') character is encountered. The resulting characters (not including the newline) 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.

`strcmp()` is a function that returns 0 if two (null-terminated) strings are the same.

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

`gets()` introduces a security vulnerability because it does not have a limit on the number of characters read. This can cause a buffer overflow, allowing a user to enter a malicious string that exceeds our buffer.

(b) You think it may be possible to unlock the mainframe, even without the correct password. Provide a hexadecimal representation of an attack string that causes the `strcmp()` call to return `0`, such that `unlock()` is then called. `gatekeeper()` should return normally, as to avoid raising any suspicion.

Essentially the buffers `secret` and `buf` need to be the same. Thus, any identical null-terminated 8-byte strings will work, as long as they don't contain premature `00`'s, for this counts as a null-terminator. For example, `0xffffffffffffff00ffffffffffffff00` works.

(c) The function `backdoor()` is located at address `0x0000000000000351`. Construct a string that can be given to this program that will cause the function `gatekeeper()` to unconditionally transfer control to the function `backdoor()`. Provide a hexadecimal representation of your attack string.

Fill up buf, fill up secret, replace return address with `backdoor`'s address. For example:

`0x 1234567812345678 1234567812345678 5103000000000000`

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

Various answers accepted here, such as using `fgets()` instead, which has the string length as a parameter.

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

OS:

- memory protection

Compiler:

- stack canaries
- compile-time string length checks
- stack address randomization

3. Caches (15 points)

For all sections, assume 512 byte, 16-bit physical address space, byte-addressable memory.

(a) Assume a 2-way set-associative, write-back cache with LRU replacement that allocates on a write-miss. Assume the cache is initially empty. Suppose we execute the following code segment which copies an array `A[256][4]`, starting at address 0x1000 into an array `B[256][4]`, starting at address 0x2000.

```
char[256][4] A;
char[256][4] B;
for (i = 0; i < 256; i++) {
    for(j = 0; j < 4; j++) {
        A[i][j] = B[i][j];
    }
}
```

We know the cache miss rate for executing the code above is 1/8.

Determine block size, # of sets, # of tag bits: block size = 8, sets = 32, tag bits = 8

(b) Assume an initially empty, direct mapped cache. Calculate the cache miss rate for the given code and different block sizes.

```
char[256][4] A;
char[256][4] B;
for (j = 0; j < 4; j++) {
    for(i = 0; i < 256; i++) {
        A[i][j] = B[i][j];
    }
}
```

4-byte block: 100%

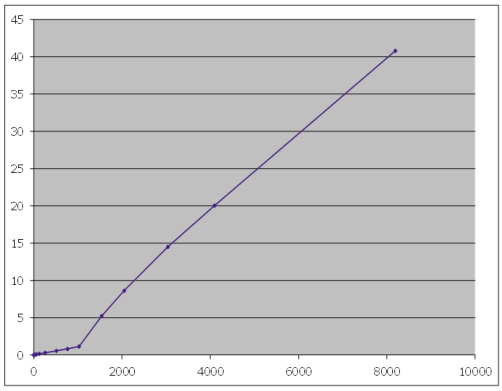
8-byte block: 50%

16-byte block: 25%

(c) Given the code below, draw the line plot (time vs. SIZE) runtime diagram. Remember to label the axes of your plot.

```
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    sum += array[j];
  }
}
```



4. Processes (10 points)

(a) After a context switch, the VPN to PPN mappings in the TLB from the previous running process no longer apply. A simple solution to this problem is to "shoot down" the TLB, by invalidating all the entries in the TLB, but this can often cause inefficiency if there is frequent context switching.

What additional information can be added to the TLB that can be utilized to reduce this inefficiency in the TLB on a context switch? *Hint: consider how processes can be uniquely identified by the MMU.*

Tagging TLB entries with the unique process ID

(b) Suppose you are in control of the CPU and operating system, and you realize that you have a process A that requires a large uninterrupted chunk of CPU time to perform its important work. What would you adjust to ensure that this can happen?

Any of these answers were accepted:

- Extend the context switch timer
- Avoid context switch
- Increase the priority of process A
- Ignore interrupts

One important consideration that a lot of answers did not take into account is that we are in control of the CPU and not another user process, so solutions like using `waitpid()` or other system calls are not correct.

(c) Consider an OS running a process A which incurs a timer interrupt at time t_1. The OS context switches to some other processes which do some work. Later, the OS context switches back to process A at time t_2. Note that process A was not run between t_1 and t_2.

Circle the items which are guaranteed to be the same at time t_1 and t_2.

Underlined items are guaranteed to be the same.

- **Register `%rbx`**: At time t_1, the OS will save all of the register values for process A, including `%rbx`. At time t_2 when the OS context switches back, it restores the value of `%rbx`. Thus we are guaranteed to have the same saved value.
- **Process A's Page Table**: Between t_1 and t_2, another process could have evicted one of process A's pages that was in physical memory, thus altering its value.
- **Instruction pointer**: Similar to register `%rbx`, this register is also explicitly saved. One point of clarification is that a context switch will only occur when a particular instruction is complete. So, at t_1 `%rip` will point to the next instruction that has **not been executed**. At t_2 when the OS context switches back, this value will still be the same so that it will in fact execute that exact instruction.
- **L1 Cache**: The L1 cache can have been changed by other processes' accesses to memory.
- **Page fault handler code**: The page fault handler code is part of the kernel (OS code) so this will not change.
- **Page Table Base Register**: The PTBR contains the location in physical memory of the page table for the currently executing process, which will not change from t_1 to t_2, so this is guaranteed to be the same.

(d) Consider the following C program, running on an x86-64 Linux machine. The program starts running at function `main`. Assume that `printf` flushes immediately.

```
int main() {
    int* x = (int*) malloc(sizeof(int));
    *x = 1;
    if (fork() == 0) {
        spoon(x);
    } else {
        *x = 8 * *x;
        printf("%d\n", *x);
    }
}

void spoon(int* x) {
    printf("%d\n", *x);
    if (fork() == 0) {
        *x = 2 * *x;
    } else {
        *x = 4 * *x;
    }
    printf("%d\n", *x);
}
```

Provide **two** possible outputs of running this code.

Output 1: Output 2:

The main sources of error were:

- Each process has its own copy of `x`, so the largest possible output is 8.
- The first `printf` in `spoon` will always print the `1` before the `2` or `4`.

All possible outputs:

- 8 1 2 4
- 8 1 4 2
- 1 8 2 4
- 1 8 4 2
- 1 2 8 4
- 1 4 8 2
- 1 2 4 8
- 1 4 2 8

5. Virtual Memory (15 points)

Below is the *entire* physical memory of a very tiny machine that implements virtual memory. All the values are in hex, and the address of each cell is the sum of the row and column headers: for example, physical address `0x1a` contains the value `0xdf`.

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	1a	aa	10	21	70	98	84	82
0x08	2c	4a	31	48	9e	02	11	0a
0x10	8b	7c	5a	92	8d	06	b4	2b
0x18	35	21	df	82	06	32	67	91
0x20	47	06	2a	02	89	18	ad	f7
0x28	18	89	a1	ff	47	3f	8e	1f
0x30	62	ef	00	11	1b	94	70	a3
0x38	22	00	5b	9a	0a	78	28	05

The virtual memory system uses 8-bit virtual addresses and 6-bit physical addresses. A single page is 16 bytes. Data is 1 byte-addressable. There is a fully assocative TLB with 2 entries that is **empty at the start of execution**. It evicts according to LRU. Assume **all page faults are handled properly** by the OS, and pages are brought in. The page table base register is `0x00`. The page table entry layout is below, where bit 0 is the least significant.

```
|      PPN      |   unused  |vld|
+---+---+---+---+---+---+---+---+
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+---+---+---+---+---+---+---+---+
```

(a) Fill in the blanks below, explaining what happens when the following reads are executed **in order**. The first column is the virtual address to be read. In the second, put "Hit" or "Miss". In the third, put the value read or "Fault" if there's a page fault.

Address	TLB	Result
0x31	Miss	0x06
0x24	Miss	Fault
0xee	Miss	0x67
0x3a	Miss	0xa1
0xe4	Hit	0x8d

(b) As concisely as possible, provide three benefits of a virtual memory system (in general, unrelated to the system on the previous page).

(i) Illusion of 2^64 bytes of space

(ii) Memory protection

(iii) Shared memory

6. Memory Allocation (10 points)

(a)
```
static void * searchFreeList(size_t reqSize) {
  BlockInfo* freeBlock;

  freeBlock = FREE_LIST_HEAD;
  while (freeBlock != NULL){
    if (SIZE(freeBlock->sizeAndTags) >= reqSize) {
      return freeBlock;
    } else {
      freeBlock = freeBlock->next;
    }
  }
  return NULL;
}
```

Consider an allocator that uses the above function to search for a free block, allocates it, and returns **exactly** that block. Assume it increases the heap size as necessary.

This strategy is **correct** but rather naive. Describe **two** ways in which it could be improved to cause less fragementation.
Note: we want improvements that cause less fragmentation; we don't care about runtime performance.

- Split the free block if it's bigger than `MIN_BLOCK_SIZE`.
- Find best block instead of first block that's big enough.

(b) The `strcpy(char* dest, char* src)` function copies the string pointed to by `src` into the array pointed to by `dest`, including the terminating null character (and stopping at that point).

Describe two ways this code may produce a runtime error:

```
char* copyFoo() {
    char* foo = "foo";
    char* copy = (char*) malloc(3 * sizeof(char));
    strcpy(copy, foo);
    return copy;
}
```

- no malloc null check
- incorrect size of string (not accounting for null-terminator)

7. Memory Bugs (10 points)

For each code section below, **briefly** describe the memory bug and any **security implications** it may have. Also, state whether this kind of bug can occur in Java, briefly explaining why or why not.

(a)
```
int val;
...
scanf("%d", val);
```
`scanf` takes an `int *`, not an `int`.

(b)
```
void read_packet() {
  char s[8];
  int i;
  gets(s);  /* reads \123456789" from stdin */
  ...
}
```
Can cause buffer overflow.

(c)
```
int* search(int* p, int val) {
   while (p && *p != val)
      p += sizeof(int);
   return p;
}
```
Pointer arithmetic is already taken of; should replace 3rd line with `p += 1` instead.

(d)
```
int* foo() {
   int val;

   return &val;
}
```
`foo` returns a pointer to a variable allocated on its stack frame, which will become inaccessible when it returns.

(e)
```
x = (int*) malloc( N * sizeof(int) );
   // <manipulate x>
free(x);
   ...
y = (int*) malloc( M * sizeof(int) );
for (i=0; i<M; i++)
   y[i] = x[i]++;
```
Accessing `x` after it is freed.

8. Pointers and Values (10 points)

Consider the C code:

```
struct foo {
  short ss[4];
  char q;
};

struct bar {
  struct bar* next;
  char x;
  struct foo baz;
};


int main() {
  int question = 1;
  struct bar b = {      // this .field notation initializes structs in
    .next = &b,         // the way you would expect
    .x = 'e',
    .baz =  {
      .ss = {1,2,3,4},
      .q = 'H'
    }
  };
  int answer = 42;

  // <<< BREAK >>>
  return 0;
}
```

The above C code runs until the shown `BREAK`. Fill in the table below with the address, value, and type of the given C expressions at that point. Answer N/A if it is not possible to determine the address or value of the expression. The first row has been filled in for you, revealing the address of `b`. Note the code was not optimized by the compiler so the variables are stored in the order they appear in the code above.

C Expression	Address	Value	Type (char/short*/etc.)
`b`	`0x1000`	the whole struct	`struct bar`
`answer`	`0xffc`	42	`int`
`b.x`	`0x1008`	e	char
`b.next`	`0x1000`	`0x1000`	struct bar *
`b.baz.ss[2]`	`0x100e`	3	short
`question`	`0x101c`	1	int

Reference

You may tear this sheet off if you wish.

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/decimal/binary help:

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Name: NetID:

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 D(base, index, scale), dest`	compute effective address (does not load) and place in register `dest`. (`dest = D + (base + (index * scale))` when `scale` is 1,2,4,8)
`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
`je/jne`	conditional jump to address if zero flag is / is not set
`js/jns`	conditional jump to address if sign flag is / is not set
`cmp`	subtract src (1st operand) from dst (2nd) and set flags, discard result
`test`	bit-wise AND src and dst and set flags, discard result

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

Reference from Lab 5

The functions, macros, and structs from lab5. These are all identical to those in the lab. Note that some of them will not be needed in answering the exam questions.

Structs:

```
struct BlockInfo {
  // Size of the block (in the high bits) and tags for whether the
  // block and its predecessor in memory are in use.  See the SIZE()
  // and TAG macros, below, for more details.
  size_t sizeAndTags;
  // Pointer to the next block in the free list.
  struct BlockInfo* next;
  // Pointer to the previous block in the free list.
  struct BlockInfo* prev;
};
```

Macros:

```
/* Macros for pointer arithmetic to keep other code cleaner.  Casting
   to a char* has the effect that pointer arithmetic happens at the
   byte granularity. */
#define UNSCALED_POINTER_ADD ...
#define UNSCALED_POINTER_SUB ...

/* TAG_USED is the bit mask used in sizeAndTags to mark a block as
   used. */
#define TAG_USED 1

/* TAG_PRECEDING_USED is the bit mask used in sizeAndTags to indicate
   that the block preceding it in memory is used. (used in turn for
   coalescing).  If the previous block is not used, we can learn the
   size of the previous block from its boundary tag */
#define TAG_PRECEDING_USED 2;
```

```
/* SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags'
   field. Also, calling SIZE(size) selects just the higher bits of
   'size' to ensure that 'size' is properly aligned.  We align 'size'
   so we can use the low bits of the sizeAndTags field to tag a block
   as free/used, etc, like this:

      sizeAndTags:
      +-----------------------------------------+
      | 63 | 62 | 61 | 60 |  . . . .  | 2 | 1 | 0 |
      +-----------------------------------------+
        ^                                       ^
      high bit                                low bit

   Since ALIGNMENT == 8, we reserve the low 3 bits of sizeAndTags for
   tag bits, and we use bits 3-63 to store the size.
   Bit 0 (2^0 == 1): TAG_USED
   Bit 1 (2^1 == 2): TAG_PRECEDING_USED
*/
#define SIZE ...

/* Alignment of blocks returned by mm_malloc. */
# define ALIGNMENT 8

/* Size of a word on this architecture. */
# define WORD_SIZE 8

/* Minimum block size (to account for size header, next ptr, prev ptr,
   and boundary tag) */
#define MIN_BLOCK_SIZE ...

/* Pointer to the first BlockInfo in the free list, the list's head.
   A pointer to the head of the free list in this implementation is
   always stored in the first word in the heap.  mem_heap_lo() returns
   a pointer to the first word in the heap, so we cast the result of
   mem_heap_lo() to a BlockInfo** (a pointer to a pointer to
   BlockInfo) and dereference this to get a pointer to the first
   BlockInfo in the free list. */
#define FREE_LIST_HEAD ...
```