**Full Name:**______________________________________

**Univ ID:**______________________________________

# Quiz II, Fall 2018 Date: Nov 1, 2018

**Instructions:**

- This exam takes 70 minutes. Read through all the problems and complete the easy ones first.
- This exam is CLOSED BOOK. You may use a double-sided A4 cheatsheet that you've prepared yourself. Any electronic devices or other paper materials are forbidden.

| 1 (40) | 2 (25) | 3 (25) | 4 (10) | Bonus (15) | Total (100+15) |
|---|---|---|---|---|---|
| | | | | | |

## Notice:

Unless otherwise noted, answer all questions in the quiz assuming a little-endian 64-bit x86 machine. There are two appendices at the end of the exam booklet: 1) a cheatsheet of x86 instructions 2) ASCII table.

## 1 (40 points) Multiple choices. Circle *all* correct or applicable answers:

**A.** Facebook boasts 3 billion users now. If you are to accurately keep track of the number of Facebook users in a C variable called `counter`, what type can `counter` be?

1. `short`
2. `unsigned short`
3. `int`
4. `unsigned int`
5. `long`
6. `unsigned long`

**B.** Which of the following statements are true?

1. Accessing data stored in memory is as fast as accessing data stored in CPU registers.
2. Accessing data stored in memory is much slower than accessing data in CPU registers.
3. A C program is compiled into x86 instructions which are directly executed by the CPU.
4. A Java program is compiled into x86 instructions which are directly executed by the CPU.
5. All x86 instructions are 8 bytes long.

**C.** Normally, the %rip register is updated after executing the current instruction to point to the next instruction. Which of the following instructions *might* cause %rip to point to somewhere other than the next instruction?

1. `jle`
2. `mov`
3. `add`
4. `call`
5. `ret`
6. `sub`

**D.** Which of the following statements about the stack are true?

1. The callee passes back the function's return value to the caller by storing it on the stack.
2. The caller passes the function's arguments to the callee by storing all of them on the stack.
3. The instruction `subq $0x18, %rsp` grows the stack by 24 bytes.
4. The instruction `subq $0x18, %rsp` shrinks the stack by 24 bytes.
5. Global variables are allocated on the stack.
6. Malloc-ed space are allocated on the stack.

**E.** When executing a *buggy* program, which of the following instructions *might* cause the running program to encounter a ”segmentation fault” upon its execution?

1. `movq (%rax,%rbx), %rbx`
2. `movq %rcx, %rbx`
3. `ret`
4. `leaq (%rax,%rbx), %rcx`
5. `subq %rax, (%rbx)`

**F.** Let `a` be an array of `int` elements. Suppose `%rdi` currently store the address of `a[0]`, and `%rsi` stores some index `i` of type `long`. Which of the following sequence of x86 instructions result in `%eax` storing `a[i]`?

1. `leal (%rdi, %rsi, 4), %eax`
2. `movl (%rdi, %rsi, 4), %eax`
3. `movl (%rsi, %rdi, 4), %eax`
4. `leal (%rdi, %rsi, 8), %eax`
5. `movl (%rdi, %rsi, 8), %eax`
6. `movl (%rsi, %rdi, 8), %eax`
7. ```
   salq $2, %rsi
   addq %rdi, %rsi
   movl (%rdi), %eax
   ```
8. ```
   salq $2, %rsi
   movl (%rsi, %rdi), %eax
   ```

**G.** Suppose register `%rax` stores C variable `long x`, which of the following instructions effectively implement `x *= 2`?

1. `movq (%rax,%rax), %rax`
2. `leaq (%rax,%rax), %rax`
3. `leaq (, %rax, 2), %rax`
4. `addq %rax, %rax`
5. `shlq $1, %rax`
6. `salq $1, %rax`
7. `shrq $1, %rax`
8. `sarq $1, %rax`

**H.** Which of the following `mov` instructions are **valid** x86 instructions?

1. `movq $0x400456, %rip`
2. `movq (%rax), (%rbx)`
3. `movb (%rax), %bl`
4. `movb %al, (%bl)`
5. `movq %al, (%bl)`

## 2 C and assembly: (25 points)

**A (5 points).** Please complete the following skeleton function `my_memcpy` to copy `n` bytes from buffer `src` to buffer `dst`. You must only add C code in the space denoted by ________.

```
void my_memcpy(void *dst, void *src, unsigned int n)
{
   char *s = (char *)src;
   char *d = (char *)dst;

   for (______________; _______________; ______________) {

       ______________________________;
   }
}
```

**B (5 points).** The following `main` function is to copy `int` array `a1` to another `int` array `a2`. Please complete the line to properly invoke `my_memcpy` to perform the copying.

```
int main() {
   int a1[2] = {1, 2};
   int a2[2];

   my_memcpy(____________,________________, _______________________);
   printf("a2: %d %d %d\n", a2[0], a2[1]); //this should print out "a2: 1 2"
}
```

**C (5 points).** Given the following `main` function, what would its output be?

```
int main() {
   char c1[3] = {'a', 'b', 'c'};
   char *c2;
   c2 = &c1[1];
   my_memcpy(c2, c1, 2);
   printf("c2: %c %c\n", c2[0], c2[1]);
}
```

The following assembly corresponds to the `main` function in the previous question (**C.**).

```
main:
  subq    $24, %rsp
  movb    $97, (%rsp)
  movb    $98, 1(%rsp)
  movb    $99, 2(%rsp)

  ____    _______, %rdi

  ____    _______, %rsi
  movl    $2, %edx
  call    mymemcpy
  ...
  call    printf
  addq    $24, %rsp
  ret
```

**D (5 points).** Which one of the above assembly instructions initialize the first element of the `c1` array to contain ASCII character 'a'? (You may just circle the instruction.)

**E (5 points).** Complete the two missing x86 assembly instructions marked by ______.

# 3 Assembly Mystery: (25 points)

Ben Bitdiddle is trying to figure out what the following disassembled function `mysterycount` does:

```
mysterycount:
   movl    $0, %edx
   movl    $0, %eax
.L5:
   cmpl    %edi, %edx
   jge     .L3
   movslq  %edx, %rcx
   cmpl    $0, (%rsi,%rcx,4)
   jle     .L4
   addl    $1, %eax
.L4:
   addl    $1, %edx
   jmp     .L5
.L3:
   ret
```

Alyssa P. Hacker decides to give Ben some hints on what `mysterycount` does by providing him with a skeleton C function. She also told Ben that `mysterycount` tries to count the number of elements in a given array that match a specific criteria.

```
1: ______
2: mysterycount( _________,    _________ )
3: {
4:    ________ count = 0;

5:    for (_____ i = 0; ________________;  i++) {

6:       if (_________________________________) {
7:          count++;
8:       }
9:   }
10:  return count;
11:}
```

**A.** (4 points) Where is the C variable `count` stored?

**B.** (4 points) What is the type of `count`? Write it down on line 4. What is the type of the return value of `mysterycount`? Write it down on line 1.

**C.** (4 points) Which function signature (omiting the return value) should `mysterycount` be on line 2?

1. `mystery(int *array, int n)`
2. `mystery(int n, int *array)`
3. `mystery(int *array, unsigned int n)`
4. `mystery(unsigned int n, int *array)`
5. `mystery(char *array, int n)`
6. `mystery(int n, char *array)`
7. `mystery(char *array, unsigned int n)`
8. `mystery(unsigned int n, char *array)`

**D.** (4 points) Where is the loop variable `i` stored ?

**E.** (4 points) What is the type of the loop variable `i`?

**F.** (5) Please complete the rest of the skeleton code by filling out ______.

# 4 Buffer Overflow (10 points + 15 bonus points)

Ben Bitdiddle has thought of a strategy to defend against buffer overflow. The following C progam gives a concrete example of Ben's strategy. After allocating a buffer on the stack, Ben stores the size of the buffer `BUFSZ` at the end of `buf` (line 4). After calling a dangerous function such as `gets` (line 5) which might have overflown the buffer, Ben terminates the program if the stored buffer size has changed (lines 6-8).

Note `gets(buf)` reads from terminal into `buf` until a newline character '\n', which it replaces with '\0'.

```
1: #define BUFSZ 4
2: void echo() {
3:    char buf[BUFSZ+sizeof(int)];
4:    *(int *)(buf+BUFSZ) = BUFSZ; //write BUFSZ as an int at address value buf+BUFSZ
5:    gets(buf);
6:    if ((*(int *)(buf+BUFSZ)) != BUFSZ) {
7:       printf("stack overflow detected!\n");
8:       exit(1); // terminate the program
9:    }
10:   return;
11:}

void divulge_secret() {
  // omitted
}
int main() {
   echo();
}
```

The corresponding disassembled program is as follows:

```
00000000000006ca <echo>:
 6ca: 48 83 ec 18           sub    $0x18,%rsp
 6ce: c7 44 24 0c 04 00 00  movl   $0x4,0xc(%rsp)
 6d5: 00
 6d6: 48 8d 7c 24 08        lea    0x8(%rsp),%rdi
 6db: b8 00 00 00 00        mov    $0x0,%eax
 6e0: e8 ab fe ff ff        callq  <gets>
 6e5: 83 7c 24 0c 04        cmpl   $0x4,0xc(%rsp)
 6ea: 75 05                 jne    6f1 <echo+0x27>
 6ec: 48 83 c4 18           add    $0x18,%rsp
 6f0: c3                    retq
 6f1: 48 8d 3d ac 00 00 00  lea    0xac(%rip),%rdi #this and next instruction do printf
 6f8: e8 83 fe ff ff        callq  <printf>
 6fd: bf 01 00 00 00        mov    $0x1,%edi  #this and the next instruction do exit(1)
 702: e8 99 fe ff ff        callq  <exit>

0000000000000707 <main>:
 70b: b8 00 00 00 00        mov    $0x0,%eax
 710: e8 b5 ff ff ff        callq  6ca <echo>
 715: b8 00 00 00 00        mov    $0x0,%eax
 71e: c3                    retq

000000000000f876 <divulge_secret>:
  # ... omitted....
```

Suppose the value of `%rsp` is `0x00007fffffffd4f8` upon entering function `echo` just before executing its first instruction `sub $0x18, %rsp`.

**A.** (5 points) What are the values of the 8 bytes starting at address `0x00007fffffffd4f8`? (Please write the 8 byte values *one by one* from lower to higher addresses. Each byte value should be represented with 2 hex-digits and successive byte values should be separated with a space.)

**B.** (5 points) What is the starting address of the buffer `buf`? (Hint: %rdi stores the value of the first argument when calling a function.)

The remaining questions are **bonus** questions.

**C.** (5 points) Suppose an attacker enters a sequence of 25 input bytes `31 32 33 34 35 36 37 38 31 32 33 34 35 36 37 38 76 f8 00 00 00 00 00 00 0a` (each byte is represented by 2 hex digits. Note the last byte `0a` is the ASCII value of the newline character). What are the values of the 8 bytes starting at address `0x00007fffffffd4f8` when `callq <gets>` returns? (Write the 8 byte values *one by one* from lower to higher addresses.)

**D.** (5 points) Following the scenario described in **C.**, will the running program return to a corrupted return address? Please explain.

**E.** (5 points) What is the sequence of bytes that the attacker should enter in order to cause the program to execute the `divulge_secret` function.

—- END OF QUIZ II —-

# Appedix: X86 Cheatsheet

## 4.1 Registers

x86 registers are 8-bytes. Additionally, the lower order bytes of these registers can be independently accessed as 4-byte, 2-byte, or 1-byte register. The register names are:

| 8-byte register | Bytes 0-3 | Bytes 0-1 | Byte 0 (lowest order byte) |
|---|---|---|---|
| %rax | %eax | %ax | %al |
| %rbx | %ebx | %bx | %bl |
| %rcx | %ecx | %cx | %cl |
| %rsi | %esx | %si | %sil |
| %rdi | %edx | %di | %dil |

...the rest is omitted...

## 4.2 Instructions

**Instruction suffixes:**

| | |
|---|---|
| ”byte” (`b`) | 1-byte |
| ”word” (`w`) | 2-bytes |
| ”doubleword” (`l`) | 4-bytes |
| ”quardword” (`q`) | 8-bytes |

**Complete memory addressing mode:** A memory operand of the form `D(Rb, Ri, S)` accesses memory at address `D + val(Rb) + val(Ri)*S`, where val(Rb) and val(Ri) refer to the value of registers Rb and Ri respectively, D is a constant, and S is a constant of value 1, 2, 4, or 8.

**Sign extension and zero extension:**

| | |
|---|---|
| movzlq $S$, $D$ | copy 4-byte-sized $S$ to 8-byte-sized $D$ and fill in the higher order 4 bytes of $D$ with zero bytes. |
| movslq $S$, $D$ | copy 4-byte-sized $S$ to 8-byte-sized $D$ and sign extend the higher order 4 bytes of $D$, i.e. fill with 0s if $S$'s sign bit is zero and fill with 1s if $S$'s sign bit is one. |

**Basic Arithmatic instructions that you might not remember:**

| | |
|---|---|
| sal / shl $k$, $D$ | Left shift destination $D$ by $k$ bits |
| sar | Arithmatic right shift destination $D$ by $k$ bits |
| shr | Logical right shift destination $D$ by $k$ bits |

**Jump instructions:**

Jump instruction following `cmp S, D`:

| | |
|---|---|
| jmp | Unconditional jump |
| je | Jump if D is equal to S |
| jne | Jump if D is not equal to S |
| jg | Jump if D is greater than S (signed) |
| jge | Jump if D is greater or equal than S (signed) |
| jl | Jump if D is less than S (signed) |
| jle | Jump if D is less or equal than S (signed) |
| ja | Jump if D is above S (unsigned) |
| jae | Jump if D above or equal S(unsigned) |
| jb | Jump is D is below S (unsigned) |
| jbe | Jump if D is below or equal S (unsigned) |

## 4.3 Calling convention

**Argument Passing:**

| Which argument | Stored in register |
|---|---|
| 1 | %rdi |
| 2 | %rsi |
| 3 | %rdx |
| 4 | %rcx |
| 5 | %r8 |
| 6 | %r9 |
| 7 and up | passed on stack |

**Return value** (if any) is stored in %rax

**Caller save registers:** %rax, %rcx, %rdx, %rdi, %rsi, %r8-11

**Callee save registers:** %rbx, %rbp, %r12-15

# Appendix: ASCII

The following table contains the 128 ASCII characters.

| Oct | Dec | Hex | Char | Oct | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|
| 000 | 0 | 00 | NUL '\0' | 100 | 64 | 40 | @ |
| 001 | 1 | 01 | SOH (start of heading) | 101 | 65 | 41 | A |
| 002 | 2 | 02 | STX (start of text) | 102 | 66 | 42 | B |
| 003 | 3 | 03 | ETX (end of text) | 103 | 67 | 43 | C |
| 004 | 4 | 04 | EOT (end of transmission) | 104 | 68 | 44 | D |
| 005 | 5 | 05 | ENQ (enquiry) | 105 | 69 | 45 | E |
| 006 | 6 | 06 | ACK (acknowledge) | 106 | 70 | 46 | F |
| 007 | 7 | 07 | BEL '\a' (bell) | 107 | 71 | 47 | G |
| 010 | 8 | 08 | BS '\b' (backspace) | 110 | 72 | 48 | H |
| 011 | 9 | 09 | HT '\t' (horizontal tab) | 111 | 73 | 49 | I |
| 012 | 10 | 0A | LF '\n' (new line) | 112 | 74 | 4A | J |
| 013 | 11 | 0B | VT '\v' (vertical tab) | 113 | 75 | 4B | K |
| 014 | 12 | 0C | FF '\f' (form feed) | 114 | 76 | 4C | L |
| 015 | 13 | 0D | CR '\r' (carriage ret) | 115 | 77 | 4D | M |
| 016 | 14 | 0E | SO (shift out) | 116 | 78 | 4E | N |
| 017 | 15 | 0F | SI (shift in) | 117 | 79 | 4F | O |
| 020 | 16 | 10 | DLE (data link escape) | 120 | 80 | 50 | P |
| 021 | 17 | 11 | DC1 (device control 1) | 121 | 81 | 51 | Q |
| 022 | 18 | 12 | DC2 (device control 2) | 122 | 82 | 52 | R |
| 023 | 19 | 13 | DC3 (device control 3) | 123 | 83 | 53 | S |
| 024 | 20 | 14 | DC4 (device control 4) | 124 | 84 | 54 | T |
| 025 | 21 | 15 | NAK (negative ack.) | 125 | 85 | 55 | U |
| 026 | 22 | 16 | SYN (synchronous idle) | 126 | 86 | 56 | V |
| 027 | 23 | 17 | ETB (end of trans. blk) | 127 | 87 | 57 | W |
| 030 | 24 | 18 | CAN (cancel) | 130 | 88 | 58 | X |
| 031 | 25 | 19 | EM (end of medium) | 131 | 89 | 59 | Y |
| 032 | 26 | 1A | SUB (substitute) | 132 | 90 | 5A | Z |
| 033 | 27 | 1B | ESC (escape) | 133 | 91 | 5B | [ |
| 034 | 28 | 1C | FS (file separator) | 134 | 92 | 5C | \ '\\' |
| 035 | 29 | 1D | GS (group separator) | 135 | 93 | 5D | ] |
| 036 | 30 | 1E | RS (record separator) | 136 | 94 | 5E | ^ |
| 037 | 31 | 1F | US (unit separator) | 137 | 95 | 5F | _ |
| 040 | 32 | 20 | SPACE | 140 | 96 | 60 | ` |
| 041 | 33 | 21 | ! | 141 | 97 | 61 | a |
| 042 | 34 | 22 | " | 142 | 98 | 62 | b |
| 043 | 35 | 23 | # | 143 | 99 | 63 | c |
| 044 | 36 | 24 | $ | 144 | 100 | 64 | d |
| 045 | 37 | 25 | % | 145 | 101 | 65 | e |
| 046 | 38 | 26 | & | 146 | 102 | 66 | f |
| 047 | 39 | 27 | | 147 | 103 | 67 | g |
| 050 | 40 | 28 | ( | 150 | 104 | 68 | h |
| 051 | 41 | 29 | ) | 151 | 105 | 69 | i |
| 052 | 42 | 2A | * | 152 | 106 | 6A | j |
| 053 | 43 | 2B | + | 153 | 107 | 6B | k |
| 054 | 44 | 2C | , | 154 | 108 | 6C | l |
| 055 | 45 | 2D | - | 155 | 109 | 6D | m |
| 056 | 46 | 2E | . | 156 | 110 | 6E | n |
| 057 | 47 | 2F | / | 157 | 111 | 6F | o |
| 060 | 48 | 30 | 0 | 160 | 112 | 70 | p |
| 061 | 49 | 31 | 1 | 161 | 113 | 71 | q |
| 062 | 50 | 32 | 2 | 162 | 114 | 72 | r |
| 063 | 51 | 33 | 3 | 163 | 115 | 73 | s |
| 064 | 52 | 34 | 4 | 164 | 116 | 74 | t |
| 065 | 53 | 35 | 5 | 165 | 117 | 75 | u |
| 066 | 54 | 36 | 6 | 166 | 118 | 76 | v |
| 067 | 55 | 37 | 7 | 167 | 119 | 77 | w |
| 070 | 56 | 38 | 8 | 170 | 120 | 78 | x |
| 071 | 57 | 39 | 9 | 171 | 121 | 79 | y |
| 072 | 58 | 3A | : | 172 | 122 | 7A | z |
| 073 | 59 | 3B | ; | 173 | 123 | 7B | { |
| 074 | 60 | 3C | < | 174 | 124 | 7C | \| |
| 075 | 61 | 3D | = | 175 | 125 | 7D | } |
| 076 | 62 | 3E | > | 176 | 126 | 7E | ~ |
| 077 | 63 | 3F | ? | 177 | 127 | 7F | DEL |