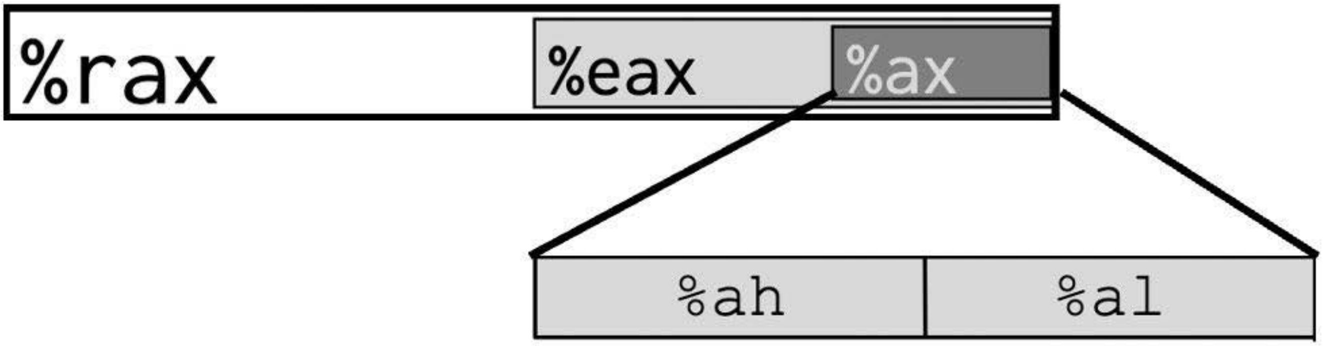
x86-64 calling conventions:

Argument registers:	%rdi, %rsi, %rdx, %rcx, %r8, %r9
Preserved registers:	%rbp, %rbx, %r12, %r13, %r14, %r15
Return value:	%rax, %rdx



mov	

Potpourri of déjà-vu multiple choices [6 points]

1. **(2 points)** How many times does function execv() return if there are no errors?

○ 0 times ○ 2 times

○ 1 time ○ 3 times

2. **(2 points)** What is the difference between the %rbx and the %ebx register on an x86-64 machine?

○ %ebx refers to only the high order 32 bits of the %rbx register

○ %ebx refers to only the low order 32 bits of the %rbx register

○ they are totally different registers

○ nothing, they are the same register

3. **(2 points)** What do you **NOT** expect to find in a Stack Frame?

○ Saved registers ○ Function name

○ Local (automatic) variables ○ Return address

Malloc [20 points]

1. (4 points) Given a system with a small amount of memory, explain which of the following memory allocation algorithm you would use: next-fit or best-fit. Make sure you explain the advantages and disadvantages of each of them.

2. **(16 points)** Consider an allocator implementation with the following characteristics:

- The **first-fit** free algorithm is used to allocate data.
- All blocks have a header with a size and a pointer to the previous block.
- The header is 16B (2*8bytes) in size.
- Positive sizes indicate the block is allocated, and negative sizes indicate it is free.
- All freed blocks are immediately coalesced if possible.
- When a block is split, the lower (first) part of the block becomes the allocated part and the upper (second) part becomes the new free block.
- If the heap doesn't have enough space to hold the data, it grows by the minimum amount needed to fit the data. Always successfully.

For the given a heap representation, only the metadata is displayed. E.g., the following heap contains an allocated block of size 16, followed by a free block of size 32. The top row contains memory addresses, and the bottom row contains the values stored at those memory addresses.

Address	0xa000	0xa008	...	0xa020	0xa028	...
Value	16	0x0000	...	-32	0xa000	...

a. (4 points) Assuming an **initially empty heap**, and given the current state of the heap represented below, which of the malloc sequence was executed?

Address	0xa000	0xa008	...
Value	-64	0x0000	...

○
```
p0 = malloc(32);
free(p0);
p0 = malloc(32);
free(p0);
```

○
```
p0 = malloc(16);
p1 = malloc(32);
free(p0);
free(p1);
```

○
```
p0 = malloc(64);
```

○
```
p0 = malloc(32);
free(p0);
p1 = malloc(16);
free(p1);
```

b. (4 points) Assuming the heap starts as drawn in the previous question, and given the final state of the heap represented below, which of the malloc sequence was executed?

Address	0xa000	0xa008	...	0xa030	0xa038	...
Value	32	0x0000	...	-16	0xa000	...

○
```
p0 = malloc(32);
free(p0);
p0 = malloc(16);
free(p0);
```

○
```
p0 = malloc(32);
free(p0);
p0 = malloc(16);
```

○ `p0 = malloc(32);`

○ `p0 = malloc(16);`

c. (4 points) Assuming the heap starts as drawn above (b), if the following malloc executes, what is the value stored in p1?

```
p1 = malloc(16);
```

○ 0xa010 ○ 0xa018 ○ 0xa020 ○ 0xa028

○ 0xa030 ○ 0xa038 ○ 0xa040 ○ 0xa048

d. (4 points) Assuming the heap starts as drawn above (b), which value can fill the blank to successfully free the first block?

`free(_________);`

○ 0xa000 ○ 0xa008 ○ 0xa010 ○ 0xa018

○ 0xa020 ○ 0xa028 ○ 0xa030 ○ 0xa038

Assembly [25 points]

3. (9 points) Assembly and Reverse-Engineering

Consider the following assembly dump

```
0000000000001139 <bloop>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 83 ec 10             sub    $0x10,%rsp
    1141:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
    1145:       48 83 7d f8 29          cmpq   $0x29,-0x8(%rbp)
    114a:       7f 1b                   jg     1167 <bloop+0x2e>
    114c:       48 8b 05 dd 2e 00 00    mov    0x2edd(%rip),%rax
    1153:       48 89 c6                mov    %rax,%rsi
    1156:       48 8d 3d b5 0e 00 00    lea    0xeb5(%rip),%rdi
    115d:       b8 00 00 00 00          mov    $0x0,%eax
    1162:       e8 c9 fe ff ff          callq  1030 <printf@plt>
    1167:       90                      nop
    1168:       c9                      leaveq
    1169:       c3                      retq
```

a. **(2 points)** How many function arguments are defined in the above function bloop?

○ 0 ○ 1 ○ 2 ○ 3

b. **(2 points)** How many local variables (not arguments) are declared in the above function bloop?

○ 0 ○ 1 ○ 2 ○ 3

c. **(2 points)** Which of the following multiplies the value within the rax register by 9?

○ `lea (,rax,9), rax` ○ `lea (rax,rax,8), rax`

○ `lea (rax,rax,9), rax` ○ `lea 9(rax), rax`

d. **(2 points)** If I saw `mov %rax, -0x8(%rbp)`, and given `char` is 1B, `short` 2**B**, `int` 4B, and `long` 8B, I would say that this local variable is what integer type?

○ `int` ○ `long` ○ `char` ○ `short`

e. **(1 points) How many loops does the function above have**?

○ 0 ○ 1 ○ 2 ○ 3

4. (16 points) Reverse Engineering x86-64 to C. Consider the following x86-64 assembly and fill-in the blanks in the C code. Use the following variable-to-register mapping:

```
n <==> %rdi          y <==> %rsi          x <==> %ecx
result <==> %eax     i <==> %edx
```

You may only use the symbolic variables such as `i`, `x`, and `result` in your C expressions ---- **do not use register names**.

```
loop:
        movl    $10, %ecx
        movl    $0, %eax
        movl    $0, %edx
        jmp     .L2
.L3:
        leaq    (%rax,%rax,4), %rsi
        movq    %rcx, %rax
        leaq    (%rcx,%rsi), %rcx
        addq    $1, %rdx
.L2:
        leaq    (%rdi,%rdi), %rsi
        cmpq    %rdx, %rsi
        jg      .L3
        ret
```

```
long loop(long n)
{

  long i = ________________;


  long result = ________________;


  long x = ________________;


  while (________________)
  {

      long y = ________________;


      result = ________________;


      x = ________________;


      i++;
  }


  return ________________;
}
```

Buffer Overflow [15 points]$□□□□3As□d□dpassword12345□□□□5saKLJ98hy□ □□□0□□□□a□□□

5. **(15 points)** Consider a **program** containing this poor-quality code, procedure vulnerable has the following disassembled form on a x86-64 machine:

```
void vulnerable(char t) {
    char password[6];
    char name[4];
    gets(name);
    password[0]='H';
    password[1]='e';
    password[2]='l';
    password[3]='l';
    password[4]='o';
    password[5]=t;
    printf("You cannot know my
password %s!\n", name);
    // here
}
```

```
vulnerable: # @vulnerable
  pushq %rbp
  movq %rsp, %rbp
  subq $0x10, %rsp
  movb %dil, -1(%rbp) # dil→8lsb of rdi
  leaq -0xb(%rbp), %rdi
  callq gets
  movb $72, -7(%rbp) # H
  movb $101, -6(%rbp)# e
  movb $108, -5(%rbp)# l
  movb $108, -4(%rbp)# l
  movb $111, -3(%rbp)# o
  movb -1(%rbp), %al
  movb %al, -2(%rbp)
  leaq -0xb(%rbp), %rsi
  mov  $0x400104, %rdi # "You cannot …"
  callq printf
  addq $0x10, %rsp
  popq %rbp
  retq
```

For the following questions, recall that:

- `gets` is the standard C library routine.
- x86-64 machines are **little-endian**.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).

Consider the case where procedure `vulnerable` is called with argument t equal to '\0', and we type "Luis" in response to gets.

a. **(3 points)** Which elements of array `password` were overwritten **when gets is called**?

○ None　　○ Only `password`[0]

○ `password`[0] and [1]　　○ `password`[0], [1], and [2]

b. **(3 points)** Which of the following stack values were corrupted?

○ Arguments　　○ Saved registers

○ Arguments and Saved registers　　○ None of the listed options

c. **(3 points)** What (exactly) was printed by the function when printf was executed?

Answer:

d. **(3 points)** What would **NOT help** with preventing code injection attacks?

○ Add a canary to the stack

○ Make the stack larger than it needs to be

○ Make the stack not executable

○ Randomize the memory address of the stack

e. **(3 points)** Which of the following cases represents a buffer overflow?

○ Writing a string of length 5 to an array of chars of size 6

○ Reading 20B from an array of size 10B

○ Writing 5B of code into a stack buffer with 10B of capacity

○ Executing code in the stack

Creation and execution of programs [24 points]

6. **(6 points)** Compiling, linking, loading. For each question, determine at which stage the action happens.
   a. (2 points) Determine the location of functions in other object files?

      ○ Compiling ○ Loading

      ○ Linking ○ Assembling

   b. (2 points) Place shared libraries in memory?

      ○ Compiling ○ Loading

      ○ Linking ○ Assembling

   c. (2 points) Determine the offset of a stack variable?

      ○ Compiling ○ Loading

      ○ Linking ○ Assembling

7. **(9 points)** Answer the questions for the code below.

```
static int x = 0, y = 5;

int what_is_this(void) {
    x = x + y;
    y = y + 1;
    return y;
}

int main(void) {
    int v = what_is_this();
    printf("%d\n", v);
    return v;
}
```

   a. **(3 points)** With respect to the Linker, which of the following is a **global** symbol?

      ○ what_is_this ○ y ○ x ○ v

   b. **(3 points)** With respect to the Linker, which of the following is a **local** symbol?

      ○ v ○ what_is_this ○ x ○ main

c. **(3 points)** With respect to the Linker, which of the following is **NOT** registered as a symbol?

○ v ○ what_is_this ○ x ○ main

8. (9 points) For each question, read the code – assume all code is given!

a. (3 points) What type of error will the code produce?

```
int main() {
    int a[4] = {0};
    for (int i=0 ; i<10000 ; i++) {
        a[i] = i;
    }
    return 0;
}
```

○ Runtime error ○ Compilation error

○ Linking error ○ Loading error

b. (3 points) What type of error will the code produce?

```
int main() {
    int a[4] = {0};
    for (int i=0 ; i<4 ; i++) {
        a[i] = add(a[1], i);
    }
    return 0;
}
```

○ Runtime error ○ Compilation error

○ Linking error ○ Loading error

c. (3 points) What type of error will the code produce?

```
int add(int a, int b);
int main() {
    int a[4] = {0};
    for (int i=0 ; i<4 ; i++) {
        a[i] = add(a[1], i);
    }
    return 0;
}
```

○ Runtime error ○ Compilation error

○ Linking error ○ Loading error