Fill out the bottom of this page with your computing ID.
Write your computing ID at the top of each page in case pages get separated.

Linux X86-64 calling convention reminder:

- first argument: `%rdi`
- second argument: `%rsi`
- third argument: `%rdx`
- fourth argument: `%rcx`
- fifth argument: `%r8`
- six argument: `%r9`
- return value: `%rax`
- return address: on stack
- seventh argument: on stack after return address
- eight argument: on stack after seventh argument

X86-64 registers reminder:

- `%rax` (64-bit), `%eax` (lower 32 bits), `%ax` (lower 16 bits), `%al` (lower 8 bits)
- (and similar for `%rbx`, `%rcx`, `%rdx`)
- `%rsi` (64-bit), `%esi` (lower 32 bits), `%si` (lower 16 bits), `%sil` (lower 8 bits)
- (and similar for `%rbp`, `%rsp`, `%rdi`)
- `%r8` (64-bit), `%r8d` (lower 32 bits), `%r8w` (lower 16 bits), `%r8b` (lower 8 bits)
- (and similar for `%r9` through `%r15`)

AT&T syntax reminder:

- `0x1234(%r9,%r10,4)` = memory at `0x1234` + `%r9` + `%r10` × 4
- `$0x12345678` = constant
- `0x12345678` = memory at `0x12345678`
- source, destination

On X86-64, popping from the stack reads the value located at (%rsp), then adds 8 to the stack pointer.

1. Consider the following C code:

```
int process_input(const char *);

int read_input() {
    char buffer[50];
    int c;
    do {
        int i = 0;
        while ((c = getchar()) != '\n') {
            buffer[i] = c;
            i += 1;
        }
        buffer[i] = '\0';
    } while (process_input(buffer) == -1);
    return process_input(buffer);
}
```

`getchar()` is a C standard library function that reads one character (one byte) from `stdin`. When assembled, this function has the following stack layout, from highest to lowest address:

- return address for `read_input` (8 bytes)
- saved value of `%rbp` (8 bytes)
- saved value of `%rbx` (8 bytes)
- unused space (8 bytes)
- buffer (50 bytes)
- unused space (14 bytes)

(a) (3 points) We can perform a stack smashing attack on this function. How many bytes into the input should we place the address we wish the program to jump to? ◯ 50 ◯ 58 ◯ 64 ◯ 66 ◯ 74 ◯ 88

(b) (6 points) Suppose we determine that the stack pointer is located `0x7000 0100` at the very beginning of `read_input`. Suppose we place a 20 byte nop-sled at the beginning of the input, followed by our injected machine code ('shellcode'). What address should we replace the return address with? You may write your answer as an arithmetic expression(e.g. `0x4000 - 72`).

(b) ____________________

(c) (3 points) When exploiting this buffer overflow, what bytes can we **not** have in our machine code? **Select all that apply.**
◯ `0x00` (`\0`) ◯ `0x0a` (`\n`) ◯ `0x20` (a space character) ◯ `0xFF`

(d) (3 points) When exploiting this buffer overflow, what bytes can we **not** have in our replacement for the return address? **Select all that apply.**
◯ `0x00` (`\0`) ◯ `0x0a` (`\n`) ◯ `0x20` (space character) ◯ `0xFF`

2. Consider the following C++ code:

```
struct Vulnerable {
    char description[128];
    long defaultValue;
    long *data;
    int size;
};

void ClearVulnerable(Vulnerable *v) {
    for (int i = 0; i < v->size; ++i) {
        v->data[i] = v->defaultValue;
    }
}
```

Assume `Vulnerable` is laid out in memory without any unused space (and with fields placed in the order declared) and that chars take up one byte, pointers take up 8, and ints take 4.

Suppose an attacker can overflow the `description` variable and then cause ClearVulnerable() to be run.

The attacker wants to use this buffer overflow to overwrite an arbitrary writeable memory location with a chosen value.

(a) (5 points) How many bytes from the beginning of the `description` should the attacker put the address to overwrite?

◯ 128 ◯ 136 ◯ 144 ◯ 148 ◯ none of these

(b) (5 points) How many bytes from the beginning of the `description` should the attacker put the value to place in the address?

◯ 128 ◯ 136 ◯ 144 ◯ 148 ◯ none of these

3. (8 points) Suppose an attacker manages to place machine code they want to run in vulnerable program's a global variable at address `0x700000`. They discover that the binary contains the following "stub" for `free()`:

```
0000000000400400 <free@plt>:
  400400: ff 25 12 0c 20 00  jmpq   *0x200c12(%rip) # 601018
                                      <_GLOBAL_OFFSET_TABLE_+0x18>
  400406: 68 00 00 00 00     pushq  $0x0
  40040b: e9 e0 ff ff ff     jmpq   4003f0 <_init+0x28>
```

(Recall that objdump outputs addresses in hexadecimal.) Suppose the vulnerable program allows the attacker to write the *address* of their injected code, `0x700000`, to a memory location of the attacker's choice. What memory location should the attacker choose to make future calls to the free stub `free@plt` invoke their injected code?

3. ____________

4. (25 points) Suppose an application has the following "gadgets" at the indicated memory addresses:

   - at `0x400`:
     ```
     popq %rdi
     popq %rbx
     ret
     ```
   - at `0x500`:
     ```
     popq %rax
     ret
     ```
   - at `0x600`:
     ```
     syscall
     ```
   - at `0x700`:
     ```
     pushq %rax
     call *(%rcx)
     ```
   - at `0x800`:
     ```
     popq %rdx
     ret
     ```

   Suppose an attacker wants to use these gadgets to perform the equivalent of

   ```
   movq $0x410000, %rdi
   movq $0x25, %rax
   syscall
   ```

   using a stack buffer overflow on a system which implements the write XOR execute mitigation. Assuming the attacker overwrites a vulnerable return address at address `0x80000`. Indicate what the attacker should overwrite the surrounding 8-byte values with to perform the equivalent of the above assembly using the above gadgets and return- or jump-oriented programming. You may leave an entry blank or write the word "any" if the value at that memory location that does not matter. You will not need to use all of the available lines to answer the question.

address	8-byte value at address
`0x80030`	____________
`0x80028`	____________
`0x80020`	____________
`0x80018`	____________
`0x80010`	____________
`0x80008`	____________
(vulnerable return address) `0x80000`	____________
`0x7FFF8`	____________

5. (12 points) Suppose a function calls `printf` passing a pointer to a buffer containing user input as the first argument and no other (intentional) arguments. The stack contains the following when printf is called (from highest to lowest address):

   - return address for function that calls printf (8 bytes) at `0x7F00C0`
   - buffer containing user input/printf argument (128 bytes $(16 \cdot 8)$) at `0x7F0040`
   - return address for `printf` (8 bytes) at `0x7F0038`

   Recall that %c will print 1 byte and arguments on the stack are always 8 bytes.

   What should that input (stored in the stack buffer and passed to printf) be in order for `printf` to write the value `16` to memory location `0x601068`? **Select all that apply.**

◯ `%xAAAAAAAA%n\x68\x10\x60\x00\x00\x00\x00\x00`

◯ `\x68\x10\x60\x00\x00\x00\x00\x00%c%c%c%c%c%cAA%n` (contains six %c)

◯ `AAAAAAAABBBBBBBB\x68\x10\x60\x00\x00\x00\x00\x00%n`

◯ `AAAAAAAABBBBBBBB%x%x%x%n\x68\x10\x60\x00\x00\x00\x00\x00` (contains three %x)

◯ `%c%c%c%c%c%c%c%c%c%cAAAAAA%nAAAA\x68\x10\x60\x00\x00\x00\x00\x00` (contains ten %c)

◯ `%c%c%c%c%c%c\x68\x10\x60\x00\x00\x00\x00\x00AA%n` (contains six %c)

6. (3 points) If an attacker can overwrite a saved frame pointer, what can they replace it with to cause the program to execute machine code at address A?

   ◯ the address A itself

   ◯ the address of a buffer containing the address A

7. (4 points) Suppose an attacker overwrites a linked list node just before the program does `node->next->prev = node->prev;`. If they make `node->next` contain a pointer to an array of function pointers, then what can they make `node->prev` contain to cause the program execute machine code at address A?

   ◯ the address A itself

   ◯ the address A, minus a constant that depends on the layout of list nodes

   ◯ the address of a buffer containing the address A

8. For each of the following exploits, identify which, if any, of the following mitigations would **prevent** them. Assume that address space layout randomization includes using a position-independent executable, and that there are no information leaks unless otherwise stated.

   (a) (5 points) Using a format string exploit to overwrite a global variable.

   ◯ write XOR execute

   ◯ address space layout randomization (ASLR) with no information leaks

   ◯ ASLR, if the program outputs a stack address

   ◯ placing guard pages around all heap allocations

   ◯ bounds checking like in "Baggy Bounds Checking"

   ◯ stack canaries (AKA stack cookies)

(b) (5 points) Using a stack buffer overflow to overwrite a local variable stored immediately after the buffer.

- ◯ write XOR execute
- ◯ address space layout randomization (ASLR) with no information leaks
- ◯ ASLR, if the program outputs a stack address
- ◯ placing guard pages around all heap allocations
- ◯ bounds checking like in "Baggy Bounds Checking"
- ◯ stack canaries (AKA stack cookies)

(c) (5 points) Replacing the return address with the address of code an attacker placed in a on-stack buffer, then allowing the program to return to this address.

- ◯ write XOR execute
- ◯ address space layout randomization (ASLR) with no information leaks
- ◯ ASLR, if the program outputs a stack address
- ◯ placing guard pages around all heap allocations
- ◯ bounds checking like in "Baggy Bounds Checking"
- ◯ stack canaries (AKA stack cookies)

(d) (5 points) Overflowing a stack-allocated buffer to overwrite a return address, then using return-oriented programming

- ◯ write XOR execute
- ◯ address space layout randomization (ASLR) with no information leaks
- ◯ ASLR, if the program outputs a stack address
- ◯ placing guard pages around all heap allocations
- ◯ bounds checking like in "Baggy Bounds Checking"
- ◯ stack canaries (AKA stack cookies)

(e) (5 points) Using a use-after-free vulnerability to replace a VTable pointer to cause the program to run code in an attacker-controlled stack buffer.

- ◯ write XOR execute
- ◯ address space layout randomization (ASLR) with no information leaks
- ◯ ASLR, if the program outputs a stack address
- ◯ placing guard pages around all heap allocations
- ◯ bounds checking like in "Baggy Bounds Checking"
- ◯ stack canaries (AKA stack cookies)

(f) (5 points) Using a heap buffer overflow to overwrite a pointer used by malloc to point to a global variable, then allowing the malloc implementation write an attacker-chosen value to that variable

- ◯ write XOR execute
- ◯ address space layout randomization (ASLR) with no information leaks
- ◯ ASLR, if the program outputs a stack address
- ◯ placing guard pages around all heap allocations
- ◯ bounds checking like in "Baggy Bounds Checking"
- ◯ stack canaries (AKA stack cookies)