“It All Depends Upon How You Look At It” Lab

### Introduction

During this lab you will learn:

* How the central processing unit (CPU) knows whether the contents of a particular word in memory is an instruction or data, and, if the latter, the type of the data.
* How to interpret character and floating point data.
* About some of the consequences of a CPU’s “knowledge” of the contents of memory.

### Starters

Type your answers into the text box following each question. You may work with someone else on this lab, and submit one copy of this, including your answers, on Canvas.

*What are the names of everyone who worked on this lab with you, including yourself?*

|  |
| --- |
|  |

### Code vs. Data

Regarding the buffer overflow exploit we discussed in class (using the example below) and referring to [Buffer Overflow](https://en.wikipedia.org/wiki/Buffer_overflow) (or other information on buffer overflow exploits you find on the internet, briefly explain how data can become code. Given this, does a CPU “know” the difference between data and code?

main() {

int ret;

ret = func();

if (!ret) {

….

}

else {

…

}

return 0;

}

int func() {

char buffer[1024];

/\* Read characters into buffer from some input source using read() \*/

/\* Compute something \*/

return 0;

}

|  |
| --- |
|  |

### Floating-Point Data

* + 1. Floating point values are treated by a CPU the way we treat numbers in scientific notation. Consider an 8-bit floating point number format in which the most significant bit (7) is the number’s sign bit (S), bits 4—6 are the base 4 exponent (E) in signed two’s complement format, and bits 0—3 are the mantissa (M) in unsigned format. A binary value in this format can be converted to decimal using this formula:

-1S \* 4E \* M

* + 1. For example, the binary value 0X2C (00101100) is
       1. -10 \* 42 \* 12 = 1 \* 16 \* 12 = 192
    2. Referring to the Wikipedia article on half-precision floating point format ( <https://en.wikipedia.org/wiki/Half-precision_floating-point_format> ), perform the following conversions.

Convert 0XE1 to decimal, interpreting the binary value as

1. A floating-point value in the format described above
2. A signed two’s complement value
3. An unsigned value

|  |
| --- |
|  |

### Representing Characters

Referring to [ASCII Table](https://www.ascii-code.com/), convert the characters of

**Hello world!**

to ASCII, expressing each character as two hexadecimal digits. The first two character conversions are completed for you.

|  |
| --- |
| 48 65 |

Research Unicode. Why is it “superior” to ASCII? Give a specific example.

|  |
| --- |
|  |

### Representing Strings

Refer to the Wikipedia articles [Null-Terminated String](https://en.wikipedia.org/wiki/Null-terminated_string) and [Endianness](https://en.wikipedia.org/wiki/Endianness). Consider a 32-bit memory (i.e., four bytes per word) containing this data:

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **D** |
| **E** | **F** | **G** | **H** |
| **\0** | **\0** | **\0** | **\0** |

where **\0** represents the null character.

What string does this represent on a little-endian machine?

|  |
| --- |
|  |

What string does this represent on a big-endian machine?

|  |
| --- |
|  |