Basic Architecture

A computer is simply a device that is designed to understand a sequence of 1's and 0's through an array of electrical circuits, and programmable hardware to take in data from a source (like a piece of code written and compiled by a user) and perform different computations or translations of that data.

Those sequences of 1's and 0's can be created in many different ways, and is a constant focus in software design to make a coherent and usable form of translation (be it through an interpreter or compiler) so that a user can write a human-readable string of characters/numbers/commands instead of writing raw binary (or hexadecimal) code.

This project will primarily focus on the foundational landscape of nearly all modern computers today:

- The C Programming Language

While most readers will likely be more familiar with more "high-level" programming languages like Python, Java, Javascript, etc, the intricate "ins and outs" of C are lesser-known and partially due to that, a lot of C code is wildly vulnerable to exploiting.

Now onto the basics:

Like mentioned previously, a computer takes in a sequence of data through physical 'wires' (copper traces in a board usually) but where does that sequence of data come from? Lets start with a very basic overview of a computer, starting with some terms:

- CPU
    - Central Processing Unit
        - This component is often referred to as the "brain" of the computer. Its fundamental purpose is to
            - FETCH (READ) operations from SRAM/Cache (will discuss when we get to stack/instruction pointers)
            - DECODE that data into operations the cpu is designed to understand
            - EXECUTE the operations determined by the decoding
            - WRITE any relevant data back to memory
    - Cache/SRAM/CPU Memory
        - Sub-component of a processor.
        - This is where current program instructions are cached for faster access when operation is deterministic
    - Registers
        - Another Sub-component of a processor

        - Can be viewed as "memory" since they store data (have data written to them), but each register only holds a single 'entry'
            - This project uses a 64-bit operating system and code, which means all memory accesses are 64-bit (capable)
            - Follow-up reading in next section. Helpful research topics would be Intel Architecture or AARCH64 (ARM v8) registers.
                - Keywords: Intel 64-bit Registers, Assembly Language Registers in Linux

- Memory (RAM)
    - This is where data is stored/read from/written to relevant for program/computer operation
    - Operating system organizes this large mass of storage into address spaces/blocks. This is where running processes access data from in most cases.
    - Data is accessed at (up-to) the bit-width of the cpu architecture/operating system.
        - Meaning an access from RAM on a 64-bit operating system is accessed on a 64-bit basis.
        - This lines up with the width of the CPU's registers for ease of reading/writing to/from memory.

- Hard Drive (Non-volatile storage)
    - This isn't very important for the project, so will just touch on high level. This is where data can be stored and retained when a computer shuts down. Along with general media/data includes the Operating System and the programs that the computer runs.

More Focused Memory Architecture

Now with the basic structure out of the way, lets focus on the memory/data part of a computer

Since we know that a CPU has a defined set of behavior, we can't do much to change what a CPU does from the hardware perspective, all we can really do is tell it what to do with instructions (code that is compiled from a program and fed into the CPU).

Simple code like "take a number and add it to another number, and store it into this register" would take the form of something like

```
MOV RAX,RCX
ADD RAX, 10
MOV RBX,RAX
```

Which in plain english

"Move a value which is stored in register RCX, and store in register RAX"
"Add 10 to the value in RAX and also store in RAX"
"Move that value into register RBX"

This code is in assembly language, which is still considered "human-readable" code, and still undergoes an additional translation (called being assembled, go figure) to be converted into the actual binary code that is stored in a computer's memory. http://ref.x86asm.net/coder64.html

MOV	RAX	RCX
8B	00	01

ADD	RAX	10
03	00	0A

MOV	RBX	RAX
8B	03	00

So the Hexadecimal code for the three lines we see would be

0x8B0001
0x03000A
0x8B0300

Hexadecimal is just an easier-to-read version of binary code, which would then look like

01000101100000000000000001b
00011000000000000000001010b
01000101100000001100000000b

I definitely don't like reading that, so we'll stick from hexadecimal whenever we need to actually refer to assembled code, or memory locations, etc

Now some basics of how this assembled code is actually stored on a computer:

Say we have this program of 3 lines of assembly code, and ignoring all other complexities involved with how the program is compiled/assembled/loaded/etc. Once the computer boots the Operating System, and the program is run, this code is loaded somewhere (anywhere) in the

computer's RAM. The CPU is told where in the system memory this code is, and will begin execution on it. For simplicity lets just say the code is in address 0xABCD0000. So we would see something like this if we can visualize the memory (you will with GDB in the project!)

Address	Data
0xABCD0000	0x8B0001
0xABCD0008	0x03000A
0xABCD0010	0x8B0300

Note the incrementing of the address is by 8 (Bytes) which is 64 bits, so assuming that each instruction is in its own separate cell of memory, each subsequent instruction will be +8 from the previous.

(Note that the example shown isn't as realistic as what the actual system does in terms of storage optimization, but it is close enough for simplicity's sake, sometimes memory will be optimized and data will be compacted based on the compiler options, and operating system/hardware features)

NOW, Regarding security, this is where things get interesting (and important!). Without getting deep into specifics of any actual vulnerabilities, say hypothetically you can somehow modify this code sitting in memory either by overwriting values, or forcing some kind of instability, etc. If you can flip a single bit then you can completely change the behavior of the program! This makes memory fundamentally much more interesting than the CPU, which blindly follows orders (with some exceptions, obviously) so the vast majority of exploits on a computer are usually manipulation of the data or leveraging logical errors in code allowing some kind of attack or misbehavior to take place.

Ref:
https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf

Guts of a C program

With the understanding of some basic building blocks of memory and assembly, now we will cover some fundamental basics of the C language. Noting again that these examples are very

simplistic and meant to give a basic understanding. More research is heavily recommended, and widely available all over the internet.

Using a similar example from previously, lets create a variable, add a number to it, and store in another variable. The C language is designed to make operations like this a lot simpler than previously, and we can do something like this:

```
int main() {
	int value1 = 1;
	int value2 = 0;

	value1 = value1 + 10;
	value2 = value1;

	return 0;
}
```

Now realistically the compiler is going to see this and just immediately create assembly code that just stores the value of 11 into value2, but let's pretend there are no compiler optimizations or anything and the code is just completely interpreted as-is, so that our assembly code will look as close as possible to our previous example, with the addition of the fundamental building blocks of a C function.

```
PUSH RBP
MOV RBP, RSP
MOV RAX,RCX
ADD RAX, 10
MOV RBX,RAX
POP RBP
RET
```

Here we have some new, and some old code. the new code is what the C compiler adds to the assembly program, and is the core behavior of a function entry/exit! The PUSH instruction tells us to put whatever value currently is in the RBP (64-bit Base Pointer) Register onto the stack (more specific details on stack later).

After that, we move the value currently in the RSP (64-bit Stack Pointer) Register into the RBP Register, which is how the C program keeps track of where it is, and where it was previously. By putting the Stack Pointer into the Base Pointer, we have now set up the C program to use any memory it might need to in its own little island, separate from any other currently running process.

Note that we don't use the stack anywhere in this small program, so effectively there isn't much actual reason to set up this stack, but the program does it regardless (cause that's how C works!)

Lets slightly adjust the program to make it a little bit more interesting! Instead of running everything in main(), lets move the arithmetic code into its own function (call it add) and then call that function from main()!

```
int add(int value1, int number_to_add) {

	return value1 + number_to_add;

}
int main() {
	int value1 = 1;
	int value2 = 0;

	value2 = add(value1, 10);
	return 0;
}
```

We would get something like this

```
PUSH RBP
MOV RBP, RSP
MOV RDI,0x1
MOV RSI,0xA
CALL 0xfunction_address <add>
MOV RBX,RAX
POP RBP
RET
```

The way 64-bit C in linux works, is that arguments for functions are stored in certain registers before issuing the CALL opcode. Typically the first argument is RDI (or EDIif it's smaller than 64-bit value). Then RSI (or ESI, again depending on the type width). And so on, I suggest checking the intel reference manual for the remaining order/precedence of the registers. The _return value_ for the function is stored in the EAX register, so when the function returns, we simply grab that value from EAX/RAX and put it wherever it needs to go.

But what does the add() function look like? If we step into the function in GDB it would look surprisingly similar, just using a different register for the added value

```
PUSH RBP
```

```
MOV RBP, RSP
MOV RDX, RSI
MOV RAX,RDI
ADD RAX, RDX
POP RBP
RET
```

Note the initial MOV instructions take values from the registers in the previous block of code, stores in the 'general purpose' registers (i.e. RDX/RAX) then performs the ADD operation, and stores the final result into the RAX register before cleaning up the base pointer (POP RBP) which returns our context to the main() function point.

Ref:
https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf