Year and Semester	: 2022 SPRING
Course Number	: CS-212
Course Title	: Practical Python
Work Number	: HA-17
Work Name	: Final Project Deliverables
Work Version	: Version 2
Long Date	: Sunday, 8 May 2022
Author(s) Name(s)	: Kyle LeDoux

Software Description

For my project, I am working on parsing a "decompiled" binary file. Essentially, the binary is commented with function calls and has labeled section and function headers. The purpose of the project is to direct a vulnerability analyst to where vulnerabilities could be present. For instance, 'cin' in C++ is asking for input. Whenever input is being asked by a program, code mediation is necessary. The analyst wouldn't have to waste time trying to find these instances because the program will direct them specifically where to go.

I also want assembly language calls – like 'call' and 'jmp' – listed out. In C and C++, an 'if statement' will be represented by an assembly language call. The "block" of code assigned to that 'if statement' can potentially be identified within the assembly code. This is important to an analyst who's trying to find the vulnerability in a low-level language, like assembly, from a highlevel language, like C. There's also the potential need to find a vulnerability in an earlier version of a software that was found in an updated version. Assembly calls will be identified and displayed but block identification is not expected to be implemented in the final submission.

Requirements

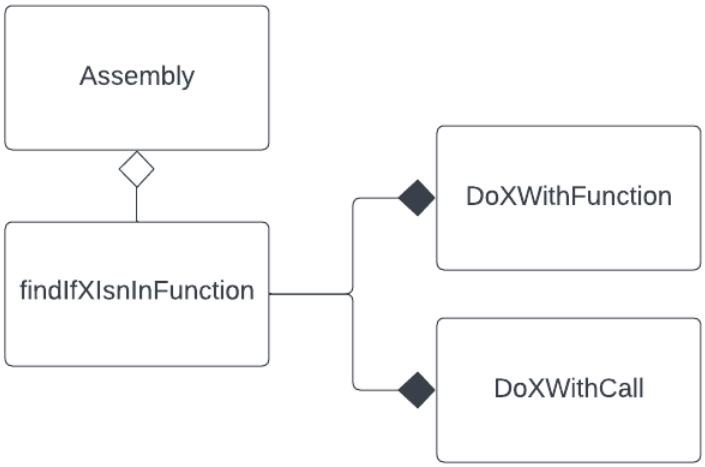
- Fully functional code
- Sections and functions of binary able to be specified and printed
- Table displaying memory location of function calls i.e. printf, scanf, etc. in C code.
- Table showing all assembly calls i.e. call, jmp, jne, etc. in C code. Display exact memory locations and assembly instructions for that call

Design Specifications

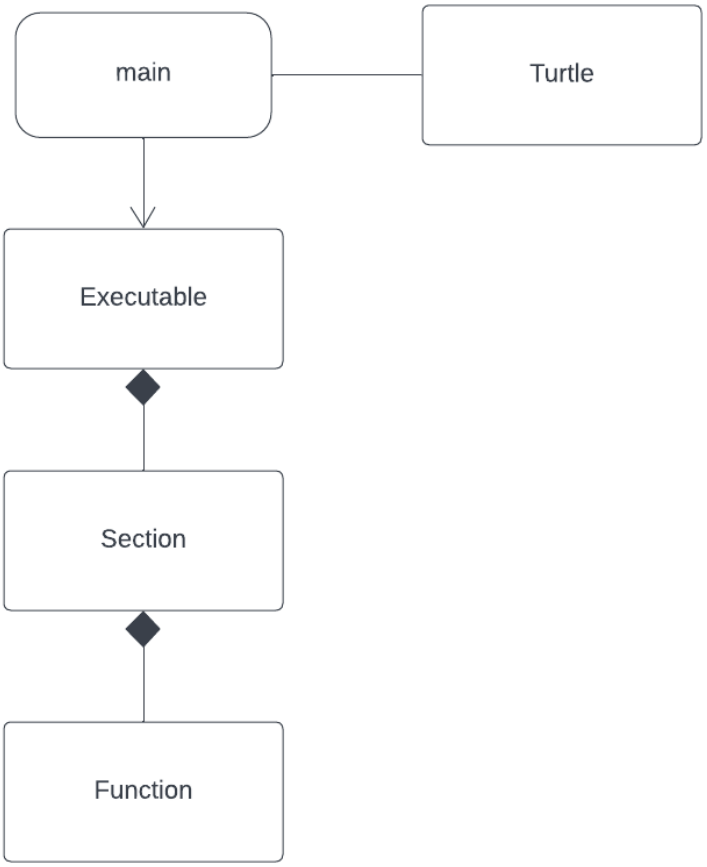
Each assembly file being parsed will have its own object associated with it. Each assembly object must create a list of section objects. If a vulnerability analyst wants to look at the section '.text' specifically, then they should be able to. Each section object will have its own function objects. For example, section '.plt' has two function objects associated with it called

‘<printf@plt-0x10>’ and ‘<printf@plt>’. Every function will have a list of dictionaries for every line. When searching through that list of dictionaries, specific aspects of an assembly line can be accessed. For instance, ‘call’ can be found by comparing an assembly line’s key ‘instruction’ from the dictionary.

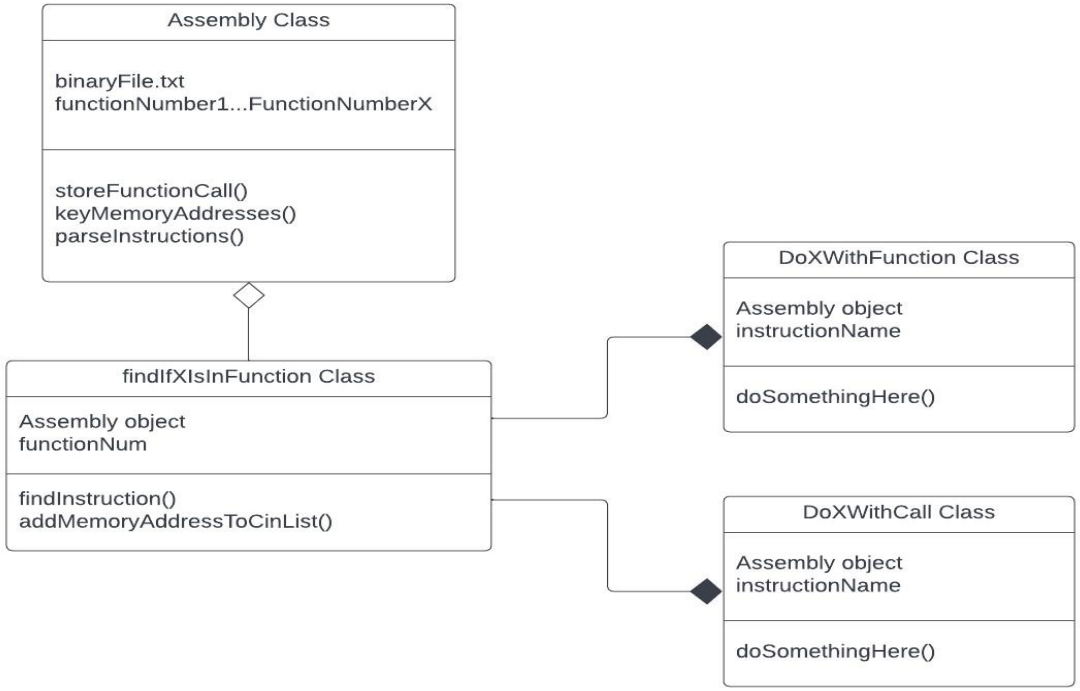
High-Level Initial Design



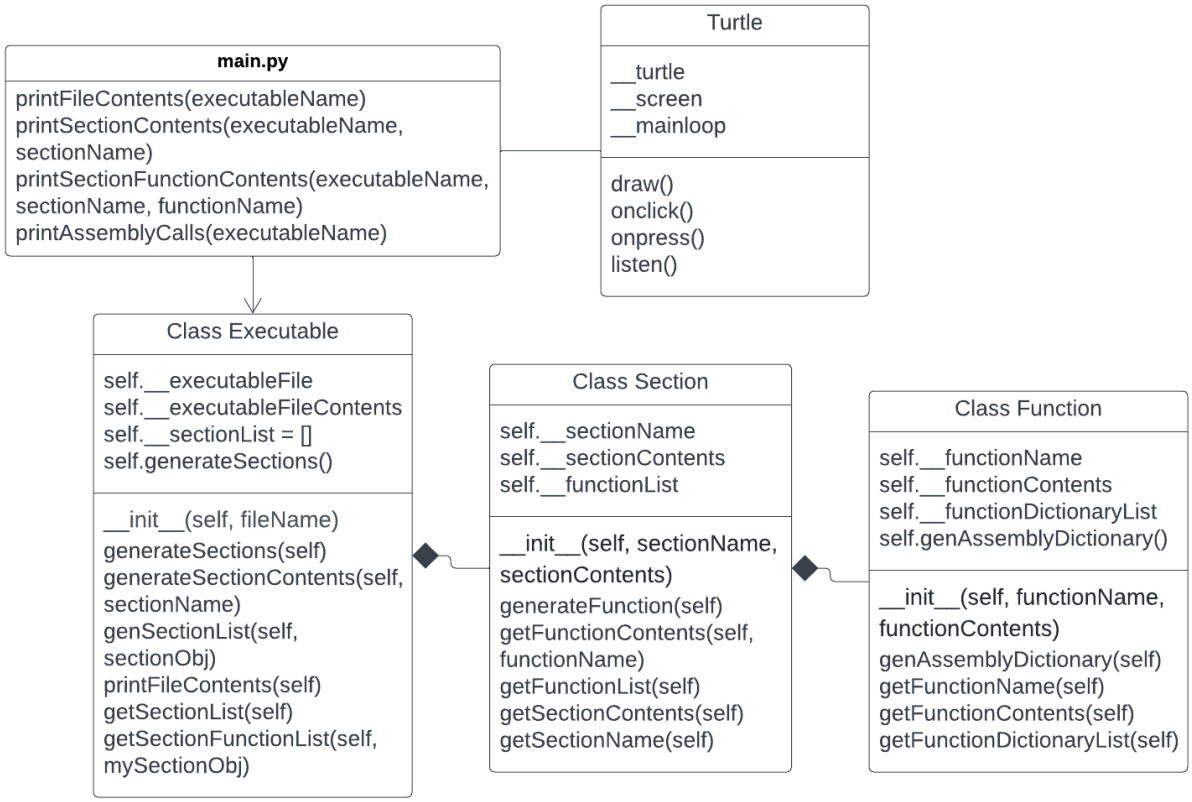
High-Level Final Design



Detailed Initial Design



Detailed Final Design



Class Responsibilities

Class Executable requires the name of the desired executable to be parsed. When called, Class Executable will generate all the section objects and add them to a list of section objects. This class will also return the list of function objects that belong to a specific section object. The user can also print the entire executable's contents.

Class Section requires the name of its section and the contents of its section. Given the contents of the section, functions will be parsed and sent to class function. The list of function objects is stored in a list within Class Section and can be called by Class Executable to return the list of function objects that belong to that section. The name and contents of the section can be printed.

Class Function requires the name and contents of the function being created. A list of dictionaries for each line is automatically generated when the object is created. This list of dictionaries allows for specific references to an assembly line's "memory address", "hex", "instruction", "registers" and "comment". One could check if the assembly instruction "call" is being used by using the key "instruction" to assess whether this particular line contains that instruction.

Other Function Responsibilities

The "main" function will allow for varying analysis. If just function calls (not assembly instruction calls) need to be displayed, that can be altered here. Creating executable class objects is also done here.

The "Graph" function uses the Turtle module for a graphical representation of function and assembly instruction calls.

Differences

I will be referring to UML diagrams "High-Level Initial Design" and "High-Level Final Design" to discuss the differences.

I hadn't identified the importance aspects of an assembly file – like sections – or the need for a vulnerability analyst to specify a section or function directly. For instance, the. plt section may not be important to analyze so it shouldn't be included as to reduce overhead and visibility of results. To solve this problem, I created a "section" and "function" class. The analysis can now be done with specific characteristics of a binary removed.

Next, I removed the "DoXWithFunction" class. There's really no need for this class because the goal of this assignment is to help a vulnerability analyst find vulnerabilities faster. Instead of doing something with the function, it's now automatically parsed; This was probably the only useful thing a class like that would have done anyways.

I also got rid of the "DoXWithCall" class. This is because I don't have enough time this semester to identify "blocks" of code myself. This would require looking into the patterns in Dr. Jia Song's research into assembly language. While I was hopeful I could fulfil that initial goal, I didn't have enough time to execute.

Lastly, I am now using the Turtle module to create a graphical representation of the data. This will make the analysis much cleaning and easier to identify potentially vulnerable areas of code.

Resume

Python Project (30 hours)

- Use classes to parse binary into assembly file objects with classes for each section and corresponding functions.
- Identify function calls in assembly to direct vulnerability analyst to user input.
- Identify assembly calls to identify blocks pertaining to high-level source code.

References

Andriesse, D. (2019). *Practical binary analysis: Build your own linux tools for binary instrumentation, analysis, and disassembly*. No Starch Press.

Kerrisk, M. (2021, February 6). *objdump(1) — Linux manual page*. man7. Retrieved April 29, 2022, from https://man7.org/linux/man-pages/man1/objdump.1.html

Miller, B. N., Ranum, D. L., & Anderson, J. (2021). *Python programming in context* (3rd ed.). Jones & Bartlett Learning.