**UNIVERSITY OF PUERTO RICO**

**RECINTO UNIVERSITARIO DE MAYAGÜEZ**

**MAYAGÜEZ, PUERTO RICO**

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING**

**TITANIUM-CODE: Programming Language for Secure Coding**

**INEL 4036 - 096 : Programming Languages**

**Prof. Wilson Rivera**

Kensy Bernardeau

Dong Wang (Robbie)

Luis Diaz

**1. Introduction**

Secure coding standards provide rules and recommendations to guide the development of secure software systems [1]. These standards ensure that the system will be secure in the developing phase; it also provides a common set of criteria that can be used to measure and evaluate software development efforts and software development process [2]. In the project of Titanium, we design the code to detect the insecure code in a file of C programming language. Since C programming is widely applied in library of Caffe (the programming library for Graphic Processing Unit), our language attaches importance to the protection of software and hardware. By using this language, we use PLY python to design a functional system to implement security checkpoints that would detect code that could lead to a security flaw when the program is being complied.

The motivation for the development of this language is to apply the secure coding standards towards the programmers. Coding standards that most programmers use are incomplete and not security centric. With this language, we can have it running security checks during compilation time such that when it detects any security flaw, the source code ceases to compile and an error is thrown.

Titanium-Code, the programming language under development, looks to be a tool that regulates and facilitates the prevention of these security flaws. Secure Coding should be something that every programmer should be aware of, thus there should be general standards set to ensure a safe code. For this purpose, this programming language strives to do just that.

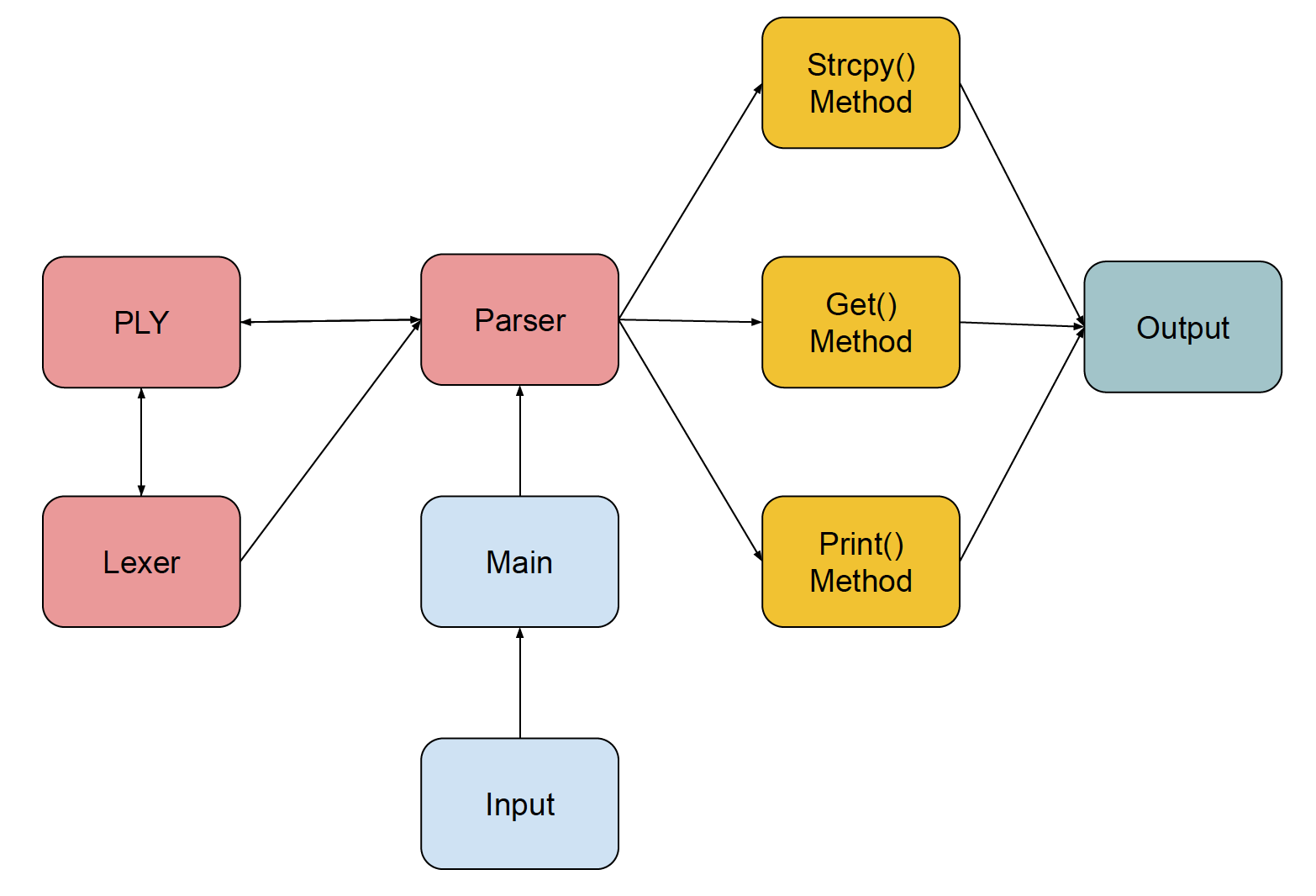
In this report, I will describe, the flowchart of design, language features, the structure of the programming language, development environment, test methodology. At the end, I will conclude this functional language and potential of this project.

**2. Language Development**

**Module Interface**

The main module of Titanium, titanium*.py* is the module that takes care of capturing the user input. The module passes the user input to the parser, *titanium\_parser.py*, which processes the input with the lexer (*titanium\_lex.py)*, and the PLY package. After processing the input, the parser then stores the appropriate user input. After we have lexer and parser, we wrote the main program to allow the file (endswith C) and import the parser. Then in the implementation.py, we build three functionalities. They are the checkpoint method for strcpy(), get(), printf() method in C program. When the insecure points are figured out, we have output which shows the number of line where the insecure coding is.

**Flowchart of Design**

****

**Development Environment**

The following tools were used to develop PLP\_Titanium:

• Python 3.4 : Language utilized to develop the project

• PLY: Package lex and yacc parsing tools

• PyCharm: IDE utilized to develop the project

• Github: Platform hosting the source code.

**3. Language Tutorial and Its Features**

To run Titanium language, the user need Python 3.4 and the PLY package installed, then download the PLP\_Titanium zip file from github and extract its content. Finally, the user needs to execute PLP\_Titanium/titanium.py in order to run the language.

This language will serve as a checkpoint before compilation: it will “read” the code and will prevent the code from being compiled if it detects any code fragment that could indicate an eventual security flaw in the system. It should detect vulnerabilities, such as buffer overflow, or a format String Exploit, among others. It should then cancel compilation and warn the user that there is a vulnerability in a specific fragment their code and would indicate how to best deal with the vulnerability. Therefore, the resulting code can have a secure structure, and the user (the programmer) can better understand and begin to code more carefully, avoiding the security risks that could arise.

Such features to be proposed include:

* Compiled
* Functional Paradigm
* Security Check during Compilation

**4. Example of a program**

An implementable security vulnerability check is to validate if a given variable is within the acceptable bounds of the type to avoid a buffer overflow. This issue could be caused when a variable with an unacceptably large value overwrites adjacent memory locations. The code can check the value of a variable and prevent the user from going forward with it if its value exceeds the bounds established.

The language should allow programmers to choose the targeting vulnerabilities the code looks for from a range of options. Since the main objective of program is to read code and look for security risks, the user should also give the code to be checked as input for the program. The language of the user’s input is also important; while Python is priority, it could be extended to various other programming languages with different syntax, which would cause different strategies in detecting security flaws.

As it was mentioned above, some simple vulnerability checks like the buffer overflow occur if a variable surpasses a specific bound or maximum value. For situations and security flaws like this, the user can also specify the range of values that is acceptable for a certain variable. The program would then warn the user and prevent compilation, then informing the programmer that there is a risk in the code that was written (in a specific case, Buffer Overflow).

The syntax of Titanium-Code, while it is not final, could follow this general structure:

//Create a CodeMonitor that has appropriate functions

CodeMon = new CodeMonitor()

//Specify Language of the input code and include the input file that will be evaluated. It could also be a .txt file or a String

CodeMon.AddCode(file = “/Downloads/Pong99.py”)

CodeMon.Lang(language = “Python”)

//Add in the security risks that the program should watch out for, and set any parameters that could be needed. For example, the lowbound and highbound.

CodeMon.BufferOverflow(lowbound = 0, highbound = 300)

CodeMon.SensitiveInfo(password = “123456”)

CodeMon.CodePointerExploit()

**We can write a code that calls a command that does one or more of the following security coding techniques:**

1. **Validate input from all external data sources including user controlled files, network interfaces, and command line arguments.**
2. **Have a layered defence with multiple strategies. If one does not prevent a vulnerability, the next layer of defence should.**

**3. Test Methodology**

**4. Conclusion**

The result of this work was a functional language that can detect insecure coding in its early stages. The two proposed main features were successfully implemented and are fully operational. PLP\_Titanium’s grammar was designed to be as close as possible to English sentences. This resulted in an intuitive grammar for users who had no prior experience in C programming. The biggest challenge faced during this project was implementing the parse.

PLP\_Titanium has potential to make other aspects of checkpoints. Some of the future features that the team is interested in developing for PLP\_Titanium are: multiple file detection, manipulation for insecure C programming. In conclusion, PLP\_Titanium has potential to become a powerful tool in the field of software development.

**References:**

[1] F. Sheldon, A. Krings, S. Yoo, and A. M. Editors, “Third Annual Cyber Security and Information Infrastructure Research Workshop COMPREHENSIVE STRATEGIES THAT MEET THE CHALLENGES OF THE 21ST CENTURY,” 2007.

[2] R. W. Yeung, “Secure Network Coding,” p. 7803, 2002.