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

2. Language Development

2.1 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 user input in data structure of tuple and list. 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 would have output, which shows the number of line where the insecure coding is.

2.2 Flowchart of Design

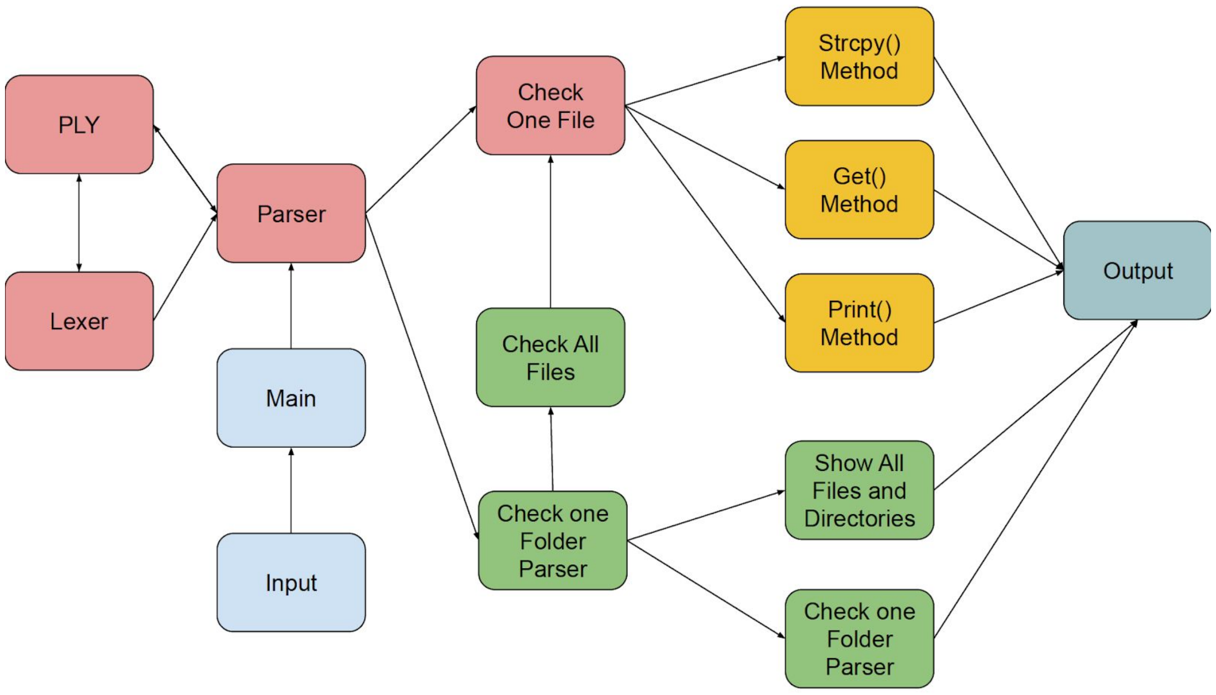

Figure 1 the Flowchart of Titanium Project

2.3 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

3.1 Language Basics

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.

On the next stage, we developed a systematic way to find all targeted c files and their insecure coding. This method should proceed before the compilation, then suggest the users whether their folder has risk potential.

3.2 Language Features

Such features to be proposed include:

- Compiled
- Functional Paradigm
- Security Check during Compilation
- Folder Check for Insecure Coding

3.3 Reference Manual

- Display help:

help

- Process a given file:

<variable name> = process("/path/to/file")

- Open a given file:

<variable name>.open()

- Show a file's content on the terminal:

<variable name>.show()

- Evaluate the vulnerabilities of the file:

<variable name>.evaluate()

- Process a given directory:

<variable name> = process("path/to/directory/")

- If we process a directory, the program will show a folder content, indicating the next level of directory and the address of .c files on the terminal:

<variable name>.show()

- If we process a directory, the system will check all ".c" files and their vulnerabilities.

<variable name>.evaluate()

- If we process a directory, the program will show all ".c" files under evaluation in this folder.

<variable name>.sAll()

- If we process a directory, the system will check all ".c" files and show their vulnerabilities.

<variable name>.eAll()

4. Description of Implementation

4.1 Implementation Fundamentals

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:

4.2 Test Methodology

The test methodology done for this project is Black Box Testing. Black Box Testing is a testing procedure where we test the functionality of the created software. In this case, we tested whether Titanium can be able to read and detect vulnerable source code files with a functional programming language translator interface whilst the tester is unaware of the inner workings of the language. As shown in the above example, it's a resounding success in detecting the potential vulnerabilities it has knowledge of, prints out a detailed analysis on where the vulnerabilities are found and outputs suggestions on how to handle such vulnerabilities. Then the tester can open and edit the files directly from the language, save the changes, and can run the evaluation function to run another analysis.

Any syntax and grammar error are handled with an output, but the session of the language is kept alive. Writing "help" on the language will display the available operations the language has.

4.3 Code Implementation

Example 1: Check the First Source Code

```
IN ->f = process("test.c")
Code File uploaded
File processed!
IN ->f.evaluate()
RUNNING: gets() check
WARNING
at line:  6
index(start- end) 00-14: gets(username)

WARNING
at line:  12
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets instead of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
No vulnerable printf functions has been found.

RUNNING: strcpy check
No strcpy functions found.

Total potential vulnerabilities detected: 2
```

Example 2: Check the Second Source Code

```
IN ->g = process("test1.c")
Code File uploaded
File processed!
IN ->g.evaluate()
RUNNING: gets() check
WARNING
at line:  12
index(start- end) 00-14: gets(username)

WARNING
at line:  19
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets instead of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
WARNING
at line:  24
index(start- end) 00-15: printf(*secret)
SUGGESTION:
Make sure printf is not taking in any user input.
In the case of user input use, make sure the input has been sanitized and the compiler has ASLR enabled.

RUNNING: strcpy check
WARNING
at line:  6
index(start- end) 00-17: strcpy(str1,str2)
SUGGESTION:
Use the str-n-func family of functions. The additional n is a parameter for maximum number of characters.

Total potential vulnerabilities detected: 4
IN ->f.open() //Opens default text editor of an Operating System
IN ->g.open()
```

Example 3: Show all the files in one directory

```
IN ->f = process("Layers/")
Code File uploaded
You are about to evaluate the folder instead of one file.
IN ->f.show()
You are about to display all the files and directories in this folder.
Want to proceed? Y/N Y
The entries are
These are the all the files and directories.
Layers/layer1/  <-- Directory
Layers/abc.txt
Layers/cba.md
Layers/test1.c  <-- Target File
Layers/test.c  <-- Target File
IN ->
```

Example 4: Evaluate all the files in one directory (only process one level)

```
IN ->f = process("Layers/")
Code File uploaded
You are about to evaluate the folder instead of one file.
```

```
IN ->f.evaluate()
You are about to evaluate the folder instead of one file.
Want to proceed? Y/NY
These are the files under evaluation.
['test1.c', 'test.c']
Layers/test1.c
RUNNING: gets() check
WARNING
at line:  12
index(start- end) 00-14: gets(username)

WARNING
at line:  19
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
WARNING
at line:  24
index(start- end) 00-15: printf(*secret)
SUGGESTION:
Make sure printf is not taking in any user input.
In the case of user input use, make sure the input has been sanitized and the compiler has ASLR enabled.

RUNNING: strcpy check
WARNING
at line:  6
index(start- end) 00-17: strcpy(str1,str2)
SUGGESTION:
Use the str-n-func family of functions. The additional n is a parameter for maximum number of characters.

Total potential vulnerabilities detected: 4
Layers/test.c
RUNNING: gets() check
WARNING
at line:  6
index(start- end) 00-14: gets(username)

WARNING
at line:  12
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
No vulnerable printf functions has been found.

RUNNING: strcpy check
No strcpy functions found.

Total potential vulnerabilities detected: 2
IN ->
```

Example 5: Show All Files in one directory (All Levels)

```
IN -> f = process("Layers/")
Code File uploaded
You are about to evaluate the folder instead of one file.
IN ->f.sAll()
showAll
['Layers/layer1/layer2/test3.c', 'Layers/layer1/test2.c', 'Layers/test1.c', 'Layers/test.c']
IN ->
```

Example 6: Evaluate All Files in One Directory (All Level)

```
IN ->f = process("Layers/")
Code File uploaded
You are about to evaluate the folder instead of one file.
IN ->f.eAll()
evaluateAll

********************* The file name and location:   Layers/layer1/layer2/test3.c
*********************************

RUNNING: gets() check
WARNING
at line:  12
index(start- end) 00-14: gets(username)

WARNING
at line:  19
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
WARNING
at line:  24
index(start- end) 00-15: printf(*secret)
SUGGESTION:
Make sure printf is not taking in any user input.
In the case of user input use, make sure the input has been sanitized and the compiler has ASLR enabled.

RUNNING: strcpy check
WARNING
at line:  6
index(start- end) 00-17: strcpy(str1,str2)
SUGGESTION:
Use the str-n-func family of functions. The additional n is a parameter for maximum number of characters.

Total potential vulnerabilities detected: 4

********************* The file name and location:   Layers/layer1/test2.c
*********************************

RUNNING: gets() check
WARNING
at line:  12
index(start- end) 00-14: gets(username)

WARNING
at line:  19
```

```
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
WARNING
at line:  24
index(start- end) 00-15: printf(*secret)
SUGGESTION:
Make sure printf is not taking in any user input.
In the case of user input use, make sure the input has been sanitized and the compiler has ASLR enabled.

RUNNING: strcpy check
WARNING
at line:  6
index(start- end) 00-17: strcpy(str1,str2)
SUGGESTION:
Use the str-n-func family of functions. The additional n is a parameter for maximum number of characters.

Total potential vulnerabilities detected: 4

********************* The file name and location:   Layers/test1.c
********************************

RUNNING: gets() check
WARNING
at line:  12
index(start- end) 00-14: gets(username)

WARNING
at line:  19
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
WARNING
at line:  24
index(start- end) 00-15: printf(*secret)
SUGGESTION:
Make sure printf is not taking in any user input.
In the case of user input use, make sure the input has been sanitized and the compiler has ASLR enabled.

RUNNING: strcpy check
WARNING
at line:  6
index(start- end) 00-17: strcpy(str1,str2)
SUGGESTION:
Use the str-n-func family of functions. The additional n is a parameter for maximum number of characters.

Total potential vulnerabilities detected: 4

********************* The file name and location:   Layers/test.c
********************************

RUNNING: gets() check
```

```
WARNING
at line:  6
index(start- end) 00-14: gets(username)

WARNING
at line:  12
index(start- end) 00-14: gets(username)

SUGGESTION
Use fgets insetad of gets, since the use of gets has been deprecated due to overflow vulnerabilities.

RUNNING: printf check
No vulnerable printf functions has been found.

RUNNING: strcpy check
No strcpy functions found.

Total potential vulnerabilities detected: 2
IN ->
```

5. Conclusion

From the result of implementation, we can observe that our programming language is fully functional one that can detect insecure coding in the early stage. The proposed main features were successfully implemented and to detect the vulnerabilities of a file and a folder. Besides, we develop the method display all the targeting files. In the view of syntax, the Titanium's grammar was designed to be close to English sentences. This would help a novice user in C programming to grasp our language by intuitive. The biggest challenge faced during this project was implementing the parser and testing the functionalities of the language altogether.

Titanium has potential to make other aspects of checkpoints. Some of the future features that the team is interested in developing for Titanium are: manipulation for insecure C programming, database of popular vulnerabilities, insecure checking for other programming languages. At last, Titanium has potential to become a powerful tool in the field of software development and software security analysis.

References:

[1] F. Sheldon, A. Krings, S. Yoo, and A. M. Editors, "Third Annual Cyber Security and Information Infrastructure Research Workshop," 2007.

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