Static Analysis Tool

Report

Software Security

MEIC-A

Group 3

2021/2022

92513 Mafalda Ferreira

92546 Rita Oliveira

1. Context

Web application vulnerabilities, like SQL injection, Cross-site scripting (XSS) and Cross-site Request Forgery (CSRF), are common. These vulnerabilities originate in programs that encode insecure and dangerous information flow and that have the user input (source), a low-integrity information interferes with high-integrity parameters of a function or variable (sensitive sink). A malicious user sees these vulnerabilities as an opportunity to alter the behavior of sensitive sink or to induce the program to perform security violations. In average projects, it can be difficult for developers to find vulnerabilities. Because of that, in this project it was developed a static analysis tool which helps developers to identify data and information flow violations that are not protected in web applications. It was focused on web server side programs encoded in Python like web framework Django.

2. Approach

2.1. Parsing and Implementation

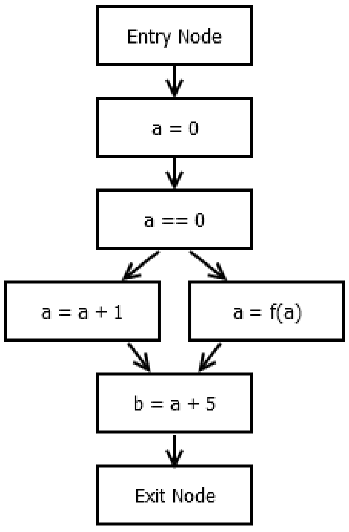
The programming language used to develop the static analysis tool was Python, which allowed to use auxiliary libraries in order to parse the Abstract Syntax Tree (AST), provided as input, into a Control Flow Graph (CFG). Each AST node is recursively processed and saved into different derivative classes of Node, according to their type. One node can contain multiple nodes within its attributes, defined when analyzing a single line of code.

2.2. Control Flow Graph

A CFG is composed of three different types of blocks. The first one is an Entry Node, the last one is an Exit node and the remaining blocks that correspond to a unique line of the code, representing a statement node. Whenever a processed block is next to a previous processed one, the second block is added to the first's children list and therefore, the first block is added to the second's parents list. This will result in continuous graph blocks with different flow paths, where a block with more than one child represents more than one different flow, which is the case for If, While and For Nodes, known as Control Flow Nodes. These nodes are represented by their test node which contains two children, the first one reserved for the first body branch block and the second one for the first orelse branch block. A potential block that is following a Control Flow Node is added to the last child of each branch, resulting in diverging flows that will carry out different program states such as assigned variables, sanitizers and tainted nodes.

The following code is represented by the following CFG and represents two different flows originated from from the test node of If Node.

```
a = 0
if a == 0:
  a = a + 1
else:
  a = f(a)
b = a + 5
```



Since the static analysis tool is developed under a CFG, it is able to detect potentially dangerous information flows specific to a given path that contains tainted and, consequently, low integrity information. This information may interfer with high integrity information and, ultimately, reach a sensitive sink, resulting in a vulnerability. This analysis is achieved by defining a Policy for each pattern.

2.3. Policy

A Policy contains a list with the pattern's security levels. These levels correspond to a list of multiple tuples with different combinations of possible entry points of vulnerabilities. Initially, every existing node contains an empty label meaning there is currently no dangerous flow.

2.4. Analysis of vulnerabilities

During the program analysis, whenever a vulnerable source is encountered by matching the id of a Name Node that represents the name of variables, functions or attributes, the node is assigned a label containing the tuple with the entry point name. This label is propagated through the nodes within a single block, until it reaches the most extrinsic node, and then propagated to the next block. This is achieved using the available policy rules specific to each node type. These rules can correspond to the least upper bound, greatest lower bound, top or bottom between two labels. The analysis will result in different labels propagated during the program flow which will allow nodes to be tainted by multiple sources.

Besides detecting explicit flows, the tool is also able to detect implicit flows which originate from the test node of Control Flow Nodes. Whenever the test node gets tainted by an existing access to a low integrity value, it is assigned a new label that is propagated to every node of both body and orelse branches, originating an explicit flow.

If a data flow passes through the sanitizer and arrives at the sink, the tool will report the vulnerability and the function used to sanitize. During the flow analysis, the program collects all sanitizes that exist in a certain flow and checks if results of sanitization arrive at the sink.

3. Execution of the Tool

To use this tool, it will be needed to run in the command line the following command: **python ./bo-analyser.py program.json patterns.json**. Inputs are a file JSON containing the program slice to analyse, represented in the form of an AST (program.json) and a file JSON containing the list of vulnerability patterns to consider (patterns.json). From the file patterns.json, the tool will report a possible vulnerability if a data flow from a source to sink even if the flow passes through the sanitizer. Output will be the report encode in a file JSON with name **program.output.**

4. Evalutation of the Tool

The testing of the tool was done using the public official tests in zip file *20-Jan.zip* provided in the fenix page. The correctness and robustness of the tool was tested by comparing the tool's results with the output available in the public tests, taking in consideration that the order of the vulnerabilities were not important. Afterwards, the tool was adjusted, in order to achieve the best results.

5. Discussion

5.1. Strengths

There are dangerous flows that can only reveal themselves after an unknown number of iterations in While and For Nodes. The implementation of a **fixed point** for **flow detection** fortifies the solution for this type of situations. During the analysis, the program considers one more iteration of the most inner loop everytime a new unregistered flow is encountered. This way, the program will go through every possible existing flow and won't leave any flow undetected in Loop Nodes.

5.2. Limitations: Imprecise Tracking of Information Flows

As well as the previous strengths, there wasn’t found any situation were the tool would lead to undetected flows. Therefore, the tool doesn’t produce any **false negative**.

On the other hand, some flows can be unduly reported during the static analysis. This can happen in situations where both Control Flow Node’s body and orelse branches are composed by the same nodes and, therefore, every branch executes the same statements, independently of the veracity of the test node. Since the program doesn’t verify the similarly between these branches, it will originate **false positives** in these situations. In the future, this could be solved simply by defining a function that goes through all nodes of each branch at the moment of the creating of the Control Flow Nodes. This function could verify if both branches had the same nodes and add a flag that marks the node as having two equal branches. This flag would be useful at the moment of the decision if a implicit flows were to be considered.

Adding to this, the tool is also limited in terms of the Break Node. In this implementation, it considers that an existing break is a regular node that doesn’t alter the direction of the flow, resulting in **false positives**. The following code snippet represents a case where, after the program reaches *break*, it continues one more loop iteration, leading the flow to the *sink*, which will then originate the detection of a false vulnerability. In a well-functioning tool, there should not be a flow between the *source* and the *sink*.

```
a = f()
while a < 10:
     if a == 5:
         a = source(a)
         break
    elif a == 20:
         a = a + 1
         sink(a)
print(“hello world”)
```

One solution for the Break Node would be to fix the flow and, during the construction of the graph, link the break node to the node following the most inner Control Flow Node.

5.3. Limitations: Imprecise endorsement of input sanitization

It was not found any sanitization functions that could be ill-used and not properly sanitize the input, in other words, it was not found false negatives, because was guaranteed that tool only capture data sanitized that arrived at the sink

There are situations where sanitization procedures are not detected by the tool. If a data sanitized is involved in binary operations that is argument of a function , the tool will not report the sanitization. For example: (2-expr-binary-ops)

```
a=b()
c=s("ola",a)
f=e(c+"oi"+d+"hi",a)
```

If **b** is source and **s** is sanitizer so variable c will be the result of sanitization. If **e** is sink then the tool will not analyse correctly the expression **c + "oi" + d + "hi"** and therefore will not detect the flow sanitized.
One way of making the tool more precise is to use the algorithms of tool Pythia (Searching for Dangerous Flows and Exploring Paths).

6. Related work

In three papers given in the project, it is verified that there exist other tools that address the same problem: PyT (*S. Micheelsen and B. Thalmann, "PyT - A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications", Master's Thesis, Aalborg University 2016*), Seldon (*V. Chibotaru et. al, "Scalable Taint Specification Inference with Big Code", PLDI 2019*) and Pythia (*L. Giannopoulos et. al, "Pythia: Identifying Dangerous Data-flows in Django-based Applications", EuroSec 2019*). These tools also targeted Python.
Like the tool developed in project, PyT also use AST and CFGs to help analyse all flows that existed in program coded in python
The other two papers that were used to support the project were: https://cs.au.dk/~amoeller/spa/ (that use CFGs and Lattice) and https://www.researchgate.net/publication/2911969_Control_Flow_Graphs_for_Real-Time_System_Analysis_Reconstruction_from_Binary_Executables_and_Usage_in_ILP-Based_Path_Analysis (that use CFGs and it was also useful to analyze the loop flows)

7. Conclusion

In this project, was achieved the goals of analyse explicit flows in the absence of branching instructions and in the presence of branching instruction. Also, was achieved the goals of analyse implicit flows and analyse potential sanitization of vulnerabilities. This tool will help developers to find vulnerabilities in their code. To improve the tool, it would be necessary to reduce false positives.

8. References

[1] Micheelsen S., Thalmann B. (May 31, 2016). PyT - A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications
[2] Chibotaru, V., Raychev, V., Bichsel, B. Vechev, M., (Jun, 2019). Scalable Taint Specification Inference with Big Code
[3] Giannopoulos, L., Tsanakas, P., Degkleri, E., Mitropoulos, D. (March, 2019). Pythia: Identifying Dangerous Data-flows in Django-based Applications
[4] Møller, A., Schwartzbach, M. (November 27, 2021). Static Program Analysis
[5] Theiling, H. (2002). Control Flow Graphs for Real-Time System Analysis
[6] Harrold, M., Tech, G., Rothermel, G., Orso, A., (n.d.). Representation and Analysis of Software