TÉCNICO LISBOA

Project Report

SOFTWARE SECURITY

GROUP 36
FRANCISCA ALMEIDA 93709
GONÇALO ANTUNES 93716
RODRIGO PEDRO 93753

4 FEBRUARY, 2022

Contextualization

The goal is to develop a tool for static analysis of Python code, that when given an input in the form of a code slice, analyses within a set of vulnerabilities if there is a source whose flow leads to a sink. Python is a language widely used in Web Frameworks (and not only), so it is important that these programs are free of vulnerabilities, therefore the development of said tool is necessary.

Approach

There are two concepts that are important to mention that make up the technique used by the tool. What we refer to as "context" corresponds to a dictionary in which the key is a declared variable and the value is an array of sources through which that variable was tainted. This implies that when a variable is declared but not tainted the array will be empty. This property is relevant for intercepting variables that constitute a source because they were never declared prior to their use (this type of variables will from now on be referred to as Rogues). These contexts are passed recursively over each node of the AST. The assignment node is the only node that updates/manipulates the contexts. In this node the variables that were not tainted until then can be tainted by the assignment. The other nodes manipulate what we call labels. A label is a structure (VarTaint) which consists of the name of the variable, if it is sanitized and by whom and whether it is Rogue or not.

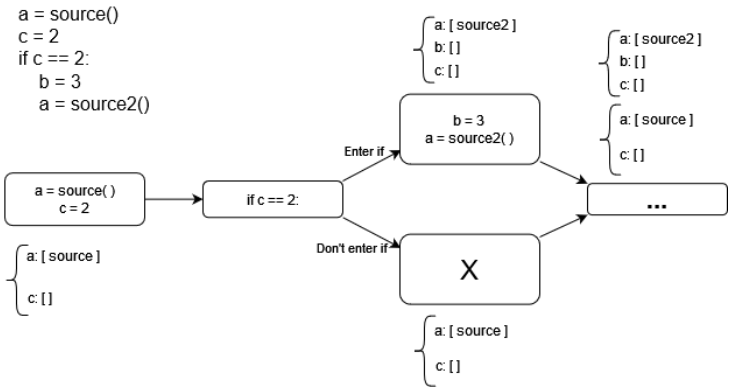

Figure 1: Example of context flow

The purpose of labels, which is an array of VarTaint, is to mark each node of the AST with the sources that affect it. As you traverse the AST, each node manipulates the labels differently:

- Name - The name node is given a context and will then analyze and identify the VarTaint(s) that exist in that context. Returns a set of labels.
- Compare & BinOp - These nodes receive the labels from the nodes below (left and right) and return a union of these labels (after removing the repeating ones).
- Call - Returns the labels resulting from applying the function to the arguments. If it represents a sanitizer, it returns updated VarTaints of the args with the sanitizer. If it represents a sink, it checks for VarTaints and registers vulnerabilities.
- Assign - Returns an updated context with the result of the assignment. Checks if the target is a sink and, if so, registers vulnerabilities.
- If - Receive a context and return a list of contexts. This list contains contexts corresponding to the case where the if is performed and to the case where it isn't (includes else if exists).

- While - Resembles the behavior of the if node, but the list contains contexts corresponding to not executing the while and executing the while 1, 2, 3, ... times.

In the case where the implicit flag is on, both the If node and While node manipulate a Stack. Whenever the program enters an if or while node, an entry is added to the stack that contains labels resulting from evaluating the conditions. This entry allows the assignment below in the AST to add these VarTaints to the context.

Behavior

The program receives as input a json that contains a python slide in AST format and a json with the vulnerability patterns. The tool looks for vulnerabilities in the slice and outputs them in a json file.

To run the tool, use the following command:

```
$ python parser.py ./slice.json ./pattern.json
```

Evaluation

To evaluate the correctness and robustness of the tool, the detection of vulnerabilities was measured by running tests on Python slices. The tests consist of detecting explicit and implicit flows between sources and sinks, some with sanitizers. The tool's output was compared with the expected output, taking into account the detection rate of vulnerabilities and the percentage of false positives/negatives.

Analysis

Strengths

The developed tool excels at detecting explicit flows with static code analysis. It takes into consideration assignments, function calls that use tainted arguments, binary expressions, etc. For every source-sink flow, the program tracks all sanitizers in order of appearance. Undeclared variables are treated as sources. For every if condition, both possible scenarios are tracked - the condition evaluating to true or false - and therefore two flows are generated. As for the while instruction, the tool tracks flows that span over multiple iterations of the while loop. A conservative approach is adopted when it comes to classes and their attributes: a tainted attribute marks the whole class as tainted.

Weaknesses: Imprecise tracking of information flows

```
def function():
    return source()

a = function()
sink(a)
```

Figure 2: False negative case

Functions declared outside the slices that may possibly interact with sources aren't checked for taintedness, thus resulting in false negatives.

In the example, our tool won't detect that *a* is tainted by a source because it doesn't inspect what happens inside *function().*

To explore this weakness, an attacker could inject malicious code inside a function declaration, and call that function, and the tool would not detect a vulnerability. To avoid said false negatives, the tool could be expanded to analyze not only the function body code, but also the return value (if it exists).

```
a = 2
while ….:
   a = source()
else:
   a = 3
sink(a)
```

Figure 3: False positive case

The tool assumes the existence of breaks in every while instruction, meaning that it tests all possible flows. In this example, the program considers the possibility of executing the while and not the else, which is not possible, and therefore triggering a vulnerability detection.

In order to avoid this situation, the tool must not test flows which realistically cannot happen, meaning that if a break doesn't exist in a while loop, the tool shouldn't assume there is a flow where a break exists.

Weaknesses: Imprecise endorsement of input sanitization

```
a.a = source
a.b = sanitize(a.a)
sink(a)
```

Figure 4: False negative case

The tool doesn't take into consideration attributes of a class. Instead, it considers the class as a whole. In this example, when a.b is sanitized, the whole object a is sanitized, while in reality attribute a.a is still tainted, so A should still be tainted in our approach.

To prevent this, when sanitizing an attribute, only that attribute should be sanitized and not the whole object. An attacker can taint an attribute of an object and then proceed to sanitize another attribute and thus mark the object as sanitized.

```
d = source
a.a = sanitize(d)
a.b = source
sink(a.a)
```

Figure 5: False positive case

Much like the previous example, assigning a source to a.b marks "a" as tainted. However, the sink takes as input attribute a.a, which should be sanitized. When sanitizing an attribute, only that attribute should be sanitized and not the whole object.

```
a = source
if False:
    a = sanitize(a)
sink(a)
```

Figure 6: False positive case

In this example, the tool will detect a sanitized flow, when in fact that flow is impossible if the condition is evaluated. Like mentioned before, cases like this can be avoided by dynamically evaluating conditions.

Some limitations were mentioned earlier and so were ways around them. The solutions proposed above would have a very positive impact on the accuracy of the tool since they reduce the cases of false positives and negatives. However, it would require additional complexity which in turn would have more negative impacts on the efficiency of the tool.

Related work

A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications [ref]

The PyT program presented in the paper, much like our tool, parses Python ASTs in order to detect vulnerabilities via static analysis. The paper emphasizes the representation of ASTs as control flow graphs (CFGs) as a means to track source-sink flows. While our tool doesn't explicitly build CFGs, it emulates the behavior of tracking all possible flows of the program slice.

Scalable Taint Specification Inference with Big Code [ref]

The paper presents Seldon, a static analysis method that transforms the taintedness detection problem into a linear optimization problem. Given a small set of annotated API calls tagged as source, sink or sanitizer, the program infers the annotations for the larger set as a means of detecting vulnerabilities in Python code. While the foundation of the problem is the same, our tool does not implement machine learning methods for detection, relying instead on simple flow algorithms.

Pythia: Identifying Dangerous Data-flows in Django-based Applications [ref]

The Pythia tool explores known web vulnerabilities such as SQL Injection and XSS in the Django framework (a python-based framework used for web development). The origin of said vulnerabilities starts with the usage of packages and dependencies that disable built-in security features from the framework. This tool resembles the program we've developed because it looks for patterns (known vulnerabilities) and marks disabled security dependencies as tainted, much like in any other data-flow analysis, and later analyzes these dependencies for vulnerabilities.

Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology [ref]

The paper presents a way of tracking taintedness in Python code by modifying the interpreter, much like Perl's taint mode. This work extends to our own tool since flow tracking is closely related to taintedness.

A Taint Mode for Python via a Library [ref]

Unlike the previous paper, this static analysis tool is implemented as a library, thus not modifying the Python interpreter. It also mentions sources, sinks and sanitizers and vulnerable flows between these specifications.

Conclusion

Taking into consideration the requirements of the project, we developed a tool for static analysis. The tool can detect basic vulnerability flows in Python code slices. Given the time frame to develop this tool, there were some instructions of Python Language that were not taken into consideration. We believe there is potential for improvement and a good starting place would be to analyze said instructions.