Project 2: Discovering vulnerabilities in JavaScript web applications

Segurança em Software, Group 32, Report

98678, Bruno Freitas 98624, João Vieira 98664, Diogo Nogueira

1 Introduction

JavaScript is the most widely used web programming language among professional and beginner programmers. In order to develop secure code and avoid exploits, developers require great knowledge on how to surpass and sanitize code vulnerabilities and must be careful in the way information flows through their code. Although many people don't have this knowledge or just have a complex application whose flows are difficult to follow, there are vulnerability scanning tools available to use. This report presents a tool that can help in identifying dangerous information flows, which might expose users' private data to other malicious users. Everyday, thousands of websites are targeted by these attacks, due to improper use of JavaScript, therefore, we present a possible solution to make developers aware of their mistakes and protect not only their own service, but also their users.

2 Explanation

After reading S. Guarnieri et. al.'s [3] article, we decided to approach this problem in a similar way and build a very simplified version of what they did. So, in short, our solution builds a diagram that represents the multiple ways information may flow through the provided program. Then, it recursively computes all possible paths through that graph with a maximum depth of K (after some testing we found that K=10 was a good value for our implementation). Finally, our tool goes through the computed paths and discovers possible source-to-sink chains and if there are any sanitizers in between.

3 Behaviour

```
1  a="";
2  b=c();
3  d(a);
4  e(b);
```

Figure 1: 1a-basic-flow.js

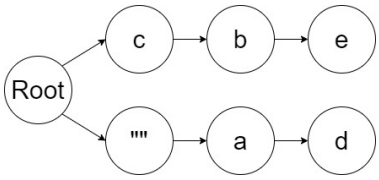

Figure 2: The graph produced by 1a-basic-flow.js

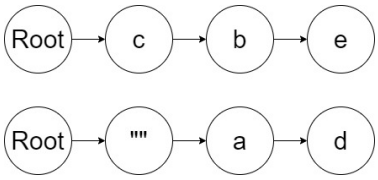

Figure 3: The paths found in the graph from Figure 1

Using one of the provided code snippets (Figure 1) as an example, our solution creates the graph in Figure 2 and then finds the paths in Figure 3. Afterwards, it looks for source-sink pairs in each path and checks their index in the path to make sure the source comes before the sink. Finally, it checks for sanitizers between them.

The connections in our graph represents an "affects" relation, meaning the parent node has an effect on the information contained within its child nodes. For instance, the right part of an assignment has an effect on the left part.

4 Tool Evaluation

We started by testing our code using all the provided code snippets and building patterns that would make sense considering each snippet. When we got a false negative, we studied the case and changed the code accordingly, in order to make it detect the vulnerability.

5 Critical analysis

5.1 Imprecise tracking of information flows

Throughout our tool evaluation, we didn't find any false negative or in other words, all illegal information flows were captured by our technique. This could be because a flow graph is built and we search for all possible flows in that graph. However, this was not true for the false positives, our tool reports some nonexistent flows. One example of this problem is with the *if* statement, because the *then* block is not separated from the *else* block. What can happen is if we had a code like Figure 4, our algorithm would build a possible path like the Figure 5. This flow path suggests that the *then* block and the *else* block can both run in a single execution of the program, which is obviously not true.

```
1    if (d < e) {
2        a = b
3    } else {
4        c = a
5    }
```

Figure 4: JS program example that produces an unduly flow

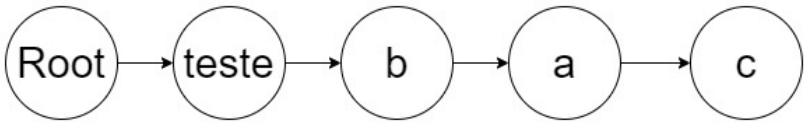

Figure 5: A path contained within the graph

5.2 Imprecise endorsement of input sanitization

During the testing analysis of our solution, we faced some code slices which could include flows that passed through sanitization functions. When a sanitizer was included in the "patterns.json" file, the tool could successfully detect which identifiers were affected. However, when a program slice has a variable which is overwritten, we do not take this in consideration and therefore, an impossible flow is detected as well. This imprecision leads to the presence of false positive cases. On the other hand, during our evaluation we weren't able to discover any false negatives. As we stated before, this could be due to the flow graph being built in conservative way, meaning we would rather get more false positives than false negatives while we search for all possible flows in that graph.

To prevent this, when an assignment to a variable is overwritten the flow should not be taken in consideration to the final evaluation of the vulnerability.

```
1     b = a
2     b = c(a)
3     e = b
```

Figure 6: JS program example that produces false positive classification - being c a sanitizer function

5.3 A more precise tool

To prevent the problem described in section 5.1, the *if* statement would have to create a fork in the code, in order to ensure that no information flows from the *then* block to the *else* block. Note that this would significantly increase the complexity of the code, because it would mean the creation of two sub-graphs that would then need to be reconnected at their ends and also lead to an increase in memory consumption because variables could became duplicated. On the other hand it would be more precise. As an example, the code in Figure 4, our program produces the graph in Figure 7, but a better solution would be create the one in Figure 8.

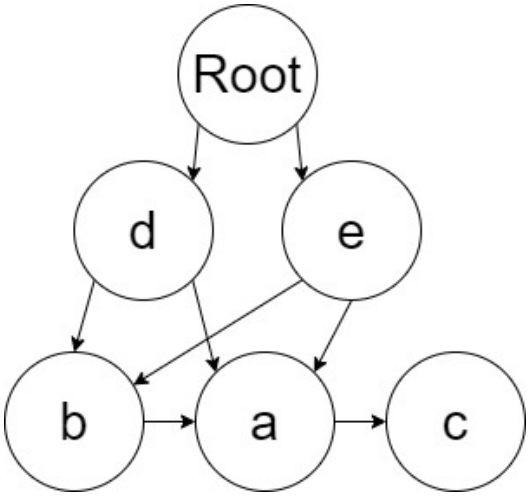

Figure 7: The way our codes handles if statements

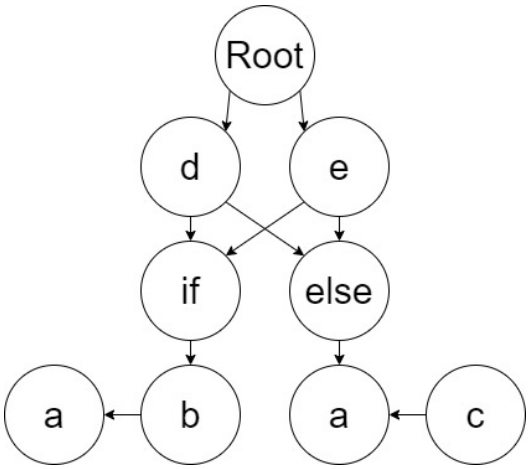

Figure 8: A better solution for handling if statements

6 Related Work

C. Staicu et. al.[1] created a static analysis tool (and a dynamic analysis tool) that detects explicit, observable implicit and hidden implicit flows and arrived at the conclusion that hidden implicit flows are extremely rare and the complexity of the code that must written in order to detect them is enormous and thus might not be worth it. In the second article by the same authors [2], they create a technique that extracts taint specifications from JavaScript libraries. This helps to make sure JavaScript files are not importing vulnerabilities from other JavaScript libraries.

S. Guarnieri et. al. [3] created Actarus, an algorithm for performing taint analysis in JavaScript code. As we stated before, we based our solution on this article. Actarus is however a much more robust and complete algorithm. The connections in the graphs created by Actarus make mathematical sense while ours is more based on *common sense*.

Hammer, et al. developed a similar flow sensitive graph-based taint analysis tool but for Java and Snelting, et al. extend their work by also taking into account arrays, pointers, abstract data types, and multithreaded programs.

7 Conclusion

In this report, we presented a simple tool that discovers web vulnerabilities in JavaScript code. Despite some imperfections on its precision detection, we could achieve great results on the standard tests given at the beginning of this project. Furthermore, it is versatile enough to detect even other different types of statements. However, in some types of codes slices, it should be way more precise, in order to improve our tool performance and reduce the detection of false positive cases.

8 References

[1] C. Staicu et. al. *An Empirical Study of Information Flows in Real-World JavaScript.* PLAS'2019.

[2] C. Staicu et. al. *Extracting Taint Specifications for JavaScript Libraries.* CSE'20.

[3] S. Guarnieri et. al. *Saving the World Wide Web from Vulnerable JavaScript.* ISSTA'11.

[4] G. Snelting et. al. *Efficent Path Conditions in Dependence Graphs for Software Safety Analysis.* TOSEM, 15(4), 2006.

[5] C. Hammer et. al. *Information Flow Control for Java Based on Path Conditions in Dependence Graphs.* S&P, 2006