Instituto Superior Técnico

Software Security

Group 10 - Taguspark

Discovering Vulnerabilities in JavaScript Web Applications

Miguel Levezinho
90756

Rafael Figueiredo
90770

Ricardo Grade
90774

1. Problem

Since JavaScript is a widely used language in client-side applications, it should be exempt of vulnerabilities that may cause harm to those applications, such as arbitrary code execution, path traversal attacks, identity thefts, among others.

For this reason, it is important that tools exist to detect if vulnerabilities are present in application code. This project aims to propose a tool that detects some illegal flows in JavaScript code in a static way, considering that an attacker has the ability of reading sinks during a program execution.

2. Approach

This tool can detect source-sink and source-sanitizer-sink flows since the completeness of a sanitizer is not ensured.

The solution accounts for the following constructs of the Esprima AST: Program, VariableDeclaration, VariableDeclarator, BlockStatement, ExpressionStatement, IfStatement, WhileStatement, AssignmentExpression, BynaryExpression, CallExpression, MemberExpression and Identifier. Other constructs are considered irrelevant.

2.1. Data Structures

Vulnerability Pattern (VP): Contains information about a specific vulnerability that was passed on in a provided pattern. It contains: Vulnerability Name, Vulnerability Sources, Vulnerability Sanitizers, Vulnerability Sinks, Vulnerability Associations (VA), to support aliasing of variables and names, and finally Vulnerability Links (VL), which stores all possible (incomplete) flows for the pattern in question.

Stack: Handles and stores information for local variables inside blocks. It contains: List of VPs, List of Identifiers Associations, and List of Identifiers Links. These last two fields serve as storage for the current flow states of certain variables when entering and exiting blocks (ifs or whiles).

Vulnerability: A source-sink or a source-sanitizer-sink flow. It contains: Name, Sources, Sanitizers and Sink.

Evaluator: Evaluates the AssignmentExpression, BynaryExpression and CallExpression. It contains: List of Vulnerability Patterns and List of Vulnerabilities. Responsible for managing and expanding the active flows.

Parser: Parses the provided AST. It contains: List of VPs, Stack, and Evaluator.

2.2. Logic

Variables Redeclaration: Variables redeclared using *let* keyword are pushed to the Stack when inside a block, to keep a copy of their VLs and VAs to allow the reposition of their original state on the end of that block.

Aliasing Process: Maps identifiers to its VA. It is Required to correctly evaluate the expressions.

Variable Assignment: When a variable gets assigned, all VLs and VAs are erased. After the aliasing process, the assignment is evaluated by attaching the right value VLs to the left value. If some become attached, the variable becomes tainted; otherwise, a VA is created to the right value. In the case where the left value is identified as a sink and the right value is tainted, a new vulnerability is discovered.

Call Expression: When a function gets called, after the aliasing process of its arguments and identifier, all VLs of its arguments get linked to a label that is created to identify the call. If the function that is being called is a **Source**, it is also linked to that label. If it is a **Sanitizer**, the sanitizer itself is linked to the label for cases where the label's flow (VLs) contains a source. If it is a **Sink**, in the case where the label has been attached with a known flow, a new vulnerability is discovered.

Binary Expression: When a binary operation is done, after the aliasing process of its components, all the links of the left value and right value get linked to a label created to identify this operation.

If Statement: For an If Statement, an evaluation of all possible flows is needed. These flows are defined as entering any given conditional statement or continuing the program. In order to make sure all possible flows are accounted for, copies of the VPs are created before each flow. After the evaluation, this copy is added to the List of the VPs. If the condition itself is tainted, then every assignment made inside the block also becomes tainted.

While Statement: The evaluation of the While Statement flows is analogous to the If Statement, with the difference being that this statement is evaluated twice, to account for out of order taint flows, possible due to looping.

3. Behaviour

The created tool receives a file path to a json file of a slice of a program in Esprima AST Syntax and a file path to a json file with the patterns of the possible vulnerabilities it is intended to find.

The tool parses the files and aims to extract all vulnerabilities in the slice based on the patterns it received. Outputs to a file the list of Vulnerabilities that was able to find.

4. Evaluation

For testing the tool, several example scripts were written [ref]. The scripts include different types of vulnerability flows to be tested, including explicit and implicit flows, using different types of constructs. Some tests also include cases that should result in no vulnerabilities, to see if the tool is tolerant to false positives.

Besides the script files themselves, each test case includes the following files, all written in json format: The script Esprima AST, a set of custom vulnerability patterns for that script and an expected output of found vulnerabilities.
A test is deemed 'passed' if the output of the tool matches the expected output file.

5. Critical Analysis

The tool created prevails the finding of all possible sink-sources or sink-sanitizer-sources over efficiency and tries to report all possible vulnerabilities, by analysing all possible paths that a program could follow when executed.

- The strengths of our tool are:

1) Explicit flows. Examples: (**Sources**: *redline, readbyte*; **Sink**: *exec)*

```
1. var link1 = readline();
2. var link2 = link1;
3. exec(link2);
```

- A vulnerability is happening at line 3, as *link2* was tainted by *link1* at line 2 which was linked to *readline* at line 1. [ref]

```
1. var link1 = readline();
2. var link2 = func(link1, readbyte());
3. exec(link2);
```

- A vulnerability is happening at line 3, as link2 was tainted by the arguments of *func*, link1 by the readline at line 1 and *readbyte* is a source. [ref]

```
1. var link1 = readline();
2. var link2 = link1 + readbyte();
3. exec(link2);
```

- A vulnerability is happening at line 3, as *link2* was tainted by the left and right values of the binary expression, link1 by the readline at line 1 and *readbyte* is a source. [ref]

2) Aliasing support. Examples: (**Source**: *document.URL*; **Sink**: *document.innerHTML)*

```
1. var alias = document.innerHTML;
2. alias = document.URL;
```

- A vulnerability is happening at line 2, as *alias* is associated with a sink and is being assigned to a source. [ref]

```
1. var alias = document;
2. alias.innerHTML = alias.URL;
```

- A vulnerability is happening at line 2, as *alias* represents the document object, our tool can detect it because it supports aliases even as a part of a member expression. [ref]

3) Sanitizer-Sink is not considered a vulnerability. Example: (**Sanitizer**: *encodeURI*; **Sink**: *document.innerHTML)*

```
1. var v = "Not Tainted";
2. var link = encodeURI(v);
3. document.innerHTML = link;
```

- There is no vulnerability happening in this slice, as *v* is not linked to any source, which causes *encodeURI* to not sanitize anything at line 2. [ref]

4) Attributes or methods of tainted members are also tainted. Example: (**Source**: *document.URL;* **Sink**: *document.write)*

```
1. var link = document.URL.substring(s, e);
2. document.write(link);
```

- A vulnerability is happening at line 2, as link is linked to *document.URL*, because this variable is tainted by its method *substring*. [ref]

5) All flows of an if/while are evaluated independently. Examples: (**Source**: *readline;* **Sanitizer**: *escape;* **Sink**: *exec)*

```
1. if (test) { link = readline(); }
2. else { link = escape(readline()); }
3. exec(link);
```

- Two distinct vulnerabilities are happening at line 3, as if the test condition evaluates to true *link* is linked to *readline* at line 1, otherwise *link* is linked to *escape* which sanitizes *readline*. Our tool outputs both. [ref]

```
1. link = document.URL;
2. while (test) { link = "Not Tainted"; }
3. document.innerHTML = link;
```

- A vulnerability is happening at line 3, as if the test condition evaluates to false *link* prevails linked to *document.URL*. [ref]

6) While Statement that is executed more that one time is evaluated. Example: (**Source**: *readline*; **Sink**: *exec*)

```
1. var link = "Not Tainted";
2. while (test) { exec(link); link = readline(); }
```

- A vulnerability is happening at line 2, as in the second iteration of the while loop *link* becomes linked to *readline*. [ref]

7) All redeclaration of variables using *let* are restored after an end of a block. Example: (**Source**: *readline*; **Sink**: *exec*)

```
1. let link = readline();
2. { let link = "Not Tainted"; }
3. exec(link);
```

- A vulnerability is happening at line 3, as *link* is linked to *readline()*, because all redeclarations of *let* variables are undone at the end of a block. [ref]

8) All tainted conditions are propagated to operations inside of an If / While block (Implicit Flow). Example: (**Source**: *document.URL*; **Sink**: *document.write*)

```
1. if (readline()) {
2.     document.write = "Constant Str";
3. }
```

- A vulnerability is happening at line 1, as the if condition is tainted by *readline*, the assignment inside of the if block is also linked to *readline*. [ref]

- The weaknesses of our tool are:

1) Our tool does not explicitly say which sanitizer was applied to which source. Example: (**Source**: *readline*, *readbyte*; **Sanitizer**: *escape*; **Sink**: *document.write*)

```
1. link1 = readline();
2. link2 = escape(readbyte());
3. document.write(link1 + link2);
```

- The reported vulnerability does not say that *escape* is being applied only to *readbyte*. So, it detail could be improved. [ref]

2) We can wrongfully consider a function pointer as a source or a sink. Example: (**Source**: *readline*; **Sink**: *exec*)

```
1. link = readline;
2. exec(link);
```

- If *readline* is a function, *link* is no source, but our tool does not distinguish it, because that information is not given in the patterns. Outputting a false positive. [ref]

3) As our tool follows all possible flows and does not evaluate the value of if/while conditions, although it outputs all illegal flows, there are flows in the program that could never happen. Example: (**Source**: *readline*; **Sink**: *exec*)

```
1. if (0) { exec(readline()); }
```

- As the if condition is always false, no vulnerability will happen. However, our tool does report the vulnerability at line 1. Outputting a false positive. [ref]

4) Only illegal flows using the tool evaluated constructs are analysed. Example (**Source**: *readline*; **Sink**: *exec*)

```
1. function vuln() { exec(readline()); }
2. vuln();
```

- As our tool does not evaluate function declarations it will not output the vulnerability it contains at line 1. Outputting a false negative. [ref]

5) Sanitizer functions can possibly not correctly sanitize the tainted sources.

- The (1) problem could be solved by storing more detailed information which would allow the tool to give a more detailed output. This would decrease the efficiency but would increase the tool precision.
- The (2) and (5) problems could be solved by receiving a more detailed input patterns. As the patterns would contain more detail it could decrease the efficiency, but it would be more precise.
- The (3) problem would be solved by evaluating, if possible, the values of the conditions. It would increase the complexity of the tool but would generate less false positives, becoming more precise.
- The (4) problem would be solved by extending the range of evaluated constructs of our tool, which would imply more complexity and time of parsing decreasing the efficiency but generating more positives.

6. Related Work

- **An Empirical Study of Information Flows in Real-World JavaScript** [ref]:
  - The goal of this paper is to analyse the costs and benefits of dynamically analysing explicit flows, observable implicit flows and hidden implicit flows.
  - The only techniques that can identify these three flows described in this paper are the NSU (No Sensitive Upgrade) and PU (Permissive Upgrade).
  - With these techniques, when an illegal flow is detected the program gets interrupted by a monitor, and therefore avoids the finding of other possible illegal flows. To solve this problem, it was proposed a modified PU that executes the program multiple times to insert Upgrade Statements each time the monitor interrupts the program allowing it to find all illegal flows.
  - Although this solution finds the source-sink pairs like our solution, it does not find source-sanitizer-sinks and additionally tracks all microflows, which are tainted variables, this is due to the fact of the goal of their solution is to analyse the number of flows that happen for each type, in order to better understand the costs and benefits of each of these types of flows.
- **Extracting Taint specifications for JavaScript Libraries** [ref]:
  - The Tazer program was developed with the same goal as the tool described in this document, that is, to develop a solution that detects vulnerabilities in JavaScript code in a correct and efficient way, but takes a very different approach, especially in terms of scope and context.
  - Firstly, Tazer was designed to be as detailed and correct as possible. Thus, the scope of Tazer reaches not only the client application, but also the modules used by the application, and all their dependencies. In contrast, our solution focuses only on the application code, and only on some supported constructs.
  - Secondly, our tool is made to be run in a static context, whereas Tazer was developed to be run in a dynamic context, using the test cases for each module to determine possible vulnerability flows, and applying that knowledge in a static context thereafter.
- **Saving the World Wide Web from Vulnerable JavaScript** [ref]:
  - The ACTARUS is the first JavaScript taint-analysis algorithm, that addresses a set of difficult challenges imposed by this language, such as prototype-chain property lookups, lexical-scoping rules for variable resolution, reflective property accesses and function pointers.
  - As our solution does not deal with functions, it does not need to use advanced assignments representations such as SSA (Static Single Assignment), which is used in ACTARUS to deal with scoped variables assignments assuring that each variable is assigned only once, keeping variables versions to address multiple assignments related to the same variable. Instead, the solution proposed in this document uses the Stack to discard the variables redeclared with the *let* keyword.
  - ACTARUS uses RHS (Reps-Horwitz-Sagiv) to map aliasing relations into a graph. Our solution implements associations relations, explained above, which is much simpler and equally solves all aliasing issues that are being considered on our problem.

7. Conclusion

To mitigate the problem that JavaScript code is vulnerable, our tool aims to find possible vulnerable code flows within the defined scope, and thus shed some light on the efficacy of the security methods being used, or their absence. This tool is good for simple screening of JavaScript code but lacks some fundamental constructs to be usable in production code. Even so, it utilizes some complex concepts used in far more complete tools to maximize its output value. This tool has potential to grow, specially by adding more supported code constructs.