

Identifying Algorithmic Complexity Vulnerabilities Caused by Input-Dependent Nested Loops

Mujahid Masood, Rafique Nazir

mujahid.masood@stud.tu-darmstadt.de, rafique.nazir@stud.tu-darmstadt.de

1. Problem Statement

The single-threaded event model of JavaScript makes it vulnerable to a specific class of denial of service attack called algorithmic complexity attacks. These attacks consist of exploiting the worst case performance of algorithms to trigger slow computations that block the event loop for a large period of time. The focus of this project is to

- Identify the functions in code.
- Identify the input parameters to the functions.
- Identify the loops using input parameters of functions.
- Impact of using function input parameters with loops in terms of execution time as well as denial of service attack.

2. Input dependent loops and execution time

Execution time of the functions is really important to write scalable applications. In normal application overall execution time of function depend on the slowest function.

Consider the code in Listing 1 we have iterate function which has max as input parameter, it has one loop which is iterating up till max.

So caller of function can pass input of 10_{10} and execution time of simple function with 1 loop will be around 11s.

Listing 1. iterate function with 1 loop

```
1 function iterate(max){
2
3
4     var start = process.hrtime();
5     var precision = 3;
6     for(var i = 0; i < max; i++){
7         var c = 10 + 5;
8     }
9
10    var elapsed = process.hrtime(start)[1] / 1000000;
11
12    console.log(
13        process.hrtime(start)[0] + " s, "+
14        elapsed.toFixed(precision)+" ms - "
15    );
16 }
```

If other functions in application are taking less time, iterate function is subject to performance bug and also security bug specially in node modules.

This problems gets interesting if we introduce nested loops i.e, 2 or 3 nested loops. Consider the code in Listing 2. Function double loop has input parameters max and 2 nested loops. Client of doubleLoop can pass input max 105 and can introduce as delay

of around 6 s, 491.041 ms . Important thing to note is function is only doing simple sum of 10 and 5 but due to input dependent loop execution time of function takes much time.

Listing 2. doubleLoop function with 2 nested loop

```
1 function doubleLoop(max){
2
3
4     var start = process.hrtime();
5     var precision = 3;
6     for(var i = 0; i < max; i++){
7         for(var j=0; j < i; j++){
8             var c = 10 + 5;
9         }
10    }
11
12    var elapsed = process.hrtime(start)[1] / 1000000;
13    console.log(
14        process.hrtime(start)[0] + " s, "+
15        elapsed.toFixed(precision)+" ms - "
16    );
17 }
18
19 doubleLoop(Math.pow(10,5));
```

As we increase the nested loops input to the parameter gets smaller. A function with 3 nested loops and input parameter max can have execution time of around 6 s with max = 103.2 Consider the code in Listing 3.

Listing 3. tripleLoop function with 2 nested loop

```
1 function tripleLoop(max){
2
3
4     var start = process.hrtime();
5     var precision = 3;
6     for(var i = 0; i < max; i++){
7         for(var j=0; j < i; j++){
8             for(var k=0; k < max; k++){
9                 var c = 10 + 5;
10            }
11        }
12    }
13
14    var elapsed = process.hrtime(start)[1] / 1000000;
15    console.log(
16        process.hrtime(start)[0] + " s, "+
17        elapsed.toFixed(precision)+" ms - "
18    );
19 }
20
21 tripleLoop(Math.pow(10,3.2));
```

3. Problems with using loops depending on function input parameters

Code in Listing 2 and Listing 3 are not only subject to performance issues but also to denial of service (DoS) attacks. Attacker can control the input parameter and can introduce delay of 10-15 seconds in the execution time of function. Node.js security experts consider any slowdown larger than one second as security relevant.

4. Avoiding the problem

Following are some approaches which can be used to avoid the problem with input dependent nested loops.

4.1 Approach 1 : Introduce Upper bound on input parameters

One simple way to avoid such problem is introducing upper bound on input parameter. Consider the code in Listing 4 which exits from the function if input exceeds the given bound which is 10^2 .

Listing 4. avoidProblem function with 3 nested loop

```
1 function avoidProblem(max){
2   if(max > Math.pow(10,2)) {
3     return;
4   }
5   for(var i = 0; i< max; i++){
6     for(var j=0; j< i; j++){
7       for(var k=0; k< max; k++){
8         var c = 10 + 5;
9       }
10    }
11  }
12 }
13 }
14
15 avoidProblem(Math.pow(10,3.2));
```

4.1.1 Problems with Approach 1

Introducing such input bound checks as in Listing 4 requires full understanding of the code and also the usage of function.

4.2 Approach 2 : Changing the logic

Other way can be changing the logic from nested loops to maybe introducing new functions which means one needs to find places where input dependent functions with loops are used in the code.

4.2.1 Problems with Approach 2

- Code is already deployed in production environment.
- JavaScript uses minified versions of files.
- JavaScript file can have 10000s lines of code.
- Different variations of functions.
 - JavaScript can use assignment of function to other variable e.g, var a = function()
 - Using anonymous functions e.g, (function())
 - JavaScript also uses nested functions e.g, (function(a, function()))
- Different variations of Loops e.g For, While, For In, forEach etc.
- Assignment of function input parameter to other variables and using that in loop.
- Using input parameter in function call which in turn uses for loop.

5. Our Approach

5.1 High Level Steps

- Statically analyse JavaScript code.
- Find out if code uses function.
- Find out if function uses input parameters.
- Find out if function uses loops.
- Find out if loops use input parameters directly or indirectly.
- Write vulnerable output.
- Manually validate the results.
 - Manually trigger the identified function.
 - Calculate the execution time of function on different inputs

5.2 Implementation Technologies

- Static Analyser : Google Closure Compiler for static analysis.
- Programming Language : Java 8
- Build tool : Maven
- Bash Script (Utility)
 - clone.sh : cloning git code to specified directory
 - search.sh: Bash script to search JavaScript Projects on Github, opens the browser session for displaying results.

5.3 Detailed Steps

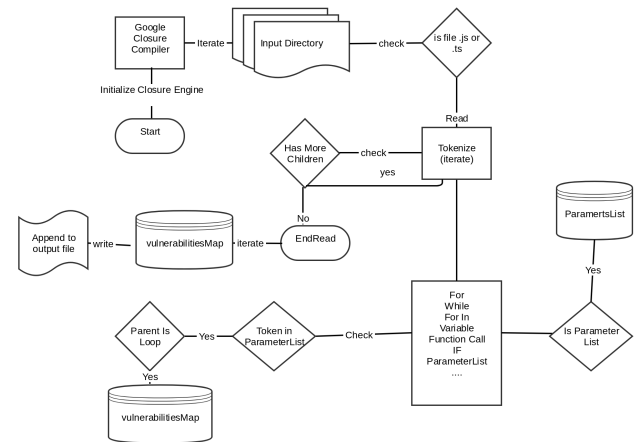


Figure 1. Implementation Detail.

- Iterate through the directory.
- Check for any .js or .ts file.
- Initialize Google Closure Engine.
- Read the JavaScript File.
- Tokenize the JavaScript code.
- Iterate through the tokens.
- Check if the token is function, save the reference for future use.
- Check if token is the parameter list, add it in parameter list.
- Check if the variable is present in parameter list.

- Check if the parent of variable is loop.
- If variable is loop and present in parameters, add it into vulnerabilitiesMap.
- At the end of all tokens print the output in text file.

5.4 Project Structure

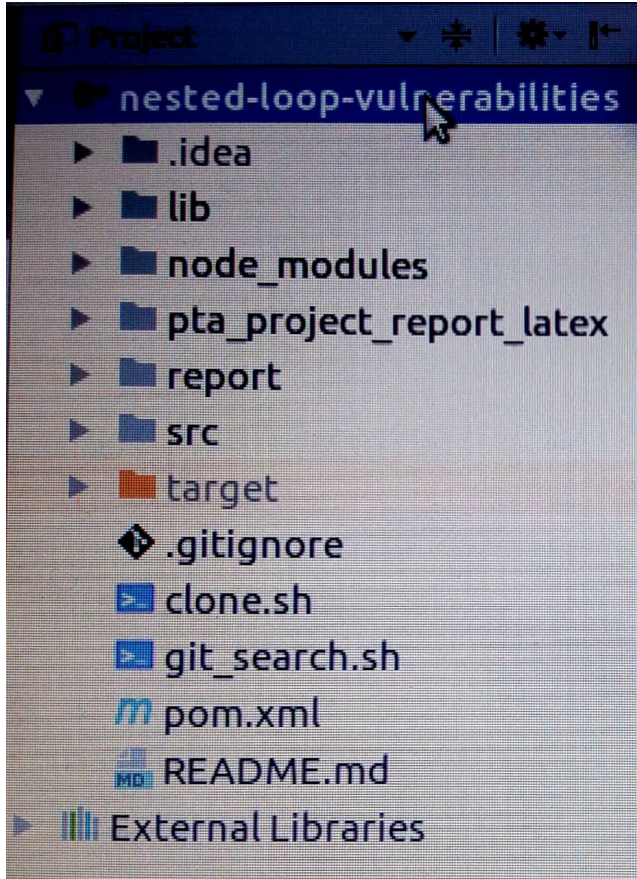


Figure 2. Project Structure.

- lib : This folder contains libraries.
- node modules : This folder contains the npm modules on which we have run the analysis.
- report : Contains the project report.
- src : Contains the source classes (Java Classes), actual code with implemented logic.
 - main
 - AnalyzerClient : Main class to invoke the analyser.
 - ClosureEngine : Class for Google closure related settings, mainly initializes the Google Closure Compiler.
 - OutputWriter : Class to write the output vulnerabilities in text file.
 - StaticAnalyzer : Class has main logic, iterates the JavaScript code, checks for functions, loops and decides vulnerabilities.
 - test
 - resources : Contains our test cases and output files

- .gitignore : Contains git ignored files.
- clone.sh : Bash script to clone the git directory to specified location.
 - It takes two input parameters git clone directory and location where to clone the project for example to run the file. sh clone.sh https://github.com/project.git /nested-loop-vulnerabilities/node modules
 - It will create new folder named project.git under node modules directory and clone the project in it.
- search.sh : Searches the JavaScript projects on github, to show the results it opens google chrome browser.
- pom.xml : It contains maven related configurations.
- README.md : Contains the instructions on how to use project.

6. Results

Following section discusses the results on our own test cases and also on node modules. In Project code index.js under resources folder contains our test cases, we will demonstrate a few from index.js file.

6.1 Manual Tests

```

70 function functionCall(max) {
71     var a = max.length;
72     for (var i = 0; i < a; i++) {
73         console.log(a);
74     }
75 }
76
77 function arrayLength(array) {
78     for (var i = 0; i < array.length; i++) {
79     }
80 }
81
82 function loopWithVarAssignemnt(max, min) {
83     var y = min - 10;
84     var x = y;
85     for (var y = 0; y > min; x++) {
86         for (var a = 0; 0 || 1; z++)
87             for (var a = 0; 0 && 2; z++)
88                 console.log("a" + a)
89     }
90 }
91 function whileConditional(max) {
92     while (max > 10 || max < 10)
93         console.log("max = " + max);
94 }
95
96 function whileLoop(max) {
97     while (true) {
98         while (x) {
99             while (max % 10) {
100                 console.log("max = " + max);
101             }
102         }
103     }
104 }

```

Figure 3. Manual Tests.

From the figure 4 we expect to find vulnerabilities on lines 72,78, 85, 92 and 99 because the functions use input dependent loops. We ran the tests using our code and output is as below.

As we can see our code correctly identifies the specified lines where possible vulnerabilities can occur along with some false positives. Output format is

- First line is file name for which analysis is done.
- Multiple dashes for separation.
- Subsequent lines shows function name in left column and line number in right column. In case of anonymous function output function on left is null.

```

1  src/test/resources/index.js
2  -----
3  functionCall          72
4  functionCall          73
5  arrayLength           78
6  loopWithVarAssignemnt 85
7  loopWithVarAssignemnt 86
8  whileConditional      92
9  whileConditional      93
10 whileLoop             99
11 whileLoop            100
12

```

Figure 4. Output of Manual Tests.

6.1.1 Validation of claim

With above functions validation is easy we added execution time to validate our claim as shown in Listing 2, 3

7. Analysing Node modules

(Aho et al. 1986)

References

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6.