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

```

function iterate(max){

    var start = process.hrtime();
    var precision = 3;
    for(var i = 0; i< max; i++){
        var c = 10 + 5;
    }

    var elapsed = process.hrtime(start)[1] / 1000000;

   console.log(
      process.hrtime(start)[0] + " s, "+
      elapsed.toFixed(precision)+" ms - "
   );
}
```

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

```

function doubleLoop(max){

    var start = process.hrtime();
    var precision = 3;
    for(var i = 0; i< max; i++){
      for(var j=0; j< i; j++){
          var c = 10 + 5;
      }
   }

    var elapsed = process.hrtime(start)[1] / 1000000;
   console.log(
      process.hrtime(start)[0] + " s, "+
      elapsed.toFixed(precision)+" ms - "
   );
}

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

```

function tripleLoop(max){

    var start = process.hrtime();
    var precision = 3;
    for(var i = 0; i< max; i++){
      for(var j=0; j< i; j++){
         for(var k=0; k< max; k++){
             var c = 10 + 5;
         }
      }
   }

    var elapsed = process.hrtime(start)[1] / 1000000;
   console.log(
      process.hrtime(start)[0] + " s, "+
      elapsed.toFixed(precision)+" ms - "
   );
}

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

```

function avoidProblem(max){
   if(max > Math.pow(10,2)) {
      return;
   }
    for(var i = 0; i< max; i++){
      for(var j=0; j< i; j++){
         for(var k=0; k< max; k++){
             var c = 10 + 5;
         }
      }
   }
}

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

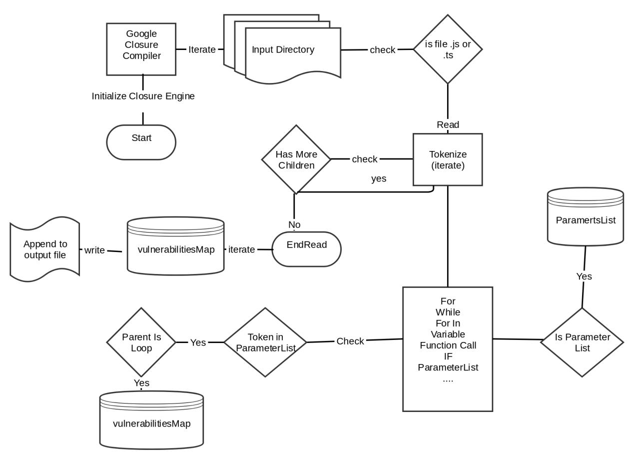

Figure 1. Implementation Detail.

- 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

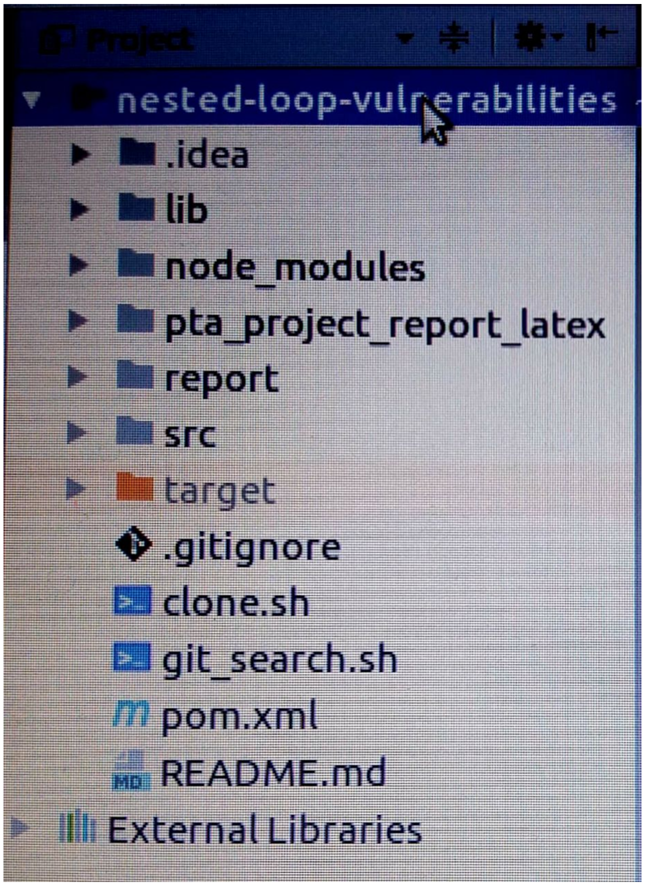

Figure 2. Project Structure.

- lib : This folder contains libraries.
- node modules : This folder contains the npm modules on which we have run the analyis.
- 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.

```
src/test/resources/index.js
-----------------------------------------------
functionCall                                     72
functionCall                                     73
arrayLength                                      78
loopWithVarAssignemnt                            85
loopWithVarAssignemnt                            86
whileConditional                                 92
whileConditional                                 93
whileLoop                                        99
whileLoop                                        100
```

Figure 4. Output of Manual Tests.

6.1.1 Validation

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

7. Analysing Node modules

We run our analysis on following node modules. In order to identify nested loops we mainly focused on video streaming,audio streaming, matrix and data structure libraries to find the code using nested loops. Table 1 shows how a table looks like.

Node Module	Total input-dependent loop vulnerabilities	Correctly Identified	False Positives
loaddash	A modern JavaScript utility library delivering modularity, performance and extras.	only 1 for loop	yes
dashjs	Library to quality framework for building video and audio players that play back MPEG-DASH	2 for loop	yes
useragent	Useragent allows you to parse user agent string with high accuracy by using hand tuned dedicated regular expressions for browser matching.	only 1 for loop	yes
lru cache	This is an LRU (least recently used) cache implementation in JavaScript.	0	yes but false positives

Table 1. Result of analyzing node modules.

7.1 Validating Node Modules

In above figure 5 code on line numbers 418 and 419 is vulnerable.After running our code we identified our code on node module user agent we identified the results as shown in the figure

```javascript
408 /**
409  * Check if the userAgent is something we want to parse with regexp's.
410  *
411  * @param {String} userAgent The userAgent.
412  * @returns {Boolean}
413  */
414 function isSafe(userAgent) {
415     var consecutive = 0
416         , code = 0;
417
418     for (var i = 0; i < userAgent.length; i++) {
419         code = userAgent.charCodeAt(i);
420         // numbers between 0 and 9, letters between a and z
421         if ((code >= 48 && code <= 57) || (code >= 97 && code <= 122)) {
422             consecutive++;
423         } else {
424             consecutive = 0;
425         }
426
427         if (consecutive >= 100) {
428             return false;
429         }
430     }
431
432     return true
433 }
434
435
```

Figure 5. Node module useragent isSafe function

```
/home/mujahidmasood/Masters/DSS/Semester5/PTAA/nested-loop-vulnerabiliti
-----------------------------------------------
Agent                                          35
Agent                                          36
Agent                                          37
Agent                                          38
Agent                                          39
set                                            88
set                                            139
toString                                       164
toString                                       166
OperatingSystem                                223
OperatingSystem                                224
OperatingSystem                                225
OperatingSystem                                226
Device                                         296
Device                                         297
Device                                         298
Device                                         299
updating                                       369
updating                                       371
updating                                       375
isSafe                                         418
isSafe                                         419
parse                                          457
parse                                          466
parse                                          475
parse                                          479
parse                                          484
parse                                          495
lookup                                         517
fromJSON                                       591
fromJSON                                       594
fromJSON                                       597
fromJSON                                       598
fromJSON                                       603
fromJSON                                       607
fromJSON                                       608
fromJSON                                       610
```

Figure 6. Node module useragent isSafe function

To validate,we introduced function execution time at the start and at the end of loop inside a function as shown in figure 7 below:

7.2 Manually Triggering Identified Vulnerable Function

A random test script of length 10^6 was provided as an input to a parse function and it took approximately 5 seconds to execute the method as shown in figure below:

(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.

```javascript
/**
 * Check if the userAgent is something we want to parse with regexp's.
 *
 * @param {String} userAgent The userAgent.
 * @returns {Boolean}
 */
var process = require('process');
var precision = 3;
function isSafe(userAgent) {
  var start = process.hrtime();
  var consecutive = 0
    , code = 0;

  for (var i = 0; i < userAgent.length; i++) {
    code = userAgent.charCodeAt(i);
    // numbers between 0 and 9, letters between a and z
    if ((code >= 48 && code <= 57) || (code >= 97 && code <= 122)) {
      consecutive++;
    } else {
      consecutive = 0;
    }

    if (consecutive >= 100) {
      return false;
    }
  }

    var elapsed = process.hrtime(start)[1] / 1000000;
    console.log(process.hrtime(start)[0] + " s, " + elapsed.toFixed(precision) + " ms - " );
  return true
}
```

Figure 7. Node module useragent isSafe function

```javascript

var useragent = require('useragent');
useragent(true);



var req = {}
var string = ''
for(var i=0; i< 10000; i++){
    string += i + " ";
}

req.headers = {'user-agent': string}
req.query = {'jsuseragent':'ok'}
var agent2 = useragent.parse(req.headers['user-agent'], req.query.jsuseragent);
```

Figure 8. Node module useragent isSafe function