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 function input-dependent nested loops by:

- Identifying the functions in code.
- Identifying the input parameters of the functions.
- Identifying the input-dependent nested loops inside the functions.
- Analyzing the impact of function input-dependent 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 depends upon the slowest function.

Consider the code in Listing 1 in which we have an iterate function,with function input parameter max,which has a single loop,iterating up till max.

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

Listing 1. iterate function with single loop

```
1
2  function iterate(max){
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 doubleLoop has input parameter max and two nested loops. Client of doubleLoop can pass input max 10^5 and can introduce a delay of around 6 seconds. Important thing to note is that function is only doing simple sum of 10 and 5 but due to input dependent loop, execution of the function takes longer time.

Listing 2. doubleLoop function with two nested loops

```
1
2  function doubleLoop(max){
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 number of nested loops,to obtain longer execution time of function,we need to pass smaller function input parameter.For example a function with 3 nested loops and input parameter max can have execution time of around 6 s with max = $10^{3.2}$ Consider the code in Listing 3.

Listing 3. tripleLoop function with three nested loops

```
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 Function Input-Dependent Loops

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 loops and upper bound on input parameter

```
1
2  function avoidProblem(max){
3     if(max > Math.pow(10,2)) {
4        return;
5     }
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
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 in the code where input-dependent nested loops are used, for example not using nested loops.

4.2.1 Problems with Approach 2

- Code is already deployed in production environment.
- JavaScript uses minified versions of files therefore it is difficult to find code usage.
- JavaScript file can have 10000s lines of code,finding such occurrences in code can be difficult.
- 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 is used in loops.

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 function input parameters directly or indirectly.
- Write vulnerable output.
- 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 Implementation: Detailed Steps

Figure 1 shows the implementation details.

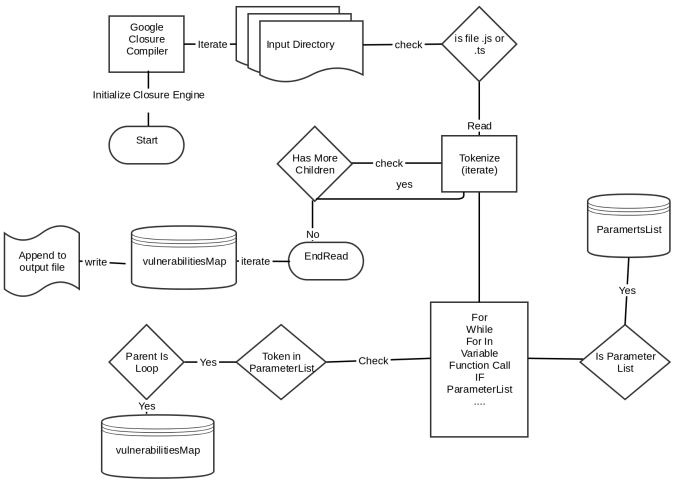

Figure 1. Implementation Detail.

- 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 parameterList, add it in parameter list.
- Check if the variable is present in parameter list.
- Check if the parent of variable is loop (while, for, forEach, for in).
- If parent of variable is loop and present in parameters, add it into vulnerabilitiesMap.
- After traversing all tokens in file, print the output in text file.

5.4 Project Structure

- lib : This folder contains jar files.
- node modules : This folder contains the npm modules on which we have performed the analysis.
- src : Contains the source code (Java Classes), actual code with implemented logic.
  - main
    - AnalyzerClient : Main class to invoke the StaticAnalyser.
    - 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 repository to specified location.
  - It takes two input parameters 1. repository and 2. location where to clone the project
    clone.sh can be run as shown below

    sh clone.sh https://github.com/project.git /node modules

    It will create new folder named project.git under node modules directory and clone the project from given online location.
- search.sh : Searches the JavaScript projects on github, to display 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.

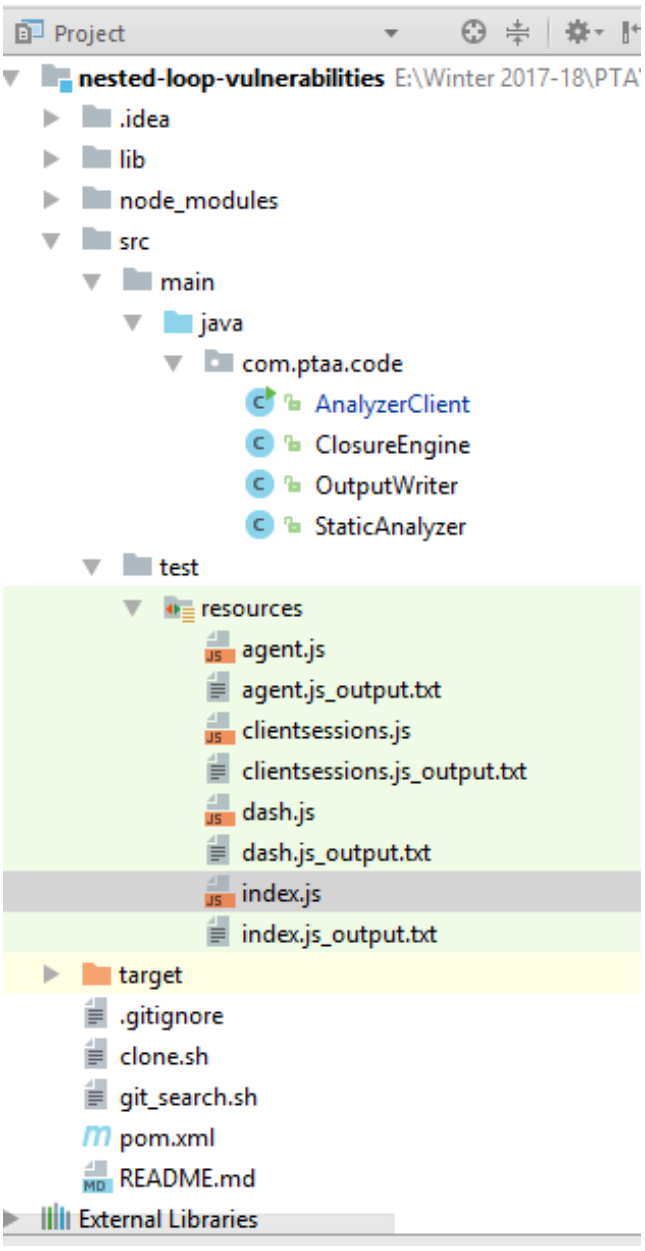

Figure 2. Project Structure.

From the figure 3. 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 shown in figure 4.

6.1 Manual Tests

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.

```
function functionCall(max) {
    var a = max.length;
    for (var i = 0; i < a; i++) {
        console.log(a);
    }
}

function arrayLength(array) {
    for (var i = 0; i < array.length; i++) {
    }
}

function loopWithVarAssignemnt(max, min) {
    var y = min - 10;
    var x = y;
    for (var y = 0; y > min; x++) {
        for (var a = 0; 0 || 1; z++)
            for (var a = 0; 0 && 2; z++)
                console.log("a" + a)
    }
}
function whileConditional(max) {
    while (max > 10 || max < 10)
        console.log("max = " + max);
}

function whileLoop(max) {
    while (true) {
        while (x) {
            while (max % 10) {
                console.log("max = " + max);
            }
        }
    }
}
```

Figure 3. Manual Tests.

```
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

In order to identify nested loops we mainly focused on video streaming,audio streaming, matrix and data structure libraries to find the code using input dependent nested loops. We found nested loops inside. All these libraries and simulations are inside node modules folder.

- tough-cookie
- dashjs
- loaddash

We identified vulnerabilities in other npm modules with single loop. All these simulations are inside resources folder of project.

- useragent
- clientsession
- glmatrix

We performed our analysis on following node modules. Table 1 shows how a table looks like.

Node Module	Has Nested Loops	Found Vulnerabilities	False Positives
loaddash	yes	yes	yes
dashjs	yes	yes	yes
useragent	no	yes	yes
tough-cookie	yes	yes	yes

Table 1. Result of analyzing node modules.

7.1 UserAgent Results

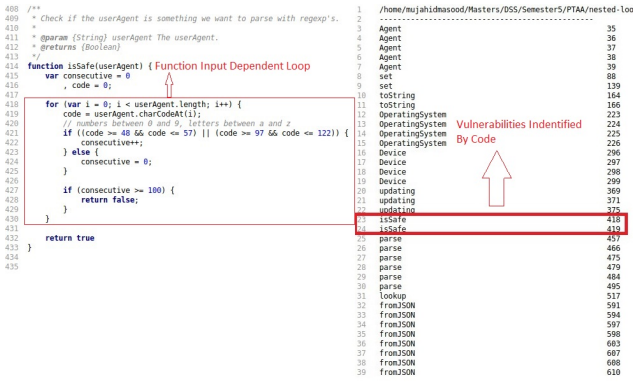

Figure 5. Node module useragent isSafe function

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 5

7.2 Validating UserAgent Results

We added the code to calculate execution time of function in the start and end of parse function as shown in Listing 5

Listing 5. Code Instrumentation

```

/**
 * Parses the user agent string with the generated
      parsers from the
 * ua-parser project on google code.
 *
 * @param {String} userAgent The user agent string
 * @param {String} [jsAgent] Optional UA from js to
      detect chrome frame
 * @returns {Agent}
 * @api public
 */
var process = require('process');
var precision = 3;
exports.parse = function parse(userAgent, jsAgent) {
    var start = process.hrtime();
  if (!userAgent || !isSafe(userAgent)) return new
       Agent();

  var length = agentparserslength
    , parsers = agentparsers
    , i = 0
    , parser
    , res;

  for (; i < length; i++) {
    if (res = parsers[i][0].exec(userAgent)) {
      parser = parsers[i];

      if (parser[1]) res[1] = parser[1].replace('$1',
           res[1]);
```

```
      if (!jsAgent) return new Agent(
          res[1]
        , parser[2] || res[2]
        , parser[3] || res[3]
        , parser[4] || res[4]
        , userAgent
      );

      break;
    }

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

7.3 Manually Triggering Identified Vulnerable Function

A random test script of length 10^4 was provided as an input to a parse function and it took approximately 7 seconds to execute the method as shown in listing 6 below:

Listing 6. Manual Triggering of vulnerable function in npm module useragent

```
var useragent = require('useragent');
useragent(true);
var req = {}
var string = ''
for(var i=0; i< Math.pow(10,4); i++){
    string += i + " ";
}

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

7.4 tough-cookie Results

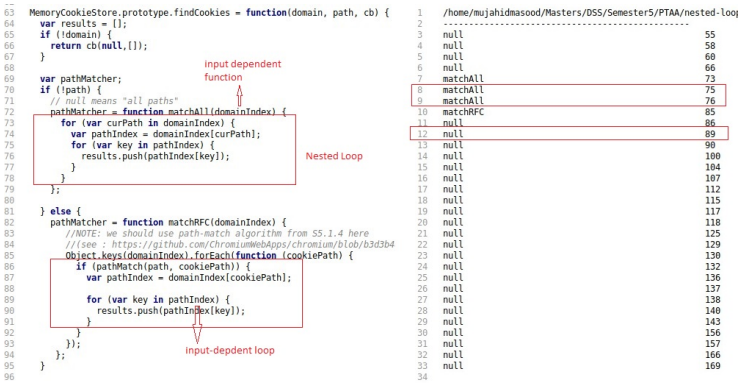

Figure 6. vulnerabilities in npm module tough-cookie in file lib/memstore.js

We found nested loops in tough-cookie as shown in figure 6.

8. Conclusion

This project correctly analyses the vulnerabilities along with 80 percentage of false positives.

8.1 Reason for false positives

Our implementation checks if the input parameter is used, if it is used at any instance of variable, function call or assignment, we output it as vulnerable. This is not the scope of project but technically some function call can make take much longer time for execution because of used parameter.

9. Future Work

- Reducing the number of false positives in the code this can be done by properly defining and documenting vulnerabilities. For example if we have 3 nested functions and all have parameters but inner function as loop which uses parameters, should this be considered vulnerable or not.
- Writing the bash or python script to make the search code with nested loop instances easy to find on github or other code repositories. for example
  git grep -l –all-match -e "for" –or -e "while"
  this command find for or while loop instance in code.
- Validating the code is bit challenging because vulnerable function can be dependent on many other calling functions and to really see it in practice one needs to fulfil all dependencies.

10. Difficulties and Challenges

Following are some places we found it difficult

- The code instances which use nested loops.
- Triggering the identified code because functions identified had many dependencies.
- Coming up with input parameter, we solved this by creating random input in case of useragent.