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 **??** 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
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 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
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 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
2  function tripleLoop(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          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 **??** and Listing **??** 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 **??** which exits from the function if input exceeds the given bound which is 10^2.

Listing 4. avoidProblem function with 3 nested loop

```
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 **??** 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.

(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.

C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

W. S. Jr. and E. B. White. *The Elements of Style*. 2000.