NoSQL Injection Detection Using Context-Free Grammars

Shaun Mathew
Department. Of Computer Science
Ryerson University
Toronto, Canada
shaun.mathew@ryerson.ca

***Abstract*— NoSQL databases allow for structured, unstructured and semi-structured data to be stored in an efficient and scalable manner. According to db-engines, MongoDB is ranked as the 5th most popular Database Management System (DBMS) with its Document Oriented Storage model. Despite its popularity, security research into attacks on NoSQL type databases are still in its infancy as opposed to its more mature twin, SQL. We propose a sound and complete algorithm, dubbed NoSQLCheck, for detecting NoSQL injection attacks based on the work of Wasserman and Su for detecting SQL injection attacks. We also provide an easily includable package in Node.js, an extremely popular tool for creating web applications, to detect NoSQL injection attacks. We achieve a 100% detection rate on the attack vectors that were tested.**

Keywords—NoSQL, Injection, Detection, SQL, MongoDB

I. Introduction

According to OWASP's 2017 list of top 10 vulnerabilities, injection attacks are ranked as the number 1 threat to web applications [1]. Due to its prevalence, several techniques have been proposed that tackle the problem at different layers. Prepared statements handle the problem by compiling the statement with placeholder variables. Since these statements have already been compiled, the user input is simply passed in as a literal. Prepared statements have some drawbacks, namely, they only work if the input we are expecting is to be treated as a string or numeric literal. If we need any form of dynamism in the query, such as allowing the user to choose specific columns in some data retrieval application or allowing for the use of conditions in their queries, prepared statements cannot be used. Sanitization is a method that attempts to cleanse the user input of any malicious strings that may cause unwarranted code execution. The issue with this method is that not all malicious strings may be properly sanitized. Wasserman and Su propose a method to detect SQL injection attacks by intercepting requests sent to the database and then determining whether they can result in an SQL injection attack based on the programmer's requirements [2]. The method proposed relies on context free grammars that defines the language of "benign" strings based on a policy function that lists what types of inputs the programmer would like to allow. This method allows for the application to accept a variety of input forms, no longer restricting the types of applications we can develop. Wasserman and Su's SQLCheck algorithm was implemented for PHP and JSP web applications and prevented all SQL injection attempts and allowed all benign strings.

In our paper, we implement a variant of the SQLCheck algorithm for the **NoSQL database system, MongoDB**. We develop this **NoSQLCheck** algorithm for web applications developed in the Node.js runtime environment. As of 2019, Node.Js is one of the most largely downloaded open source softwares for implementing Javascript code exterior to a browser. According to a 2019 developer survey by stackoverflow, Node.Js is ranked as the most popular technology for web developers and Javascript is ranked as the most commonly used programming language [3]. Given the ubiquitous nature of Node.js, an easily includable library that detects and alerts users of NoSQL injection attacks would be a noteworthy addition. We also perform a real-world evaluation of NoSQLCheck with various attack vectors found around the web. Since NoSQL isn't as old as SQL, there are few to no attack vector lists. We therefore have to craft our own from various resources found around the web that describe NoSQL injection attacks. Some sources include this github repository [4] and OWASP's NoSQL Injection checklist [5]; we also try several benign examples to make sure NoSQLCheck is not blocking valid requests.

II. Overview of approach

A. NoSQLCheck Algorithm

Wasserman and Su's SQLCheck algorithm has 3 main phases – input tracking, grammar construction and query injection detection [2]. The first phase involves tracking the user-input through various string manipulation functions such as NoSQL sanitization functions. This is accomplished by using a random sequence of characters that surround the user input. We shall refer to these random sequences as metacharacters and are denoted by " ☾ " and " ☽ ".

In the grammar construction phase, we first construct a context free grammar that describes the structure of all possible NoSQL queries. MongoDB's query syntax follows a JSON-like structure; this, in addition to MongoDB's query reference, allows us to create a grammar that is tailored to the NoSQL syntax. In its unaugmented state, the grammar can be used to construct a parser that can detect whether a query has valid syntax. Note that the parser does not check issues pertaining to the compilation of the query, such as, referencing fields in the query that do not exist in the document. The parser constructed from this grammar is also incapable of detecting injection attacks. In order to do so, we must augment the grammar using a predefined or user defined a policy. A policy defines a set of non-terminals that specify what forms the user's input can take. For example, if we had a policy that says ["num"], all user input supplied to the program can only have a numeric form. Passing in an arbitrary javascript object or even a string will cause the parser to throw an error. For each term in the policy, we construct a new terminal as follows: term_a -> term | ⦅ term ⦆. We then replace every right hand side reference of term in the grammar with term_a.

Lastly, in the query injection detection phase, we construct our query parser using an Earley Parser due to its ability to handle left-recursive grammars and parse all context-free languages [6]. We pass our NoSQL query with ⦅user-input⦆ to our parser. If the parser fails to parse the query, it is either malformed or is a NoSQL injection attack. On the other hand, if the query parses correctly, it is a benign query and thus we can pass our query (without the metacharacters) to the DBMS.

B. MongoDB NoSQL Query Syntax

MongoDB stores records as documents. These documents are stored in collections which are analogous to JSON objects.

```
{
   "name": "Shaun",
   "age": 24,
   "likes": ["Computers", "Security"],
   "school" : {
            "name" : "Ryerson",
            "Country": "Canada"
         }
}
```

MongoDB query syntax is based heavily on JSON syntax (barring some technicalities), which consists of key value pairs, where every key is either a MongoDB specific keyword or a specific field in document. Values are quite a bit more complicated since they can take on a plethora of values. The MongoDB reference is not clear on what particular types of values are allowed and lastly the type of allowed values can change depending on the key. We provide a few examples to further explain the query syntax.

```
db.students.find({name: "Shaun", "school.name":
"Ryerson"})
```

This query finds a student whose name is Shaun and who goes to a school known as Ryerson.

```
db.students.find(db.students.find({"school.name":
"Ryerson", "likes" : {"$in": ["Computers"]}})
```

This query finds a student who goes to Ryerson and likes Computers. Notice how we are using the query operator $in to see if "Computers" is in the list of what the students likes.

```
db.students.find(db.students.find({age : {$gt: 23}})
```

This query finds all students who are older than the age of 23. Notice how keys can be either quoted or unquoted. This is an important distinction to make when constructing our NoSQL grammar, since adhering solely to the JSON specification would result in benign queries being rejected due to parser failing to parse them.

C. An Example Attack

Consider the following code excerpt written in Node.js using the mongoose package.

```
app.post('/login_unsafe', function(req, res) {

  var username = req.body.username;
  var password = req.body.password;

  var query = {"username" : username, "password":
  password};

  User.find(query).then(function(value) {
       res.send(value);
     })

});
```

Fig. 1. Vulnerable Node.js Code

Here the username and password are stored in their respective variables and are used to construct a NoSQL query. That query is then passed to a find function which simply replies with the result of the query. Note that in a more realistic scenario, the user would be redirected to a welcome page.

A malicious user can exploit the fact that no sanitation or input validation is used in order to construct a malicious query that will allow him to login as any user. Consider the following raw JSON sent to the server:

```
{
"username": "Shaun",
"password": {"$gt": ""}
}
```

Fig. 2. Exploiting login_unsafe route

Such as query allows the user to login as “Shaun” without requiring a password.

III. ALGORITHM AND FORMAL DEFINITIONS

A. *Formal Definitions*

Wasserman and Su propose some formal definitions in order to motivate their algorithm. We briefly summarize them here with some examples for clarity.

Web Application

A web application is defined as a set of inputs $\{i_1 \dots i_n\}$ that are passed to or not passed to some set of filter functions $\{f_1 \dots f_n\}$. These filtered inputs, f(i), are then combined with a set of static strings $\{s_1 \dots s_n\}$ to construct a query q. Looking at an excerpt from our code example in Figure 1, var query = {"username" : **username**, "password": **password**}, the underlined terms would be our static strings and the **bolded terms** would be our filtered inputs. Note that in our example, we are not using any filter functions (e.g. sanitization functions) and thus the filter functions can simply be considered as being simple identity functions.

Valid Syntactic Forms

An input is considered to have a valid syntactic form if it can be generated by some policy U. For example, consider the policy [“num”]. Any input that can be generated using the production rules for num, i.e. numbers, are valid syntactic forms. An arbitrary document such as {$gt:””} would not be a valid syntactic form w.r.t U because it cannot be generated by the policy.

Command Injection Attack

An input vector is considered a command injection attack if it produces a query that can be generated by a grammar G that describes NoSQL and there exists a filtered input that is not a valid syntactic form w.r.t our policy U.

B. *Algorithm*

Input tracking

Wasserman and Su describe tracking the input through the use of metacharacters ☾ ☽. This can be done as follows: in the final stage, before the input is used to construct the query, a string is constructed as follows “☾” + JSON.stringify(input) + “☽”. The stringify operation is needed in order to escape quoted characters and to expand potential JSON objects (e.g. {$gt : “”}). Once the query is ready to be checked by NoSQLCheck, we need to remove the quotes preceding or following the metacharacters and unescape the strings that were escaped by our JSON.stringify function. These steps are necessary because parsers only accept input that is fed as a string.

Grammar Construction

NoSQL grammar construction is more involved due to the large variety of inputs and the inherent recursive structure of MongoDB’s document syntax. We outline the key aspects of the grammar here.

```
object -> "{" _ "}" {% function(d) { return {}; } %}  |
          "{" _ pair (_ "," _ pair):* _ "}"
```

(Note that _ refers to an arbitrary amount of white space)

A document object can be the empty object ,{}, or consist of multiple pairs.

```
pair -> field_to_value | operator_to_value
```

A pair is a key value pair, where a key is either a regular field in a document or a MongoDB operator. A value can be any language primitive (strings, numbers, booleans, null values) or more complicated values such as an object or array. Notice that we are referring to object, which is a high-level non-terminal in our grammar; this gives us our recursive document structure.

```
operator_to_value -> comparison_op | logical_op |
    element_op | eval_op | array_op | comment
```

The operator_to_value can be divided into 6 main groups. This is how they are delineated in MongoDB’s reference manual and as such we have decided to do the same. The values for each operator to value pair are determined by the reference for each one of MongoDB’s keywords [7]. In places where the value can be determined to be a language primitive, the value is assigned as such. In places where the value is ambiguous or underspecified in the reference, we simply assign it to be a value or an array.

```
element_op -> element_op_exists | element_op_type
element_op_exists -> ("$exists" | quote "$exists" quote |
    quote2 "$exists" quote2) _ ":" _ boolean
element_op_type -> ("$type" | quote "$type" quote | quote2
    "$type" quote2) _ ":" _ (num | array_of_string |
    array_of_num | string)
```

To augment our grammar to include our policy we perform the following transformation as described in Wasserman and Su’s paper:

For every non-terminal in the policy we construct a new production rule as follows:

```
term_a -> term | ☾ term ☽
```

Example:

```
num_a -> num | ☾ num ☽
```

We then replace every reference to term with term_a excluding, of course, the production rule for the term.

Parser generation

To generate the parser, we use the Earley parsing algorithm. This algorithm is capable of generating a parser for all context-free languages.

The complete algorithm is as follows:

1. Generate an augmented grammar using a policy U
2. Generate a parser for this grammar using the Earley parsing algorithm
3. Augment our query input with metacharacters to encapsulate user input. Construct an augmented query from the query input and the original query.
4. Use the generated parser to parse the augmented query. If it parses successfully, the original query is benign, otherwise, it is potentially malicious.

IV. Implementation

The context-free grammar is implemented using nearley.js, a popular context-free parser generator. We implemented our NoSQLCheck algorithm in Node.js. The package requires a global install of nearley since parsers need to be dynamically generated based on the user policy specified in the NoSQLCheck constructor. The metacharacters are created using Node.Js's secure random function, crypto.randomBytes(n). We chose 8 bytes for our metacharacters which are then encoded into base 58 (i.e. alphanumeric characters only). Usage of the NoSQLCheck package is extremely simple as illustrated in the code example below:

```
const nosqlcheck = require('./nosqlcheck.js');
const checker = new
nosqlcheck(["id","key","string","num","boolean"]);

app.post('/login', function(req, res) {

  var username = req.body.username;
  var password = req.body.password;

  var original_query = {"username" : username,
  "password": password};
  var augmented_query = {"username" :
  checker.trackInput(username),
 "password": checker.trackInput(password)};

  var is_benign = checker.checkQuery(augmented_query);

  if(is_benign) {
    console.log("Benign Query");
    User.find(original_query).then(function(value) {
      res.send({
        "is_benign": true,
        "login": value
      })
    })
  } else {
    console.log("Potentially Malicious Query");
    res.send({
      "is_benign": false
    })
  }

});
```

Fig. 3. Using the NoSQLCheck Package

All the programmer needs to do is require the package, call the constructor with a policy, call checker.trackInput() on user input in the stage just before calling checker.checkQuery() on the augmented query. In our tests, NoSQLCheck introduced an overhead of about ms for every query.

V. Results

We generated attack vectors by creating operator_to_value expressions that could cause data leakage or unwarranted access to user data. This was done by looking at popular NoSQL attack patterns across various websites and constructing queries of that form [4].

TABLE I. NoSQL Detection Rate

Detection Metrics		
Attack Vectors (Prevented/Total)	***Benign Vectors (Allowed/ Total)***	***Additional Response Time***
92	1000/1000	~3ms

Fig. 4. Effectiveness of NoSQLCheck in detecting attack vectors

VI. Limitations

Due to the underspecified reference for argument types in queries involving MongoDB keywords, we had to use more generic value types so as not to disallow benign queries. We also make no attempt to check if the arguments of the $where query contain malicious input. This is because it can be completely arbitrary code. Therefore, if your query utilizes a $where operator, it is paramount that it only contains developer code and no user input. Our approach is tailored to detecting malicious queries; therefore, it does not work on the aggregation portion of MongoDB's pipeline. Lastly, we use JSON.stringify() to serialize input from the user; this is fine with one caveat, arbitrary function code is removed. In order to rectify this, we would need to first convert the function code into a string.

VII. Future Work

We would like to verify our method on a larger source of attack and benign vectors to ensure its robustness. We would also like to develop a method for detecting malicious injection in queries involving the $where operator. This can most likely be accomplished by constructing a context-free grammar for Javascript.

VIII. References

[1] "Top 10-2017 Top 10," *OWASP*. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10. [Accessed: 04-Nov-2019].

[2] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” ACM SIGPLAN Notices, vol. 41, no. 1, pp. 372–382, Dec. 2006.

[3] “Stack Overflow Developer Survey 2019,” Stack Overflow. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools. [Accessed: 04-Nov-2019].K. Elissa, “Title of paper if known,” unpublished.

[4] Swisskyrepo, “swisskyrepo/PayloadsAllTheThings,” GitHub, 30-Oct-2019.[Online]. Available: https://github.com/swisskyrepo/Payloads AllTheThings/tree/master/NoSQL Injection. [Accessed: 04-Dec-2019].

[5] “Testing for NoSQL injection,” OWASP. [Online]. Available: https://www.owasp.org/index.php/Testing_for_NoSQL_injection. [Accessed: 14-Nov-2019].M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[6] P. Tarau, “Earley Parser.” [Online]. Available: http://www.cse.unt.edu/~tarau/teaching/NLP/Earley parser.pdf. [Accessed: 11-Dec-2019].

[7] “Operators,” *Operators - MongoDB Manual*. [Online]. Available: https://docs.mongodb.com/manual/reference/operator/. [Accessed: 04-Dec-2019].