Application of Graph Databases for Static Code Analysis of Web-Applications

Daniil Sadyrin [0000-0001-5002-3639], Andrey Dergachev [0000-0002-1754-7120], Ivan Loginov [0000-0002-6254-6098], Iurii Korenkov [0000-0002-8948-2776], and Aglaya Ilina [0000-0003-1866-7914]

ITMO University, Kronverkskiy prospekt, 49, St. Petersburg, 197101, Russia
dssadyrin@itmo.ru, dam600@gmail.com, ivan.p.loginov@gmail.com, ged.yuko@gmail.com, agilina@itmo.ru

Abstract. Graph databases offer a very flexible data model. We present the approach of static code analysis using graph databases. The main stage of the analysis algorithm is the construction of ASG (Abstract Source Graph), which represents relationships between AST (Abstract Syntax Tree) nodes. The ASG is saved to a graph database (like Neo4j) and queries to the database are made to get code properties for analysis. The approach is applied to detect and exploit Object Injection vulnerability in PHP web-applications. This vulnerability occurs when unsanitized user data enters PHP unserialize function. Successful exploitation of this vulnerability means building of "object chain": a nested object, in the process of deserializing of it, a sequence of methods is being called leading to dangerous function call. In time of deserializing, some "magic" PHP methods (___wakeup or ___destruct) are called on the object. To create the "object chain", it's necessary to analyze methods of classes declared in web-application, and find sequence of methods called from "magic" methods. The main idea of author's approach is to save relationships between methods and functions in graph database and use queries to the database on Cypher language to find appropriate method calls. Also, some unobvious ways of calling other PHP "magic" methods, which help to find more appropriate "object chains" are considered. The approach was successfully tested on the vulnerability CVE-2014-1860 discovered in Contao CMS.

Keywords: static analysis · graph database · Cypher · PHP · Object injection

1 Introduction

Graph databases are used in many areas like bioinformatics [1], social networks [2], chemistry [3], and static code analysis [4]. A graph database (over a countably infinite set of labels Σ) is a pair $G = (V, E)$ where V is a finite set of nodes,

$E \subseteq V \times \Sigma \times V$ is a finite set of edges. There are two basic ways to explore and graphically depict connected data: Resource Description Framework (RDF) triple stores and labeled property graphs.

The abstract RDF syntax is a set of triplets called an RDF graph. An RDF triplet contains three components: a subject that is an URI reference or an empty node; predicate, which is an URI reference; an object that is an URI reference, literal, or empty node. RDF triplet written in order: subject, predicate and object. A predicate is also known as a triplet property. An RDF graph is a set of RDF triplets. A set of RDF graph nodes is a set of subjects and objects of triplets of a graph.

The property graphs are graphs in which attributes (properties) are assigned to edges and/or vertices of the graph. A database in a graph model is a graph whose vertices and edges are typed. A vertex or edge type is a collection of attributes (properties) attributed to a vertex or edge.

2 Graph search queries in terms of formal languages

One of the major problems, related with graph databases, is to find specific paths in it. A path in graph G is a sequence $p = (v_0, a_1, v_1)...(v_{n-1}, a_n, v_n)$ of edges of G. Constraints on the paths can be expressed in several ways: conjuctive queries, shortest path queries, in terms of formal languages.

Constraints in terms of regular languages are used to search patterns in graph. The regular expressions for an alphabet Σ are defined by the following form: $E ::= \emptyset \mid \varepsilon \mid a \mid (E \circ E) | (E + E) | E^*$, where $a \in \Sigma$. If E is a regular expression then $L(E)$ is a regular language. A regular path query (RPQ) is an expression of the form $x \xrightarrow{r} y$ where x and y are variables and r is regular expression over Σ. Let r be a regular expression and G be a graph. A path $p = (v_0, a_1, v_1)...(v_{n-1}, a_n, v_n)$ in G matches r, if $a_1 a_2 ... a_n \in L(r)$.

Context-free languages also can be used as constraints. A context-free grammar G can be defined as 4-tuple: $G = (V, T, P, S)$ where V is a finite set of nonterminals containing S, T is finite set of terminals, P is a set of production rules in the form of $\alpha \to \beta$ where $\alpha \in V$ and $\beta \in (V \cup T)^*$, and S is start symbol. An answer to a context-free path query (CFPQ) is usually a set of triples (A, m, n) such that there is a path from the node m to the node n, whose labeling is derived from a non-terminal $A \in V$ of the given context-free grammar.

For example, let's consider the following context-free grammar: $S \to aSa$ and $S \to bSb$. This grammar generates the language of even-length palindromes.

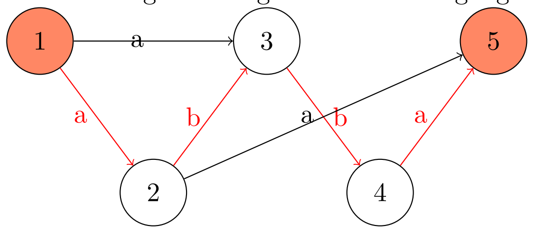

Fig. 1: Graph with path accepted by language L = $\{ww^R, w \in \{a, b\}^*\}$

Context-free path queries are more expressive, than regular path queries, but such type of queries is not real-time and requires large amounts of memory [12]. Another way to increase the expressiveness of regular parh queries is conjunctive regular path queries. A conjunctive regular path query (CRPQ) is an expression of the form $\exists \overline{z}\ ((x_1 \xrightarrow{a_1} y_1) \wedge ... \wedge (x_n \xrightarrow{a_n} y_n))$ where $\overline{z}$ is a tuple of variables from $\{x_1, ..., x_n, y_1, ..., y_n\}$ and r_i is an RPQ over Σ for i $\in [n]$. A conjunctive query over graph is an expression of the form: $\exists \overline{z}\ ((x_1 \xrightarrow{a_1} y_1) \wedge ... \wedge (x_n \xrightarrow{a_n} y_n))$ where $\overline{z}$ is a tuple of variables from $\{x_1, ..., x_n, y_1, ..., y_n\}$ and $\{a_1, ..a_n\} \in \Sigma$. Conjunctive queries (and even first-order queries) on graphs are limited, they can only express ”local” properties.

3 Path querying algorithms

Most existing graph querying languages, including SPARQL [5], Gremlin [6], Cypher support only regular languages as constrains. There are several approaches for evaluating RPQ: approach based on mapping to finite automaton [7] and on searching for rare labels and starting breadth-first search [8]. Algorithms for evaluating conjunctive regular path queries are studied in [9], [10]. cfSPARQL [11] is the single known graph query language to support context-free path constraints. The most of context-free path query evaluation algorithms are based on extending the known context-free parse techniques to the graph input. GSSLR algorithm [13] is based on Tomita recognizer, subgraph queries algorithm [14] uses Early parser, context-free path querying with structural representation of result [15] is based on generalized top-down parsing algorithm (GLL). Also, algorithm, introduced in [17], constructs annotated grammar in order to evaluate context-free path query. Algorithm in [16] is desighned to use fast boolean matrix multiplication and GPU.

4 Graph databases in static code analysis

One of the important usages of graph data models is a static code analysis. Static analysis is a proven approach for detecting mistakes in the source code early in the development cycle. Since static analysis does not compile or run the code, it can be applied at an early state of development. This section investigates the usage of graph databases in static code analysis tools.
Graph-based analysis of JavaScript source code repositories [18] detects deadcode, potential division by zero, and other mistakes using Neo4j graph databases and openCypher for evaluating regular path queries.
GREENSPECTOR [21] uses Neo4j and Cypher query language for finding coding mistakes in a special graph data model called “call graph” via pattern-matching.
Wiggle [22] is a prototype graph-model code-query system. It performs such kinds of analysis like exploring type hierarchy, override hierarchy, type attribution, method call graph and data flow analysis.

Class-Graph [23] uses Neo4j and Cypher query language to collect structural insights about Java projects and to store, compute and visualize a variety of software metrics and other types of software analytics (method call hierarchies, transitive clojure, critical path analysis, volatility and code quality).
Paper [24] presents an approach to detect behavioral design patterns from source code using static analysis techniques. This approach used Neo4j and uses graph query language Gremlin for doing graph matching to perform structural analysis, behavioral analysis, semantic analysis, Program Dependence Analysis, Control Dependence Analysis and Data Dependence Analysis.
jQAssistant [25] is a static code analysis tool using the graph database Neo4j and Cypher query language. It is used for detection of constraint violations, generating reports about user defined concepts and metrics, deetecting common problems like cyclic dependencies or tests without assertions in Java projects. jQAssistant allows definition of rules and automated verification during a build process. Rules can be expressed as Cypher queries.
Joern [26] analyzes a code base using a robust parser for C/C++ and represents the entire code base by one large property graph stored in a Neo4j graph database. This allows code to be mined using complex queries formulated in the graph traversal languages Gremlin and Cypher. Exploring program structure, call graph, data flows, methods, types and other can be performed by Joern tool.
NAVEX [27] combines dynamic analysis that is guided by static analysis techniques in order to automatically identify vulnerabilities and build working exploits. It uses extention of Joern for PHP language to find vulnerabilities via searching the enhanced code property graph using Gremlin queries and Neo4j graph database.

5 Static code analysis problem

To extract information for static analysis, it is necessary to present the program code in the form of AST (Abstract Syntax Tree). Next, the syntax tree is translated into a graph representation - ASG (Abstract Semantic Graph), for this it is necessary to complete the information from AST with semantic information. The resulting graph structure that accomodates the information, would be a direct representation of packages, classes, interfaces, types, methods, fields and containing relationships like dependencies, containment, calls, coverage, etc. Queries are made to the constructed graph to retrieve the necessary code properties, for example, obtain functionDefinition relationships to get a program call stack.

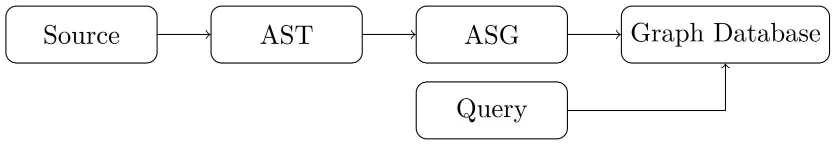

Fig. 2: Flowchart of static analysis engine

6 Proposed Approach

It is proposed to apply this approach for searching and exploiting of the Object Injection vulnerability in PHP web applications. Object Injection attack is part of the OWASP [19] vulnerability classification. The vulnerability arises when unsanitized user data enters the PHP unserialize function. The result of exploiting the vulnerability is to build a string with a chain of serialized PHP objects. In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

- The web-application must have a class which implements a PHP magic method (such as __wakeup or __destruct) that can be used to carry out malicious attack
- All of the classes used during the attack must be declared when the unserialize() function is being called

To exploit this vulnerability, we need to do static analysis of declared classes in web-application code. During static analysis, we build a class hierarchy based on the inheritance of each class. All defined methods, properties and class constants are transformed to data symbols and stored in the analysis environment. When analyzing the code of declared methods, it is necessary to take into account these object-oriented features in PHP [20]:

- Object-sensitive Methods: __set_state(), __sleep(), __invoke(), __clone(), __toString(), __construct()
- Field-sensitive Methods: __get, __set, __isset, __unset
- Invocation-sensitive Methods: __call, __callStatic
- Calls using keywords: parent::, self::, static::

Let's consider an example of the class with __wakeup method, which has ”new” operator in it's code. __construct method of Vuln class is called and arbitrary file is unlinked.

```
class Vuln {
        public function __construct($file) {
                unlink($file);
        }
}
class test {
        public function __wakeup() {
                $this->a = new Vuln($this->test);
        }
}
unserialize('O:4:”test”:1:{s:4:”test”;s:13:”/tmp/test.php”;}');
```

In the following example non-existent method call in ___destruct leads to ___call method being executed:

```
class method_test {
    public function ___call($name, $arguments) {
        echo "call '$name' " . implode(', ', $arguments). PHP_EOL;
                unlink($this->file);
    }
    public function ___destruct() {
            $this->notexisting(1,2,3);
    }
}
unserialize('O:11:"method_test":1:{s:4:"file";s:13:"/tmp/test.php"}');
```

For each method, it is performed a check for possible other method calls and "dangerous" functions. Using this information we can create object injection chains. It is necessary to take into account some unobvious ways to call PHP "magic" methods, when analyzing web-application source codes. It helps to find additional relationships between method calls and build proper "object chain". No one of the previously reviewed open-source projects solves the task of searching PHP object injection vulnerability.

7 Implementation

Constructing AST from source codes is done using nikic's PHP-Parser utility [28]. Each AST node is an object of class representing this node. Obtaining relationships is done by calling "traverse" method of an object of NodeTraverser class, declared in PHP-Parser.

```
$nodeTraverser = new PhpParser\NodeTraverser;
$nodeTraverser->addVisitor(new ChangeMethodNameNodeVisitor);
$traversedNodes = $nodeTraverser->traverse($nodes);
```

All constructed nodes are bypassed, and there is done a check whether the node is the object of some class:

- Class_ - represents AST of whole declared class.
- ClassMethod - represents AST of class method code.
- MethodCall - represents AST corresponding to method call inside other methods.
- FuncCall - represents AST corresponding to function call inside other methods.

Information is extracted from nodes, saved into CSV files and imported into Neo4j database. Methods and functions are represented using nodes labeled "Method" and "Function" and linked by relationship named "CALLS". Nodes labeled "Method" have properties: name - method name, class_name - name of the class where method is declared. ”Function” nodes have the following properties: name - function name, vuln (indicates that it is ”dangerous” function, True or False). ”CALLS” relationship stores class field that calls the method in it’s property. Relationships between method and function calls are created by the following queries written on Cypher language:

```
MATCH (n1:Method),(n2:Method) WHERE n1.class_name=line[0]
AND n1.name=line[1] AND n2.name=line[3]
MERGE (n1)-[r:CALLS {property:line[2]}]->(n2)''')
```

```
MATCH (n1:Method),(n2:Function) WHERE n1.class_name=line[0]
AND n1.name=line[1] AND n2.name=line[2]
MERGE (n1)-[r:CALLS {property:''}]->(n2)''')
```

To obtain a sequence of method calls, we execute a query to the database:

```
MATCH p = (a: Method) - [r: CALLS * 0..10] -> (b: Function {vuln: True})
WHERE a.name IN ['___wakeup', '___destruct'] RETURN p
```

This query returns all paths in graph database from nodes representing method with the ”magic” name (___destruct / ___wakeup) to nodes representing dangerous function (marked with property ”vuln” equal to ”True”).

For demonstrating the approach, we took Contao CMS with vulnerability CVE-2014-1860 [29] and obtained a sequence of method calls leading to PHP unlink function being called with an arbitrary file name as an argument.

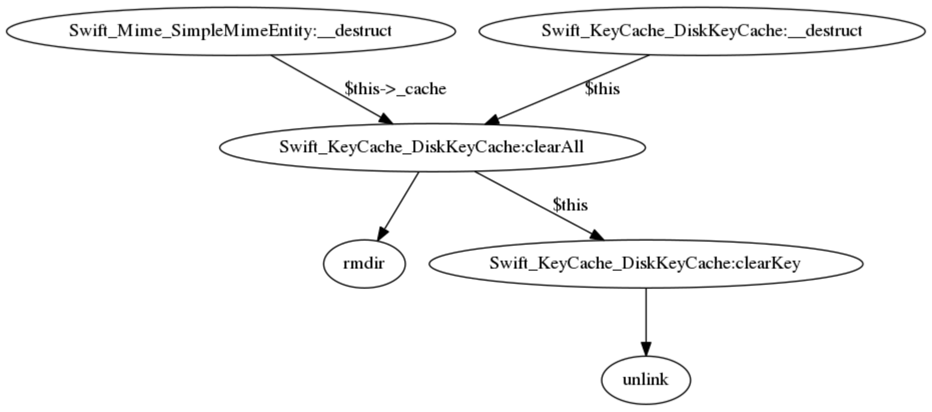
Swift_Mime_SimpleMimeEntity:__destruct
Swift_KeyCache_DiskKeyCache:__destruct
$this->_cache
$this
Swift_KeyCache_DiskKeyCache:clearAll
$this
rmdir
Swift_KeyCache_DiskKeyCache:clearKey
unlink

Fig. 3: CVE-2014-1860 methods call graph

8 Conclusion

Summing up, we propose method for static code analysis of scripts written in PHP programming language. We create graph database that stores relationships between code properties. Using this information creation of object injection chains is done. Further work may consist in applying an approach to searching deserialization vulnerabilities in application code with frameworks in Java (Weblogic, Tomcat, Spring) or .NET. (Nancy, Breeze), and reducing time complexity of graph database queries using context-free path queries.

References

1. Fiannaca A. et al. BioGraphDB: a new GraphDB collecting heterogeneous data for bioinformatics analysis //Proceedings of BIOTECHNO. – 2016.
2. Cattuto C. et al. Time-varying social networks in a graph database: a Neo4j use case //First international workshop on graph data management experiences and systems. – ACM, 2013. – C. 11.
3. Hall R. J., Murray C. W., Verdonk M. L. The Fragment Network: A Chemistry Recommendation Engine Built Using a Graph Database //Journal of medicinal chemistry. – 2017. – T. 60.
4. Yamaguchi F. et al. Modeling and discovering vulnerabilities with code property graphs //2014 IEEE Symposium on Security and Privacy. – IEEE, 2014. – C. 590-604.
5. Prud E. et al. SPARQL query language for RDF.(2006). – 2006.
6. Rodriguez M. A. The Gremlin graph traversal machine and language (invited talk) //Proceedings of the 15th Symposium on Database Programming Languages. - ACM, 2015. - C. 1-10.
7. Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. SIAM J. Comput. 24, 6 (December 1995), 1235-1258. DOI=http://dx.doi.org/10.1137/S009753979122370X
8. Koschmieder, Andre and Leser, Ulf. (2012). Regular Path Queries on Large Graphs. 7338. 10.1007/978-3-642-31235-9_12.
9. Pablo Barceló Baeza. 2013. Querying graph databases. In Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems (PODS '13). ACM, New York, NY, USA, 175-188. DOI: https://doi.org/10.1145/2463664.2465216
10. Bienvenu, Meghyn and Ortiz, Magdalena and Šimkus, Mantas. (2013). Conjunctive Regular Path Queries in Lightweight Description Logics. IJCAI International Joint Conference on Artificial Intelligence. 761-767.
11. Zhang X. et al. Context-free path queries on RDF graphs //International Semantic Web Conference. - Springer, Cham, 2016. - C. 632-648.
12. Kuijpers J. et al. An experimental study of context-free path query evaluation methods //Proceedings of the 31st International Conference on Scientific and Statistical Database Management. – ACM, 2019. – C. 121-132.
13. Medeiros, Ciro and Musicante, Martin and Costa, Umberto. (2019). LL-based query answering over RDF databases. Journal of Computer Languages. 51. 75-87. 10.1016/j.cola.2019.02.002.
14. Sevon, Petteri and Eronen, Lauri. (2008). Subgraph Queries by Context-free Grammars. Journal of Integrative Bioinformatics. 5. 10.1515/jib-2008-100.
15. Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free path querying with structural representation of result. In Proceedings of the 13th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '17). ACM, New York, NY, USA, Article 10, 7 pages. DOI: https://doi.org/10.1145/3166094.3166104.

16. Rustam Azimov and Semyon Grigorev. 2018. Context-free path querying by matrix multiplication. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18), Akhil Arora, Arnab Bhattacharya, George Fletcher, Josep Lluis Larriba Pey, Shourya Roy, and Robert West (Eds.). ACM, New York, NY, USA, Article 5, 10 pages. DOI: https://doi.org/10.1145/3210259.3210264
17. Hellings, J. (2015). Querying for Paths in Graphs using Context-Free Path Queries.
18. Szárnyas, Gábor. Graph-based analysis of JavaScript source code repositories, FOSDEM, Graph devroom (Brussels, 2018)
19. OWASP T. Top 10-2017 The Ten Most Critical Web Application Security Risks
20. Azis I. M. F., Kom M. Object Oriented Programming Php 5. – Elex Media Komputindo, 2005.
21. GREENSPECTOR tool. Available: https://greenspector.com/en/articles/2017-06-12-analyse-statique-code-bdd-orientee-graphe/
22. Urma, Raoul-Gabriel and Mycroft, Alan. (2015). Source-code queries with graph databases - With application to programming language usage and evolution. Science of Computer Programming. 97. 10.1016/j.scico.2013.11.010.
23. Michael Hunger: Class-Graph, leverages Cypher to collect structural insights about your Java projects Available: https://github.com/jexp/class-graph
24. Abdelsalam, Khaled and Kamel, Amr. (2018). Reverse Engineering State and Strategy Design Patterns using Static Code Analysis. International Journal of Advanced Computer Science and Applications. 9. 10.14569/IJACSA.2018.090178.
25. jQAssistant tool. Available: https://jqassistant.org
26. Available: https://github.com/ShiftLeftSecurity/joern
27. Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. 2018. NAVEX: precise and scalable exploit generation for dynamic web applications. In Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18). USENIX Association, Berkeley, CA, USA, 377-392.
28. A PHP parser written in PHP. Available: https://github.com/nikic/PHP-Parser
29. CVE-2014-1860. Available: https://github.com/contao/core/pull/6730