Detection of remote code execution vulnerabilities using abstract interpretation

Roman Kholin
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Leninskie Gory, Moscow, 119991, Russian Federation
romankholin94@seclab.cs.msu.ru

Dennis Gamayunov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Leninskie Gory, Moscow, 119991, Russian Federation
gamajun@seclab.cs.msu.su

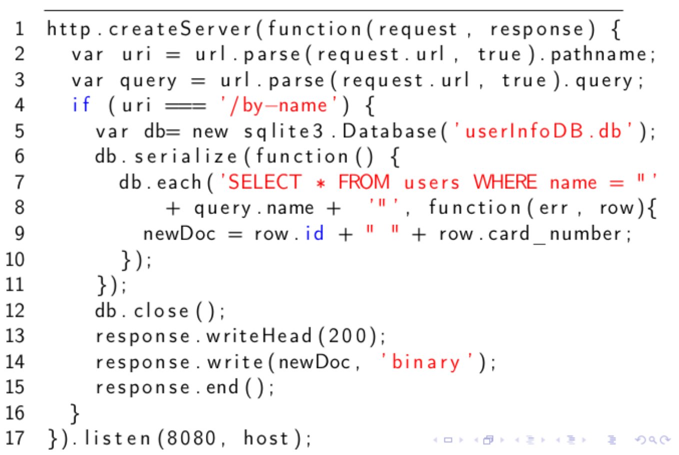
***Abstract*—“Remote code execution” attacks are among the most dangerous types of attacks, they can lead to the disclosure of confidential data, unauthorized deletion of data, device control interception and other vulnerabilities, so it is important to be able to find vulnerabilities in programs as quick as possible along the program’s life cycle. The vulnerabilities search is time-consuming, so there is a serious scientific and industrial effort to automate the process as much as possible.**
We propose a method for generalized code injection vulnerabilities detection for a wide class of injections with help of symbolic execution (one of the methods of abstract interpretation).

***Index Terms*—concolic testing, JavaScript, SMT, web applications security.**

I. Introduction

Code injection is the exploitation of a computer bug that is caused by processing invalid user input. The reason of possibility of the remote code execution is that there are functions in the program that may execute code in some programming language (for example, “eval” function in JavaScript, “echo” function in the php or the function for executing database query “mysql_query” in the php), and this code can be generated dynamically so user input can get into it, i.e. a “injection” into code via user input can occur. SQL injection is “injection” into the code that infiltrates the input of the “query to the database” functions (for example, the input “mysql_query” in PHP); code injection is an “injection” into the code that tamper the input of the function “eval” (“eval” mean to “execute” code; there are such functions, for example, in JavaScript and PHP programming languages). In this paper functions like mentioned above are called “sensitive functions”, and the argument they take will call “request to sensitive function”.

“Fig. 1” shows vulnerable JavaScript backend code for the web application. Here, db.each is a sensitive function, ’SELECT * FROM users WHERE name = ”’ + query.name + ’”’ is a request to a sensitive function, and the input data gets into the query via query.name. We can make such a query that query.name becomes equal to ’OR’ a’=’a and after that information about all rows of the table will be returned, which was not explicitly provided by the service logic.
We propose to generalize the problem of detecting code injection vulnerabilities for a wider class of injections and to search them using one of the methods of abstract interpretation — symbolic execution [1], [2]. To do this, we will search sensitive functions in programs and what input data to submit to the input of the program (in the case of a web application, what request to do to the web application), so that the program run with this input data also reaches the given sensitive function and executes it. In this work, a prototype implementation for various database drivers and NodeJS was made. To implement a prototype solution to the problem, we modified the JavaScript test case generation tool “ExpoSE” [3], [4] using the JavaScript code instrumentation tool “Jalangi2” [5]–[7]. With the help of concolic execution, it is possible to traverse all the paths of program execution, if the number of such paths is finite; if the number of such paths is infinite, then it is possible to traverse a significant part of them. Passing through the branch statements, we collect the constraints on the input data using concolic execution. Using these constraints and SMT solvers (eg “Z3” [8]), it is possible to generate attack input data that satisfies new, not yet passed program execution paths. Having reached the sensitive function, we can generate a query that leads us to this point in the program.



```
http.createServer(function(request, response) {
  var uri = url.parse(request.url, true).pathname;
  var query = url.parse(request.url, true).query;
  if (uri === '/by—name') {
    var db= new sqlite3.Database('userInfoDB.db');
    db.serialize(function() {
      db.each('SELECT * FROM users WHERE name = "'
          + query.name + '"', function(err, row){
        newDoc = row.id + " " + row.card_number;
      });
    });
    db.close();
    response.writeHead(200);
    response.write(newDoc, 'binary');
    response.end();
  }
}).listen(8080, host);
```

Fig. 1. Vulnurable to sql-injection code on JavaScript

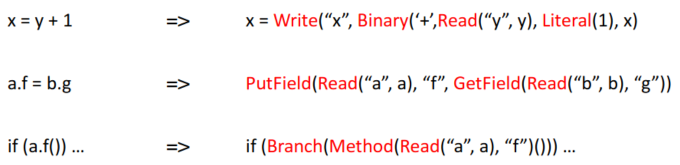
The structure of this article is as follows: the Overview provides a description of the technologies and the standard techniques that we use here; in Implementation we present the algorithm of our prototype solution; in Evaluation, we describe stand applications and the results of working of our tool on them; in the Related Work subsection, we present similar frameworks and techniques that we have found; in Future work we describe ideas for improving our tool; in Conclusion we summarize the results of the study.

II. Overview

In this section we briefly describe the frameworks and technologies used in the paper.

A. Jalangi2

Jalangi2 is a framework for writing dynamic analyses for JavaScript. Jalangi2 instruments the program-under-analysis to insert callbacks to methods defined in Jalangi2. An analysis writer implements these methods to perform custom dynamic program analysis. Jalangi2 performs analysis during the execution of the program. "Fig. 2" contain an example of the code after instrumentation. Write, Read, Binary, Literal, PutField, GetField, Branch, Methods - the name of the callbacks that can be implemented in your analysis (this is not a complete list).



```
x = y + 1        =>      x = Write("x", Binary('+',Read("y", y), Literal(1), x)

a.f = b.g        =>      PutField(Read("a", a), "f", GetField(Read("b", b), "g"))

if (a.f()) ...   =>      if (Branch(Method(Read("a", a), "f")())) ...
```

Fig. 2. Simplified demonstration of Jalangi2 instrumentation of code

B. Z3

Z3 is Theorem Prover, a cross-platform satisfiability modulo theories (SMT) solver. Satisfiability modulo theories (SMT) is the problem of determining whether a mathematical formula is satisfiable. It generalizes the Boolean satisfiability problem (SAT) to more complex formulas involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings. Formulas may not have a decision procedure, or the task of formula satisfaction may be NP-hard, but in practice, solvers perform well due to the heuristics that are implemented in their algorithms. One of the most important achievements of modern SMT solvers is their ability to resolve formulas over string data types - i.e. simulate a lot of primitive operations on string variables (concatenation, slicing, inserting a substring into a string, etc.) and find specific examples of strings that satisfy a set of logical formulas on variables that use these operations. Moreover, the most advanced versions of modern SMT solvers also support checking whether a certain word belongs to a given language described by a regular grammar, among the operations available in the SMT solver.

C. Concolic Execution

Concolic Execution (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path. Symbolic execution is used in conjunction with an automated theorem prover or constraint solver based on constraint logic programming to generate new concrete inputs (test cases) with the aim of maximizing code coverage. Initially, symbolic (and concolic) tools were developed only for compiled programming languages (like "DART" [9], "CUTE" [10], "KLEE" [11] for C languge), but over time they added support for interpreted programming languages.

The program can be represented as a binary tree - the so-called computational tree (see "Fig. 3"). Each vertex is the execution of a conditional statement, each edge is the execution of a sequence of commands that are not a conditional statement, each path from the root divides the set of input data into equivalence classes. One of the goals of concolic execution is to provide one instance for each such equivalence class, i.e. generate a test - the input data on which the program will go through a unique path from the root to the leaf.

The concolic execution algorithm can be described as

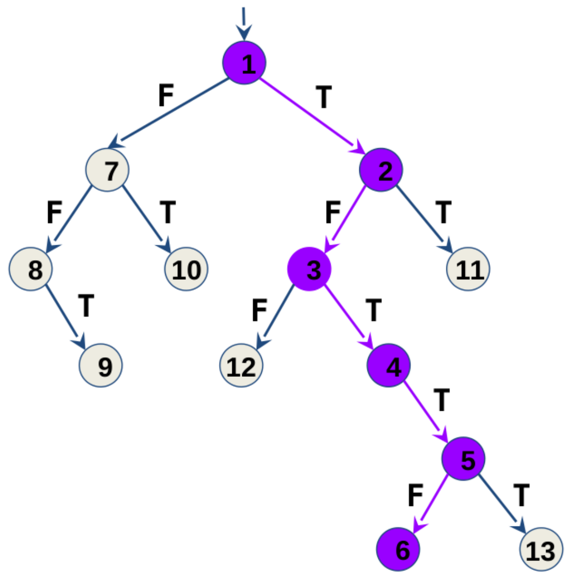


Fig. 3. Example of a figure caption.

follows:
At the beginning, a random data set is generated and added to the queue. Then, until the queue becomes empty, the input data set is popped from the queue, the program is executed on this input data set, the execution trace is recorded, fixing all passed branch points in the program and all operations

written on the program variables along this path. In this case, specific operations are executed on concrete values of variables, and symbolic ones are executed on symbolic variables. This allows for each specific path to describe the constraints on the variables (path constraints), which lead to the execution of the program along a specific path. In this case, the constraints have the format of a logical formula, which uses the logical operations "and", "or" and a set of some of the simplest operations on variables: comparisons, simple arithmetic operations, simple operations on strings (concatenation, slicing, deletion whitespace characters on the left and right, etc.).

When passing a branch point, we try to build an alternative path for executing the program:

1) we consider a set of path constraints up to a given branch point, select the part of the formula that describes the choice of the execution path at a given point. If we have already tried to generate a set of input data for such a program execution path, then we finish building an alternative program execution path; otherwise we continue
2) We build a new set of constraints on variables, using the negation of the last condition;
3) we resolve this set of constraints on variables by finding a set of specific values of symbolic variables that satisfies all the conditions of path constraints;
4) add the resulting set of input data to the queue.

The most important and critical part of this process, however, is the ability to effectively model operations on symbolic variables (reducing their effort to a set of simple atomic operations) and resolve the resulting sets of logical formulas (path constraints) for all types of variables used in the program. The main method for such resolution of formulas in the field of dynamic program analysis is currently the Satisfiability Modulo Theory, which was described above.

D. ExpoSE

ExpoSE is a dynamic symbolic execution engine for Node.js applications. ExpoSE automatically generates test cases to find bugs and cover as many paths in the target program as possible. ExpoSE bypasses all program execution paths by using a concolic execution technique, implementing callbacks in Jalangi2, and using Z3 as a solver to generate new input data.

III. Implementation

At this stage of development, our prototype solution is looking for a hotspot using the concolic execution technique implemented by the ExpoSE tool. We find a call of the crateServer function of the nodejs http standard library [12]. This function takes as input the function f(request, response) - a request processing function, where request is an object of the http.IncomingMessage class that describes the request, response - an object of the http.ServerResponse class that describes the server response (for more details, you can read in the documentation here). Instead of executing the crateServer function, we run the function f(request', response'), where request' and response' are objects that model a server request and a server response. The main thing in request' is the concolic "url" and "methods" variables that model the input. Since the analysis is performed dynamically, before calling the next function in the program, we can check whether it is a hotspot. If it is really a hotspot, then we can restore the request, which, by sending it to the server, we will get to a given point in the program with such a state of the variables. By passing this url to sqlmap [13], we can verify whether it is possible to transfer such data to this hotspot that they lead to a vulnerability.

IV. Evaluation

To evaluate the prototype, several stand web applications were made. When creating them, we pursued the following goals: they should have sql-injection, code-injection; stands must use different database drivers; stands should be written on different web-frameworks. We got the following stands:

1. The application keeps a list of users of some site. The application has a main page, as well as 3 more that are linked from the main page: /users - a list of users, /by-id - a page with information about a user that has a specific id parameter, /by-name - a page with information about user having a particular concrete name parameter. When following the /by-name link, the name parameter is passed, which is processed on the server in a vulnerable way - without any checks, it is part of the SQL query that searches the database for the desired user. Because of this, any information can be extracted from the database, which leads to a SQL injection vulnerability. When following the /by-id link, the id parameter is passed, which is processed on the server in a vulnerable way - without any checks, it is input to the eval command, which allows remote code execution on the server and leads to a code-injection vulnerability. The application is written based on the built-in framework, as well as the third-party mysql library, which is driver and designed to work with the mysql DBMS.

2. The application keeps a list of users of some site. The application has a main page, as well as another one, which is linked from the main page: /users - a page with information about the user, which has a specific username parameter. When following the /users link, the username parameter is passed, which is processed on the server in a vulnerable way - without any checks, it is part of the SQL query that searches the database for the desired user. Because of this, any information can be extracted from the database, which leads to a SQL injection vulnerability. The application is written based on the built-in framework, the framework for creating express web applications, as well as the third-party sqlite3 library, which is driver and designed to work with the sqlite DBMS.

V. Related Work

In this review, we focused on dynamic methods of program analysis and vulnerability search.
Wasserman et al. [14] formally defines what a vulnerability

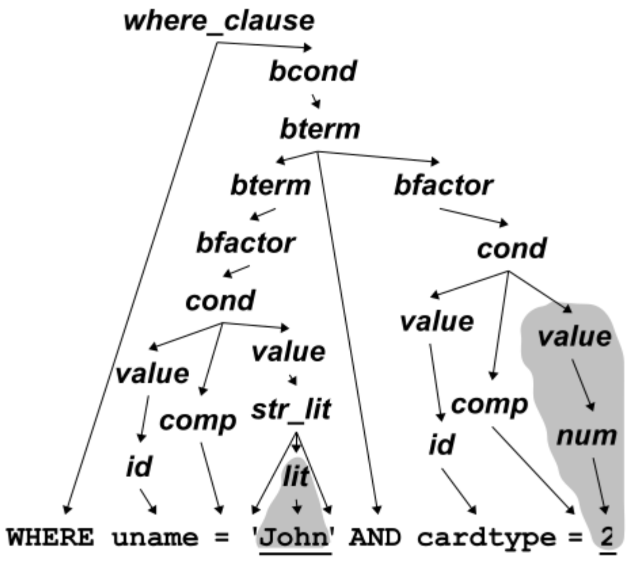


Fig. 4. Example of a "good" query.

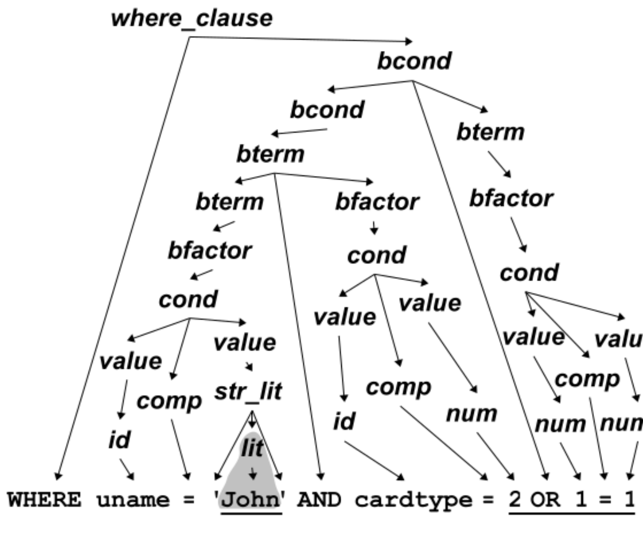


Fig. 5. Example of a "bad" query.

is based on the following consideration: "good" ("Fig. 4") queries have the same parse trees, "bad " are non-good ("Fig. 5')' parse trees. If during any execution of the program the requests will have the same AST, then there is no injection in this place, otherwise - maybe, so there may be false positive results. The paper provides an algorithm on how to understand from the input data whether they lead to an injection. The algorithm is exact in the sense that if it gives the result "no injection", then it really does not exist.

The following paper by Wasserman et al. [15] provides a conservative algorithm for checking that there are no injections in the program (conservative in the sense that if the algorithm says that there is no injection, then there is definitely no injection. False-positive results are possible). The algorithm is as follows:

1) for each hotspot, we build a context-free grammar of the language, consisting of strings that can fall into the input of this hotspot (the language that the grammar defines is larger than the real language, which consists of strings that can fall into the input of this hotspot);
2) build a grammar of "bad" lines (i.e. injections);
3) we check that two languages have no intersection. This method was done on the basis of the algorithm from Minamide [16], who made their own algorithm based on the work of Reps et al. [17]–[21] on the reachability of the language of the grammar.

The method was implemented for the PHP language.
Exploit generation was presented in the following paper by Wasserman et al. [22] The algorithm is as follows:

1) using the concolic execution method, we traverse the program;
2) if we encounter a hotspot while bypassing the program, then we build the grammar of the language for which member strings may work as input to the hotspot (due to the fact that we use concolic execution and not static methods, this grammar is calculated more accurately);
3) we intersect this language with the language of "bad" strings;
4) if the intersection is not empty, then we take any string from the resulting language, which will be exploit.

It also uses the methods from Minamide [16] and Reps et al. [17]–[21]
The method was implemented for the PHP language.
In Kieyzun et al. [23] concolic execution is considered to cover various paths in the program and detect vulnerabilities and mutation of input parameters in order to select an attack vector that leads to the exploitation of a vulnerability. As vulnerabilities, injections of SQL commands and first-order JavaScript code (first-order SLQi and XSS) and injection of second-order JavaScript code (second-order XSS) are considered. This work is interesting not only for the combination of techniques used to detect vulnerabilities, but also for the active use of third-party existing tools, as well as for the first in the literature automatic detection of second-order XSS vulnerabilities using the concolic database.

VI. Future work

There are several directions for further development of our tool:

1) We evaluated our tool only with synthetic examples. The fact, is that it is quite difficult to automatically apply

tool to some real-world web applications due to their code size and complexity. As one of the future steps, it is possible to test our tool with open source web applications

2) When finding a hotspot, we do not use information about what requests can come to the input of this hotspot, although we actually do have them - thanks to concolic execution, we have all the constraints on this data available. It would be possible to implement an approach close to the one proposed by Wasserman et al. team, but we expect loss of precision due to the fact that the input data arriving at the hotspot input is not always described by a context-free grammar.
3) Extend our model to find XSS and other code injection classes.
4) The request' and response' models in our solution are not precise, they only have a few parameters. Their refinement would increase the reliability of our tool.

VII. Conclusion

In the paper, we presented a prototype web application vulnerability search tool based on abstract interpretation, and also presented a description of the stands on which this tool was tested.

Acknowledgment

The results of the project “Automatic search for vulnerabilities in web applications based on big data processing”, carried out as part of the National Technology Initiative Competence Center Program “Big Data Storage and Analysis Center”, supported by the Ministry of Science and Higher Education of the Russian Federation under the Agreement of the Lomonosov Moscow State University with the National Technology Initiative Project Support Fund dated August 15, 2019 No 7/1251/2019.

References

[1] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[3] B. Loring, D. Mitchell, and J. Kinder, “Expose: practical symbolic execution of standalone javascript,” in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 196–199.

[4] B. Loring, D. Mitchell, and J. Kinder,, “Sound regular expression semantics for dynamic symbolic execution of javascript,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 425–438.

[5] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 615–618.

[6] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, “Jalangi: A selective record-replay and dynamic analysis framework for javascript,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[7] “Jalangi2: Dynamic analysis framework for javascript.” [Online]. Available: https://github.com/Samsung/jalangi2

[8] L. d. Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[9] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[10] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008, pp. 209–224.

[12] “Node.js v17.8.0 documentation.” [Online]. Available: https://nodejs.org/api/http.html

[13] “sqlmap: automatic sql injection and database takeover tool.” [Online]. Available: https://sqlmap.org/

[14] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[15] G. Wassermann and Z. Su, “Sound and precise analysis of web applications for injection vulnerabilities,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[16] Y. Minamide, “Static approximation of dynamically generated web pages,” in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 432–441.

[17] T. Reps, M. Sagiv, and S. Horwitz, *Interprocedural dataflow analysis via graph reachability*. Datalogisk Institut, Københavns Universitet, 1994.

[18] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[19] D. Melski and T. Reps, “Interconvertbility of set constraints and context-free language reachability,” *ACM SIGPLAN Notices*, vol. 32, no. 12, pp. 74–89, 1997.

[20] T. Reps, “Program analysis via graph reachability,” *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[21] D. Melski and T. Reps, “Interconvertibility of a class of set constraints and context-free-language reachability,” *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 29–98, 2000.

[22] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, “Dynamic test input generation for web applications,” in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 249–260.

[23] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of sql injection and cross-site scripting attacks,” in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 199–209.