IJTRET

International Journal of Trendy Research in Engineering and Technology
Volume 4 Issue 5 Proceedings of NCCETCC'20
ISSN NO 2582-0958

Proceedings of National Conference on Cutting Edge Technologies in Computing & Communications
held at Sriram Engineering College, Chennai on 6th March 2020

AN INTEGRATED APPROACH FOR EFFECTIVE INJECTION VULNERABILITY ANALYSIS OF WEB APPLICATIONS

[1]S.Sangeetha,[2]Mrs.S.Kalaivani
[1]Student, [2]Associate Professor, Department of Computer Science and Engineering,
Sri Venkateswara College of Engineering and Technology,
Thiruvallur, Thirupachur, Tamil Nadu 631203

ABSTRACT

Vindictive clients can assault Web applications by abusing infusion vulnerabilities in the source code. This street numbers the test of recognizing infusion vulnerabilities in the server-side code of Java Web applications in an adaptable and compelling manner. We propose an incorporated methodology that flawlessly consolidates security cutting with half breed limitation illuminating; the last coordinates automata-based settling with meta-heuristic inquiry. We utilize static examination to extricate insignificant program cuts applicable to security from Web programs and to create assault conditions. In benchmark applications, show that our methodology (actualized in the JOACO instrument) is essentially more successful at recognizing infusion vulnerabilities than cutting edge draws near, accomplishing 98% review, without delivering any bogus caution. We additionally analyzed the requirement fathoming module of our methodology with cutting edge imperative solvers, utilizing six diverse benchmark suites; our methodology effectively tackled the most noteworthy number of limitations (665 out of 672), without delivering any erroneous outcome, and was the one with minimal number of break/bombing cases.

Keywords: *Vulnerability detection, constraint solving, static analysis, search-based software engineering.*

1.INTRODUCTION

Symbolic execution and constraint solving represent a state-of-the-art approach used in security analysis to identify vulnerabilities in software systems. Symbolic execution executes a program with symbolic inputs and at the end generates a set of path conditions. Each of them corresponds to a constraint imposed on the symbolic inputs to follow a certain program path, i.e., a constraint characterizing a possible execution. By solving these constraints with a constraint solver, one can determine which concrete inputs can cause a certain program path to be executed. In the context of security analysis this approach is used [1], [2], [3], [4] to detect injection vulnerabilities, i.e., program locations in which certain malicious inputs can alter the intended program behavior. Roughly speaking, this approach consists of solving the constraints obtained by conjoining the path conditions (generated by the symbolic execution) with attack specifications provided by security experts. The main strength of this approach is that vulnerability detection yields a limited number of false positives, since the concrete inputs determined with constraint solving prove the existence of vulnerabilities. However, the effectiveness and precision of this approach are challenged by two main problems that affect symbolic execution and constraint solving [5]: 1) path explosion

and 2) solving complex constraints (e.g., constraints involving regular expressions or containing string/mixed or integer operations). Notice that while these problems are independent from the context in which symbolic execution and constraint solving are applied, the solutions to mitigate them can be tailored for a specific context. Nevertheless, existing proposals [1], [2], [3], [4] in the context of vulnerability analysis have not fully addressed theseproblems.

The path explosion problem is triggered by the huge number of feasible program paths that symbolic execution has to explore in large programs. To mitigate this problem in the context of vulnerability analysis, in previous work [6] we proposed an approach to extracting security slices from Java programs. A security slice contains a concise and minimal sequence of program statements that affect a given security sensitive program location (sink), such as an SQL query statement. Symbolic analysis can then be performed on security slices instead of the whole program; in this way path conditions are analyzed only with respect to the paths leading to sinks instead of every path in the program. Since, according to our experience [6], the number of sinks in a program is low1 and security slices are much smaller (approx. 1%) than the program containing them, this approach can effectively mitigate the path explosion problem.

The problems related to solving complex constraints are mainly due to the support for strings and their operations. In general, solving constraints that contain string operations requires to analyze the implementation of these operations, unless they can be treated as primitive functions in the constraint solver. However, there are typically hundreds of string operations in a given programming language that cannot be solved because their semantics is not known to the constraint solver (e.g., java.lang.Object.hashCode()andjava.lang.String.forma t(...)); we denote these operations as unsupported operations. Existing approaches support only a limited number of string operations—such as concatenation, assignment, and equality—as primitive functions. More complex operations have to be analyzed and transformed into an equivalent set of basic constraints containing primitive functions. This task is often not trivial and requires proficiency in the input language of the solver. A constraint solver that supports a limited set of operations can fail in solving constraints that contain unsupported operations, resulting in missedvulnerabilities.

To partially mitigate this problem, in previous work [7] we proposed a fallback mechanism to extend existing string constraint solvers for dealing with constraints with unsupported string operations. This mechanism, implemented in the ACO- Solver tool, used an off-the-shelf automata-based string constraint solver combined with a search-driven constraint solving procedure based on the Ant Colony Optimization meta-heuristic [8].

The goal of the work presented in this paper is to provide a scalable approach, based on symbolic execution and constraint solving, to effectively find injection vulnerabilities in source code, which generates no or few false alarms, minimizes false negatives, and overcomes the path explosion problem and the one of solving complex constraints.

We propose a new analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid

constraint solving. We leverage our previous work on security slicing [6] to mitigate the path explosion problem, by generating during the symbolic execution only the constraints that characterize the security slices of the program under analysis. This step allows us to identify paths and statements in the program where vulnerabilities can be exploited; this helps make the remainder of the approach scalable. The generated constraints are then preprocessed in order to simplify the followingstep.

The next step uses a hybrid approach that orchestrates a constraint solving procedure for string/mixed and integer constraints with a search-based constraint solving procedure. The idea behind this hybrid solving strategy is to solve a constraint through a two-stage process:

1. First, our solving procedure solves all the constraints with supported operations, by leveraging automata-based solving for solving string and mixed constraints,and linear interval arithmetic for solving integer constraints. In both cases, constraint solving rules are expressed using recipes that model the semantics of the operations. In particular, we provide recipes for many string/mixed operations, including 16 input sanitization operations from widely used security libraries [9], [10], and commonly used integer operations. In this way, the constraints involving supported operations can be efficiently solved, without transforming them into a set of primitivefunctions.

2. In the second stage, we use a search-driven solving procedure, which is based on and extends our previous work [7]. This procedure leverages the Ant Colony Optimization meta-heuristic to solve the remaining constraints which contain unsupported operations. The search space of this procedure is represented by the input domains as determined in the first stage; the search is driven by different fitness functions, depending on the type of theconstraints.

The solver in the first stage is used to reduce (possibly in a significant way) the search space, i.e., the domains of the string and integer variables, for the search- driven solving procedure; hence, it makes the search in the second stage more scalable and effective.

Our approach constitutes a targeted security analysis method, since its target (i.e., the type of vulnerabilities to analyze) can be specified by providing the corresponding threat models, i.e., generalized attack specifications (also called attack patterns) associated with different types of sinks. The approach analyzes the satisfiability of a threat model in conjunction with the path condition that leads to a given sink. In this sense, our approach can be considered general, since it can detect any type of vulnerability whose threat model can be described using regular expressions. To show this generality, in this paper we provide the threat modelsfor five common types of vulnerabilities: crosssite scripting (XSS), SQL injection (SQLi), XPath injection (XPathi), XML injection (XMLi), LDAP injection—LDAPi). Moreover, the approach is also language-independent, since the modeling of string/mixed and integer operations proposed in this paper—although provided in the context of the Java language—can be easily ported to other languages. Our integrated technique achieves high effectiveness in detecting vulnerabilities. In particular, we assessed the vulnerability detection capability of our tool (JOACO) by comparing it, using a benchmark comprising a set of diverse and representative Webapplications

2.ExistingSystem:

In This Existing System propose an approach to assist security auditors by defining and experimenting with pruning techniques to reduce original program slices to what refer to as security slices, which contain sound and precise information. To evaluate the proposed approach, This System compared our security slices to the slices generated by a state-of-the-art program slicing tool, based on a number of open-source benchmarks. On average, our security slices are 76% smaller than the original slices.

3.SYSTEMDESIGN

3.1.Constraints and ConstraintNetworks

The following definitions are based on the ones presented in the constraint solving literature [16], [17]. Let Y = y1, . . . , yk, k > 0 be a finite sequence of variables and D1, . . . , Dk a sequence of domains, with each variable yi ranging over the respectivedomaindom(yi)=Di.AconstraintcoverYisarelationoverY,i.e.,c⊆ D1×· · ·×Dk; Y is also called the scope of the constraint and k is its arity. Informally, a constraint c on some variables is a subset of the cartesian product overthevariabledomainsthatcontainsthecombination ofvaluesthatsatisfyc.A constraintnetworkRisatriple(X,D,C),whereXisafinitesequenceofvariables x1,...,xn,eachassociatedwithadomainD1,...,Dn,andC={c1,...,ct}isa setofconstraints;thescopeofeachconstraintci,denotedbySi,isasubsequence ofX.AconstraintnetworkR=(X,D,C)canberepresentedasahypergraphH= (V, S) where the set of nodes V corresponds to the set of variables X of the network, and S = Si , . . . , St is the set of hyperedges that group variables belonging to the same scope. The union (∪) and intersection (∩) operators for constraintnetworksreturnanotherconstraintnetworkandaredefinedasfollows: giventwoconstraintnetworksR1=(X1,D1,C1)andR2=(X2,D2,C2),R3=R1⊕R2, where ⊕∈ {∪, ∩}, R3 = (X3, D3, C3) with X3 = X1 ⊕ X2, D3 = D1 ⊕ D2, and C3 = C1 ⊕ C2. R1 is a subgraph of R2, denoted by R1 ⊆ R2, if and only if R1 ∩ R2 =R1.

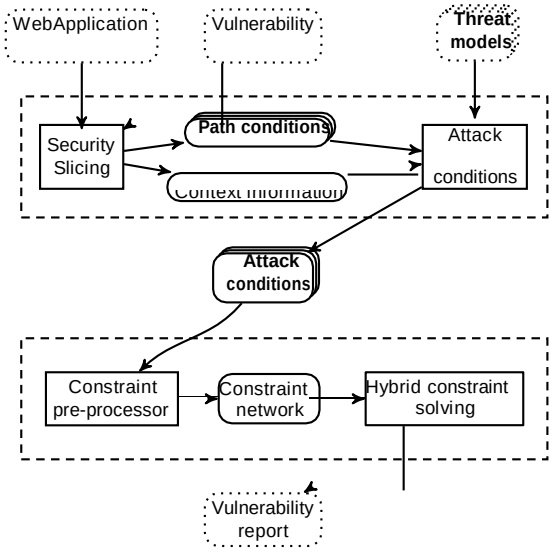

Figure 1. Overview of the approach

3.2.Ant ColonyOptimization

Ant Colony Optimization (ACO) [8] is a widely used metaheuristic search techniques for solving combinatorial optimization problems. It is inspired by the observation of the behavior of real ants searching for food. Real ants start seeking food randomly; when they find a source of food, they leave a chemical substance (called pheromone) along the path that goes

from the food source back to the colony. Other ants can detect the presence of this substance and are likely to follow the same path. This path, populated by many ants, is called pheromone trail and serves as a guidance (e.g., positive feedback) for the other ants. In ACO, these observations are translated into the world of artificial ants, which can cooperate to find a good solution to a given optimization problem. The optimization problem is translated into the problem of finding the best path on a weighted graph. Artificial pheromone trails are numeric parameters that characterize the graph components (i.e., nodes and edges); they encode the "history" in approaching the problem (and finding its solutions) by the whole ant colony. ACO algorithms also implement a mechanism, inspired by real pheromone evaporation, to modify the pheromone information over time so that ants can forget the (search) history and start exploring new search directions. The artificial ants build their solutions by moving step-bystep along the graph; at each step they make a stochastic decision based on the pheromonetrail.

3.3Injectionvulnerabilities

Injection vulnerabilities are program locations in which certain malicious input can be "injected" into the program to alter its intended behavior or the one of another system. An injection may occur when the user input is passed through the program to an interpreter or to an external program (e.g., a shell interpreter, a database engine) and the input data contain malicious commands or command modifiers (e.g., a shell script, an additional constraint of an SQL query). An injection vulnerability arises when the input is not properly validated or sanitized in correspondence of a sink. Injection vulnerabilities can cause serious damage to a system and its users. For example, an attacker could compromise the systemsunderlying the application or gain access to a database containing sensitive information. The "OWASP (Open Web Application Security Project) Top 10 2013" report [18] shows that injection vulnerabilities are the most common application security risk for Web applications. There are several types of injection vulnerabilities. In this paper we focus on the following five types, for which we give a short overview and an example based on the CWE (Common Weakness Enumeration)dictionary.

4. CONCLUSION

This work addresses the challenge of analyzing the source code of a Java Web application for detecting injection vulnerabilities in a scalable and effective way. We have proposed an integrated approach that seamlessly combines static analysis-based security slicing with hybrid constraint solving, that is constraint solving based on a combination of automata-based solving and meta-heuristic search (Ant Colony Optimization). We use static analysis to extract minimal program slices from Web programs relevant to security and to generate the attack conditions, i.e., conditions necessary for the slices to be vulnerable. We then apply a hybrid constraint solving procedure to determine the satisfiability of attack conditions and thus detect vulnerabilities. The experimental results, using a benchmark comprising a set of diverse and representative Web applications/services as well as security benchmark applications, show that our approach (implemented in the JOACO tool) is significantly more effective at detecting injection vulnerabilities than state-of-the-art

approaches, achieving 98% recall, without producing any false alarm. We also compared the constraint solving module of our approach with state- of-threat constraint solvers, using six different benchmarks; our approach correctly solved the highest number of constraints (665 out of 672), without producing any incorrect result, and was the one with the least number of time-out/failing cases. In both scenarios, the execution time was practically acceptable, given the offline nature of vulnerability detection. As part of future work, we plan to extend our integrated vulnerability detection approach with support for widely used Java Web frameworks such as Spring [77]. We also plan to incorporate dynamic symbolic execution to further enhance ourapproach.

5.REFERENCE:

[1] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst, "Automaticcreation of SQL injection and cross-site scripting attacks," inProceedings of ICSE'09. IEEE, 2009, pp.199–209.

[2] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, andD. Song, "A symbolic execution framework for JavaScript," inProceedings of S&P'10. IEEE, 2010, pp.513–528.

[3] X. Fu, M. Powell, M. Bantegui, and C.-C. Li, "Simple linear string constraints," Form. Asp. Comput., vol. 25, no. 6, pp. 847–891,2013.

[4] Y. Zheng and X. Zhang, "Path sensitive static analysis of Web applications for remote code execution vulnerability detection," in In Proceedings of ICSE'13. IEEE, 2013, pp.652–661.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.0098-5589

[6] J. Thom´e, L. K. Shar, D. Bianculli, and L. Briand, "Security slicing for auditing common injection vulnerabilities," J. Syst. Softw., 2017, (in press) https://doi.org/10.1016/j.jss.2017.02.040.

[7] J. Thom´e, L. Shar, D. Bianculli, and L. Briand, "Search-driven string constraint solving for vulnerability detection," in Proceedings of ICSE'17. IEEE, 2017, pp.198–208.

[8] M. Dorigo and K. Socha, "An introduction to ant colony optimization," IRIDIA, Tech. Rep. TR/IRIDIA/2006-010,2006.

[9] Apache,"StringEscapeUtils,"https://commons.apache.org/proper/commons- lang /javadocs/api-3.1/org/apache/commons/lang3/StringEscapeUtils.html,2017.

[10]OWASP, "OWASP ESAPI,"https://www.owasp.org/index.php/Category:OWASP Enterprise Security API,2017