Partitioners Track: Generating Security Vulnerabilities in Source Code

Felix Schuckert[1]

Abstract: This paper describes a framework, which modifies existing source code to generate security issues. An example plugin for generating SQL injection in Java source code is described. The generation process is based on static code analysis techniques like dataflow analysis and abstract syntax trees. The framework is evaluated with the help of Java projects from GitHub. One modified project was successfully used in a capture the flag event as a challenge.

Keywords: Software security, sql injection, generating, static code analysis

1 Introduction

Modern software products usually have a connection to the internet. All of them are exposed to attackers that want to exploit the software. The top security issues remained the same over the last years. Software developers have to protect their software from attackers by developing secure software. As stated in the paper [PE09], software security can't be provided only by the operating system. The software must have no vulnerabilities as well. Developers usually write tests, if common security vulnerabilities in their programs do exist, but this will not guarantee the security of the developed software. Instead it will verify the defined function requirements [Ba]. Static code analyzer tools help to find existing security issues in source code. These issues have to be reviewed and fixed manually by software developers. Fixing such issues requires a high amount of skill in software security. Another problem is that some security issues might not be detected by using code analyzer. At best all software developers should have software security skills to prevent software security issues preemptively in the development process.

How can developers gain software security skills? There are existing projects that provide manually created exercises which contain security issues. These exercises can be used to learn how the security issue can be exploited and how it can be fixed. The projects usually provide only a few exercises per security issue category. Having a framework that can be used for generating security issues in existing source code helps to generate a huge amount of exercises. The modified source code can be used to find issues in projects that contain more than a few lines of code, just created for the exercise. The modified project may be utilized to exploit and fix the issue as well.

Currently there are many papers about evaluating static code analyzers for detecting bugs and security issues. Some evaluated existing tools to detect security issues against SAMATE [SA10] database [GD13]. Also non-commercial static analysis tools like Yasca and

[1] HTWG Konstanz, Informatik, Brauneggerstr. 55, 78462 Konstanz, felix.schuckert@htwg-konstanz.de

FindBugs were evaluated for detecting security issues. [HHA14] Another research compared the detection of SQL injection vulnerabilities using static code analyzer and penetration testing. Static code analyzers found the vulnerabilities with a higher success-rate, but also have a higher false positive rate [AV09]. The evaluation of using static code analyzer for security verification was researched on existing tools like MOPS, Splint et cetera [ZSR14]. The usage of static code analyzers for detecting security issues is a common way of preventing vulnerabilities. Using static code analyzer for generating security vulnerabilities is the main goal of this work. There already exists a research about transforming *Statement* calls in Java to *PreparedStatement* calls using the refactoring features from Eclipse [AFS11]. This paper introduces a framework which is used for generating security vulnerabilities in existing source code.
Section 2 explains how the framework was implemented to generate security vulnerabilities. The next section 3 describes how the generation process is done using a SQL injection. The last section 4 shows how the modified source code looks like and points out the usage of the modified source code in a capture the flag event.

2 Framework for Generating Security Vulnerabilities

The framework implementation is method based. It will only generate security issues inside a method. The current implementation is focused on Java source code. The generation process requires basically three steps. The framework is using a plugin system for adding new generation processes. Each step can be implemented by plugins. In figure 1 the corresponding components are shown. The first step is marking methods with summaries. As described in [BC07] adding summaries to methods allows to scan in a wider scope than only scanning methods. Summaries will be added to method. These summaries can be included in the analysis of other methods, where corresponding methods will be called. For example the SQL injection plugin detects sanitize methods, which will be disabled in the generation process. The second step is the identification step, which searches for points of interests (POI). These will be used for generating the security vulnerability. As seen in the figure, the first two steps are using PMD [PM]. This tool is using abstract syntax trees (AST) for analyzing and provides features like dataflow analysis, XPath queries et cetera [Co05]. The last step is using the POIs found by the identifier and is modifying the AST to create the security vulnerability. This step is not done by using PMD. Instead it is using the tool Spoon [Sp]. This tool is using abstract syntax trees as well, but has the ability to modify it. Also the tool will generate the corresponding source code. It uses a simplified AST, which allows easier analyzing and modifying [Pa15]. The framework, explained in this paper, provides the functionality to find AST nodes in the Spoon AST from PMD nodes. This is basically done by the source code position. A plugin which implements these components will be able to generate security vulnerabilities in source code.

3 Plugin: SQL Injection

Evaluation of OWASP top ten [OW15] did show that injection is a top security issue category, which can be generated in Java source code. SQL injection was selected as the first

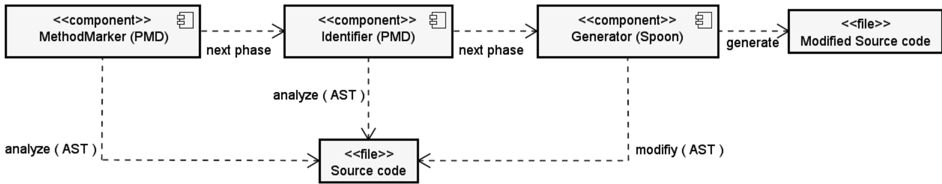

Fig. 1: Framework components in generation process.

plugin implementation for the framework because of the relevance and ease of implementation. This is used as a representative for the injection category. Each plugin requires a transformation model, which describes what source code parts have to be found and modified to generate the vulnerability. Figure 2 shows the transformation model for generating a SQL injection issue. The generation is similar to the detection of such an issue in source code. An examination of the Lapse+ project [OW] did show three basic parts of an injection vulnerability. Tainted sources which usually will be data from the user. These require no transformation to create the vulnerability. But they are important for checking, if any sanitize method will be used on them. Sanitize is the second part, which will remove or check for any suspicious characters. These methods have to be made useless so that the generated vulnerability can be exploited. Useless means that the function call still exists, but will have no effect and the program flow will still be the same as before. The last part is the sink. These are methods, where unchecked inputs should not be reached. The plugin will search for *PreparedStatements* instead, because they will be transformed to *Statements*. The corresponding method calls for setting the data and the SQL query statement have to be found as well. These parts are found by using PMD.

Tainted data sources have to be analyzed if they reach sanitize methods and sinks. This requires a modified tainted analysis instead of the usual tainted analysis described in [BC07]. It uses the dataflow analysis provided by PMD to detect sanitize method calls on tainted data. These calls will be saved and modified later in the generation step to be useless. The modified data flow analysis will check if tainted data will reach any set method call from the *PreparedStatement*. This will guarantee that the generated SQL injection can be exploited. The generator module implementation will use the found data and replace the *execute...()* call with the corresponding call using a *Statement*. The String concatenation will use the found SQL query statement and the set method calls. Exchanging the *execute...()* call will not change the behavior of the program in normal usage. It will only change the behavior, if the vulnerability will be exploited.

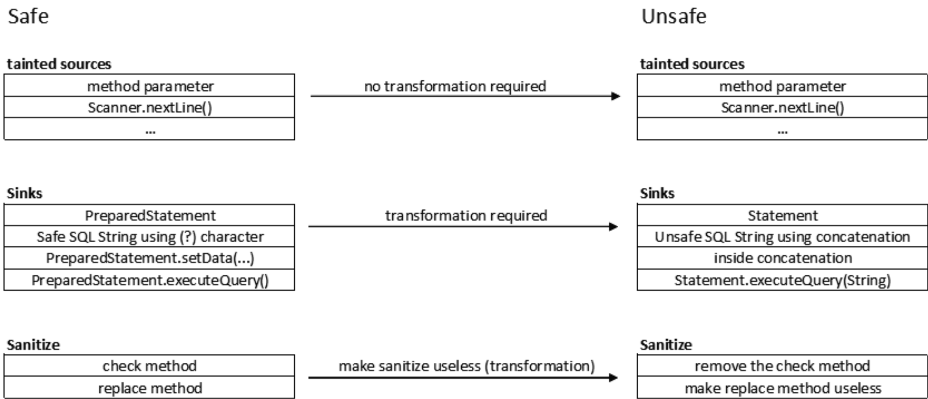

Fig. 2: Transformation model for creating a SQL injection vulnerability.

4 Evaluation and Usage

The generation process was evaluated using existing projects from GitHub. Following example shows an unmodified code snippet:

```
Connection connect = DriverManager.getConnection(
    "jdbc:mysql://127.0.0.1/test","","");
PreparedStatement ps = connect.prepareStatement("insert into teams
    (tname) values (?)");
ps.setString(1,tname);
ps.executeUpdate();
```

The plugin generated following source code by replacing the execute method call.

```
Connection connect = java.sql.DriverManager.getConnection(
    "jdbc:mysql://127.0.0.1:3306/test ?allowMultiQueries=true", "", "");
PreparedStatement ps = connect.prepareStatement("insert into teams
    (tname) values (?)");
ps.setString(1, tname);
String qryStr = "insert into teams (tname) values ('" + tname + "' )";
Statement statement=connect.createStatement();
statement.executeUpdate(qryStr);
```

The plugin also added the parameter for allowing multiple queries. This will encourage the developers, who are learning the SQL injection vulnerability, by having more possibilities to exploit the issue. Using multiple queries allows to add SQL commands like dropping the database or changing the admin password. The modified source code was evaluated using FindBugs [Fi] and Codepro AnalytiX [Co]. Both tools found the generated security vulnerability. Training the skills of developers to prevent SQL injection issues are still important because the issues found by tools have to be fixed manually.

The UCSB iCTF Competition [iC] provides a capture the flag event each year. These are attack/defense events where every team will maintain a server with running services. Each of these services have security vulnerabilities, which have to be fixed on the own server and have to be exploited on other servers to receive points. In 2015 each participating team had to submit one service with a security vulnerability. A project from GitHub modified by this framework was used for one service. This shows the first real usage of the framework. The event had 35 teams/services. The generated and submitted service was the fourth exploited in the contest. The submitted service was the second most exploited service from different teams. Stolen flags from the service were just at the 17th place. The main reason for this was, that some teams did find an exploit which caused a denial of service. This denial did also deny other teams from scoring points by using their exploits. The teams had to find and fix the SQL injection issue and additionally had to fix the database to get the service up on their own servers. This shows how important the prevention of SQL injection issues are. The framework allowed to generate exercises without any hassle using existing projects from GitHub.

5 Conclusion

Developers have to know security issues that have to be prevented during development. The developed framework allows to generate exercises, which can be used to teach developers how security issues can be exploited and how to prevent them. It can create different scenarios by using different source code. The issue will remain the same, but it will teach the developers to find the generated security issues in modified real life projects.

A basic framework for generating SQL injection was developed successfully. It uses abstract syntax trees for the analysis and generation parts. Dataflow analysis are used to do the modified tainted analysis. This allows to verify, if the vulnerability can be exploited. First usage in a CTF event showcased that the generated issues are easy to exploit. It requires further research to generate vulnerabilities with different complexity factors. This can be accomplished by having different permutations of the generation process.

The evaluation requires further steps. It has to be used as exercises for developers and the learning effect has to be examined. Further research in scanning other files, than source code like configuration files would allow to generate security vulnerabilities like cross-site-scripting. Also having the ability to generate vulnerabilities in memory unsafe languages as C/C++ would allow to generate buffer-overflow issues. As stated in the paper [EDB06] memory safe languages like Java do not allow to write memory parts which are not allocated for the corresponding variable. This will prevent any buffer overflows inside a Java program. This research did show an example generating SQL injections in the source code. Similar analyzing methods can be used in C/C++ because PMD supports it too. The generation part requires another tool for modifying C/C++ source code.

Literatur

[AFS11] Abadi, Aharon; Feldman, Yishai A.; Shomrat, Mati: Code-motion for API Migration: Fixing SQL Injection Vulnerabilities in Java. In: Proceedings of the 4th Workshop on Refactoring Tools. WRT '11, ACM, New York, NY, USA, S. 1–7, 2011.

[AV09] Antunes, Nuno; Vieira, Marco: Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In: Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on. S. 301–306, 2009.

[Ba] Baca, Dejan: Automated static code analysis: A tool for early vulnerability detection. ISBN 978-91-7295-161-7.

[BC07] Brian Chess, Jacob West: Secure Programming with Static Analysis. 2007. ISBN 978-0321424778.

[Co] CodePro AnalytiX. https://developers.google.com/java-dev-tools/codepro/, last visit: 2015-09-29.

[Co05] Copeland, Tom: PMD Applied. 2005. ISBN 0-9762214-1-1.

[EDB06] Emery D. Berger, Benjamin G. Zorn: DieHard: probabilistic memory safety for unsafe languages. ACM SIGPLAN Conference on Programming Language Design and Implementation, S. 158 – 168, 2006. ISBN 1-59593-320-4.

[Fi] FindBugs SourceForge. http://findbugs.sourceforge.net, last visit: 2015-09-29.

[GD13] Garbiel Diaz, Juan Ramon Bermejo: Static analysis of source code security: Assessment of tools against SAMATE tests. Information and Software Technology, 55:1462–1476, 2013.

[HHA14] Hamda Hasan AlBreiki, Qusay H. Mahmoud: Evaluation of static analysis tools for software security. Innovations in Information Technology (INNOVATIONS), 2014 10th International Conference on, S. 93 – 98, note = ISBN 978–1–4799–7210–4, 2014.

[iC] iCTF. https://ictf.cs.ucsb.edu/, last visit: 2015-12-12.

[OW] OWASP Lapse+. https://lapse-plus.googlecode.com/files/LapsePlusTutorial.pdf, last visit: 2015-09-29.

[OW15] OWASP Top Tent. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, last visit: 2015-09-03.

[Pa15] Pawlak, Renaud; Monperrus, Martin; Petitprez, Nicolas; Noguera, Carlos; Seinturier, Lionel: Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software: Practice and Experience, S. na, 2015.

[PE09] Pistoia, Marco; Erlingsson, Úlfar: Programming Languages and Program Analysis for Security: A Three-year Retrospective. SIGPLAN Not., 43(12):32–39, Februar 2009.

[PM] PMD GitHub. https://pmd.github.io/, last visit: 2015-09-03.

[SA10] SAMATE - Software Assurance Metrics and Tool Evaluation. https://samate.nist.gov, last visit: 2015-12-13.

[Sp] Spoon. http://spoon.gforge.inria.fr/, last visit: 2015-10-02.

[ZSR14] Zhioua, Zeineb; Short, Stuart; Roudier, Yves: Static Code Analysis for Software Security Verification: Problems and Approaches. In: Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International. S. 102–109, 2014.