Technische Universität München

Automated Vulnerability Check in Continuous Integration

Bachelor Thesis

Supervisors: Prof. Dr. Alexander Pretschner, Aleieldin Salem
Email: {alexander.pretschner, salem} @ in.tum.de
Phone: +49 89 289 – 17, 314
Starting date: immediately

Fakultät für Informatik
Lehrstuhl 22
Software Engineering
Prof. Dr. Alexander Pretschner

Boltzmannstraße 3 85748
Garching bei München

Tel: +49 89 289 17885,
+49 89 289 17314

Web: http://www22.in.tum.de

Context

Using components with known vulnerabilities is still one of the most frequent causes for security incidents today [5]. It is of the essence, therefore, to possess a comprehensive overview of the software components utilized by the software, in order to mitigate the vulnerabilities they may withold prior to releasing the software. On the one hand, tools like Maven[1] provide a reliable overview of the used components and libraries. On the other hand, entities like NIST through its National Vulnerability Database (NVD) provide the required information about existing security vulnerabilities. By augmenting the two breeds of tools/repositories we can address the problem at hand via highlighting the vulnerable components used by the software.

There are few tools that deliver this functionality, the most prominent of which is *OWASP-Dependency-Check*, a Maven plugin that checks a project's components against the NVD. By searching through the database for each dependency, the tool is able to detect every associated Common Vulnerability and Exposure (CVE)[2] entries associated with different components. Nevertheless, the tool does not take any further actions to determine whether an identified vulnerable component has any security impact on the program e.g. by pinpointing the code segments that utilize the vulnerable components. In other words, the output of OWASP-Dependecy-Check merely informs the user that they are importing a vulnerable component.

Despite the vagueness of the tools output–which usually leads to a noticeable rate of false positives, developers and testers cannot afford to ignore the reported vulnerabilities. Consequently, they are obliged to manually search for the vulnerabilities associated with the imported components, identify the vulnerable functions/modules, trace those back to their own code, and reason about the technical soundness of the reported security risk. Needless to say, the usefulness of the OWASP tool within an automated Continuous Integration environment plummets due to the cost incurred from addressing false alarms on regular basis.

Goal

The goal of this thesis is to extend the OWASP-Dependency-Check tool so as to enhance its usability in a continuous integration environment. We plan to achieve this by upgrading the tool's output such that it (a) highlights the segment of code that uses the vulnerable component, and (b) reports whether such vulnerability actually compromises the security of the software.

In this project, we evaluate our tool on data provided by Secunet AG. Within this context, the OWASP-Dependecy-Check tool is fed a Maven's *pom.xml* file including information about the project, its modules, configurations, imported components, et cetera. Furthermore, projects are written primarily in *Java*, and utilize Java archive *.jar* files.

[1] https://maven.apache.org/
[2] http://cve.mitre.org/

Technische Universität München

Fakultät für Informatik
Lehrstuhl 22
Software Engineering
Prof. Dr. Alexander Pretschner

Boltzmannstraße 3 85748
Garching bei München

Tel: +49 89 289 17885,
+49 89 289 17314

Web: http://www22.in.tum.de

Given the original vulnerable JAR components report generated by OWASP-Dependency-Check, our extended version of the tool is expected to do the following. Firstly, for every single reported component, the tool should investigate the source code in pursuit of segments that use the vulnerable component. This includes mining the JAR file's bytecode, extracting lists of exposed functions, and matching them to calls within the source code. Secondly, the tool should retrieve the CVE's associated with the vulnerable module from the intial report. Thirdly, due to the fact that CVE entries are often vague–as they are meant to be read by humans, the tool should parse the CVE entry's description and extract the class of vulnerability e.g. buffer overflow, remote code execution, and so forth. Lastly, the tool should support a basic, qualitative feedback mechanism that gives a qualitative measure of whether the identified segment of source code exposes a vulnerability. For example, using a component vulnerable do buffer overflow should not be reported as long as the necessary boundary checks are enforced/implemented.

Since the primary objective of this project is to enhance the original Maven plugin, our evaluation is based on comparing the performance of, both, the original and extended versions of such plugin. We evaluate performance using two dimensions viz., time/resource consumption and technical soundess. For the former dimension, we record the time taken to generate the vulnerability reports, and measure the overhead added by the extended plugin. As for the latter dimension, we reside to human expertise to measure the false positives and false negatives rates. Our primary hypothesis is that the extended version of the OWASP-Dependency-Check plugin is expected to report false positive rates less that its original counterpart. The secondary hypothesis of this thesis is that the extended plugin is expected to report false negative rates that are, at least, as low as its original counterpart.

Work-plan

1. Develop an understanding of the JVM ByteCode and .class-Format
2. Inspect the OWASP-Dependency-Tool and find possible optimizations to reduce amount of false positives
3. Design and implementation of the Maven plugin
    a. Extract functions from vulnerable Java component
    b. Identify source code segments that utilize the extracted functions
    c. Retrieve CVE entries associated with vulnerable component
    d. Parse retrieved CVE entries and extract vulnerabilities
    e. Report whether the source code exposes the vulnerabilities
4. Evaluate the extended plugin
    a. Run original and extended plugins against different projects
    b. Record evaluation metrics e.g. time/resource consumption
    c. Consult human experts and record false positives and false negatives
    d. Compare the outputs of the two versions
5. The final thesis document must contain:
    a. Description of the problem and motivation for the chosen approach
    b. Description of the theoretical background
    c. Implementation description
    d. Evaluation of implementation and the reduction of the amount of false positives
    e. Conclusions and future work

Deliverables

- Maven-Plugin able to run and demonstrate desired functionality
- Source Code available as official fork of the OWASP-Tool on GitHub
- Final thesis report written in conformance with TUM guidelines

Fakultät für Informatik
Lehrstuhl 22
Software Engineering
Prof. Dr. Alexander Pretschner

Boltzmannstraße 3 85748
Garching bei München

Tel: +49 89 289 17885,
+49 89 289 17314

Web: http://www22.in.tum.de

References

[1] Eric Bruneton. Asm user guide, 2011.

[2] The Eclipse Foundation. Aether api, 2014.

[3] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. 2014.

[4] Oracle. Java documentation, 201.

[5] Open Web Application Security Project (OWASP). 2013 top 10 list, 2013.

[6] Jim Smith and Ravi Nair. *Virtual Machines*. 2005.