Tracking known security vulnerabilities in proprietary software systems

Cadariu, M; Bouwers, EM..; Visser, J.; van Deursen, A.

DOI
10.1109/SANER.2015.7081868

Publication date
2015

Document Version
Accepted author manuscript

Published in
Proceedings - 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering

Tracking Known Security Vulnerabilities in Proprietary Software Systems

Mircea Cadariu*‡, Eric Bouwers*, Joost Visser*†, Arie van Deursen‡
* Software Improvement Group, The Netherlands
† Radboud University Nijmegen, The Netherlands
‡ Delft University of Technology, The Netherlands
m.cadariu@sig.eu, e.bouwers@sig.eu, j.visser@sig.eu, arie.vandeursen@tudelft.nl

***Abstract*—Known security vulnerabilities can be introduced in software systems as a result of being dependent upon third-party components. These documented software weaknesses are "hiding in plain sight" and represent low hanging fruit for attackers. In this paper we present the Vulnerability Alert Service (VAS), a tool-based process to track known vulnerabilities in software systems throughout their life cycle. We studied its usefulness in the context of external software product quality monitoring provided by the Software Improvement Group, a software advisory company based in Amsterdam, the Netherlands. Besides empirically assessing the usefulness of the VAS, we have also leveraged it to gain insight and report on the prevalence of third-party components with known security vulnerabilities in proprietary applications.**

I. Introduction

The *OWASP Top Ten* [3] exposes typical software security flaws of software systems. In 2013, *Using Components with Known Vulnerabilities* [2] is listed as number 9 in this list. Known vulnerabilities are security flaws which are disclosed through public communication channels. When a component with a vulnerability is used in a software system the risk of leaking data or loosing control over servers increases.

Known vulnerabilities are easy to track by attackers and they enable automated large scale exploits. To illustrate this, consider Shodan, a search engine which has been shown to be useful in identifying Internet-facing industrial control systems [5]. A keyword search query on Shodan containing the term *Jetty 6.1.1* retrieves the Internet addresses of 464 hosts that expose their online services using this open-source vulnerable web server [8], for which the vulnerability description can also be accessed online [12].

The driving motivation of the work presented in this paper is that known vulnerabilities may be widespread and easy to exploit, but they can, and *should*, also be leveraged as efficiently as possible for benign purposes, such as to trigger corrective measures that reduce the opportunity for security breaches in software systems.

For example, automated solutions such as OWASP Dependency Check [15] are available to statically scan a software system and generate a report which lists those components for which a known security vulnerability is available. However, despite OWASP's awareness raising effort and the availability of these tools, it has been reported that approximately 1 out of 4 Java component downloads from the Maven Central Repository features a software component with a known vulnerability [19].

To understand this situation better, we present a case-study in which we embedded one of the available vulnerability scanning tools in the existing software quality monitoring process that SIG offers as a service to its customers. We call this monitoring service extension the Vulnerability Alert Service. With this set up, we first established how large the problem of using components with known security vulnerabilities actually is. Secondly, we evaluated the usefulness of the VAS, by observing the handling of over 400 alerts and by conducting interviews with VAS operators.

In short, this paper offers the following contributions:

- An empirical investigation into the prevalence of using components with known security vulnerabilities in practice
- A description of a process to handle alerts addressing security concerns
- An evaluation of the usefulness of vulnerability alerts in the context of external software product quality monitoring

II. Automated Known Vulnerability Detection

In this section, we describe a way to determine known vulnerabilities in Java projects. A *known* vulnerability has a CVE identifier and is identified by the OWASP Dependency Check tool. The use of this tool currently restricts the analyzed software systems to Java projects built with Maven. In the following subsections we provide some background information of these concepts and explain the matching mechanism used by the tool.

A. Maven

Maven[1] is the most widely used software tool used by Java developers to automate the application build process. One of its features is dependency management for which POM files are used – XML files in which (third-party) dependencies to be included in the build are specified.

B. CVE

The Common Vulnerabilities and Exposures (CVE) identifiers were introduced in 1999 to enable comparison between the existing security tools at the time [1]. Their format is

[1] http://www.maven.org

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

CVE–<year>–<identifier> (e.g.: CVE-2011-2730) and their description contains among others, the CPE and the vulnerability description.

The CPE identifiers uniquely identify the applications affected by the CVEs [6]. For example, the CPE identifier for the vulnerable Java library Apache Commons FileUpload is: `cpe:/a:apache:commons_fileupload:1.0`.

C. OWASP Dependency Check

After executing a selection procedure [7], the OWASP Dependency Check 1.0.5 [15] is selected in this study for automating the vulnerability scan. We extended the tool to be able to extract systems dependency information from the POM files of Java systems and to create list of the dependencies associated known vulnerabilities in XML, HTML or plain text formats. The extraction process only handles direct dependencies. Support for transitive dependencies is currently not available.

D. Matching Mechanism

We can extract dependency identification information from software projects from multiple sources. Considering applications in the Java ecosystem, we can find traces of third party code in their bytecode, JAR files, import statements or build manifest files (such as Maven POMs). These traces have to be matched with their counterparting CPEs in order to retrieve a list of CVEs. Below is an example match automatically conducted by Dependency Check using information from Maven POMs:

Maven dependency: org.*mortbay*.**jetty** **jetty** 6.1.20
CPE: cpe:/a:*mortbay*:**jetty**:6.1.20

The tool is based on partial token set overlap, in which tokens from the CPE identifier(*mortbay*, *jetty*) are searched in the tokens of the Maven dependency identifiers. If all CPE tokens are present, then it is declared a successful match, no matter if there are other tokens present in the Maven dependency description (such as *org*, in our example).

Working on the level of application names instead of matching them by code provides the following advantages:

- the approach is more lightweight – less data to be processed and stored than when working with snippets of binary code
- it removes the need to always obtain the source code of an application for which a vulnerability is disclosed
- it avoids the need to abstract from language-specific elements (e.g: compiler specific meta-data in binary code)

The trade-off that we have to make is in terms of accuracy. Using the name-based approach may result in false positives as a result of component identification incoherences between CPE and the application names found in software projects. This shortcoming would be avoided if CPE identifiers would align perfectly to the way in which component names can be retrieved from software systems.

After obtaining the list of CPEs from input systems, getting the corresponding CVEs means querying our data source to retrieve the CVEs indexed under the input CPEs.

E. Precision/Recall Analysis

To study false positive and false negative rates we used precision and recall. Precision is *the fraction of retrieved results which are relevant*. Recall is *the fraction of the relevant results which are retrieved* from the data set.

We scanned the Maven dataset [16] with Dependency Check and created a list of matched pairs from the output files. By manually labelling a random selection of 50 matches as being either true positive or false positive we obtained the false positive rate.

For studying recall, we constructed a dataset of library Maven identifiers for which we know that it has already one or more CVE entries assigned. This way, we can create specific inputs for which we expect the tool to capture, and see how many of the expected flags actually get raised. The false negative rate is determined by how many of these input elements do not get flagged, when in fact they should have.

Most of the matches for the precision study are false positives. Among the total of 50 matches, 7 were matched correctly, yielding a precision value of 7/50≈0.14. The majority of matches for the recall study are correct matches. The recall value for this dataset is 47/59≈0.80.

These values do not immediately deem the technique unusable, the low precision value indicates that only 1 out of every 7 alerts would be a true positive. On the other hand, the dataset used is large and contains many projects which are not necessarily used in an industrial context. Taking into account that any security issue found is useful, we proceeded with evaluating the tool in practice.

III. Vulnerable Dependencies in Practice

This research has been conducted within the Software Improvement Group (SIG), an advisory company based in Amsterdam, the Netherlands. Among its services, it provides Software Risk Assessment [18], Software Monitoring [10] [11] and Security Risk Assessment [20]. The professionals employed at SIG take the role of external quality evaluators for the software systems developed by various customers from multiple domains, such as banks, public transportation companies, governmental organizations, etc.

To assess the state of practice in the industry with regards to known security vulnerabilities, we used 75 proprietary systems currently or previously monitored or analysed by SIG. They come mostly from large Dutch companies which operate in the banking, public transportation, governmental, consultancy, electronic payments and household utilities fields. Technologically, these projects are Java projects built with Maven which include their third-party dependencies in POM files. The sizes of these projects range from 1.7 to 968 KLOC, and have between 4 and 238 dependencies.

After scanning the projects with the OWASP Dependency Check tool the resulting lists of security vulnerabilities was manually pruned of false positives. The results of the analysis show that 54 out of 75 projects use at least 1 (and up to 7) vulnerable libraries.

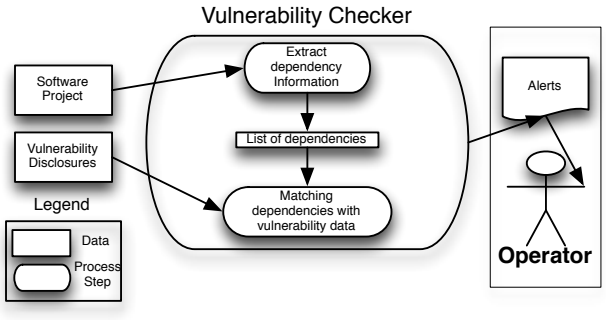

Fig. 1. The Vulnerability Alert Service process

IV. Vulnerability Alerting

The Vulnerability Alert Service is illustrated in Figure 1. As *process input* we have two elements: a software project and vulnerability data. Using Fawcett et al.'s *activity monitoring problems* [9] terminology, the inputs are derived from the *positive activity indicators*. For our context, these are: a project is found to include a library with known vulnerabilities, or a vulnerability was disclosed that is found to affect one of the libraries of the monitored projects.

This input is destined for the *vulnerability checker* (in our study OWASP Dependency Check takes this role) which has two tasks: extract dependency data, recognize them and match them with known vulnerabilities. The software projects are input for the extracting task, which produces a list of recognized dependencies. This list and vulnerability disclosures are input for the matching task.

Upon a successful match, the application generates an alert, which is consumed by a human operator. After acknowledging them, the operator proceeds to filter the alerts based on usefulness, and then reports them to the interested party.

V. Usefulness Evaluation

For evaluation, we constructed a three-part evaluation study. The first step is embedding the monitoring process in the daily operations of our host company and collecting the alerts which are raised. The second part covers the task of collecting relevant information from the research context with regards to the usefulness of the alert. This data is then used in the subsequent step, data analysis.

We used two data collection procedures: interviewing and direct observation. For both activities, technical consultants (TCs) from our host company participated. Both were not time-boxed, in order to allow sufficient time for thorough responses to emerge. See Table I for an overview of our data.

With interviewing, a series of questions regarding specific findings among monitored software systems were asked to technical consultants. After interviewing we have observed that generally, without this alerting service, the consultants would not consider outdated libraries and their security impact. Some considered that these findings would be easy to communicate to the clients, as the vulnerabilities are presented under a standard form which they can refer to. On the other hand, due to the fact that upon closer inspection these vulnerabilities do not make the application vulnerable right away, they were not immediately disclosed to the client. These findings would be presented at established meetings where multiple types of updates are presented.

TABLE I. Data collected

Data collection step	Nr. of alerts	Nr. of TCs	Nr. of projects
Interview-based	4	4	4
Observation-based	449	4	34

For the observation-based study, for a period of time, the technical consultant responsible with observing the evolution of software systems was informed of the alerts that were issued the day before, and he/she was asked to point out the useful ones which were written down on a note. This way, we can get insight on the usefulness of the alert methodology as a whole in its real context of use. The technical consultants considered two thirds of the alerts as useful.

One of the useful findings[2] produced in our research was communicated to the security officer at a large dutch banking organization. He was pleased with this finding and insisted on the application of corrective measures to remove the security vulnerability from their product. In addition, he expressed his interest in this type of findings and encouraged the responsible persons on behalf of our host company to continue contributing with such security-related findings. This provides further evidence for the practical usefulness of this information to technical consultants and the potential to make an positive impact at the customers.

VI. Discussion

The data collected in Section III and Section IV-A shows that the problem of depending on third party components with known vulnerabilities is prevalent in the software industry. The use of these types of components increase exploitation risks if the vulnerable libraries are not updated, and it is a general signal that this aspect is not properly taken care of in the industry.

The data and experiences in Section IV-A indicate that it is possible to to use tools like the OWASP Dependency Check in external software product quality assurance. While the raw figures state a false positive rate of over 70%, the false positive rate observed while deploying the solution is experienced to be lower. Reasons for this discrepancy can be that some vulnerable components are simply not widely used in industry.

An other explanation for the deviation is that even though the reported security vulnerability is incorrect, the information in the alert can still be considered useful. For example, many false positives are due to the fact that the `mysql`-connector jar is flagged with the vulnerabilities of the `mysql` database server. Even though this is technically a false positive, since the vulnerability is not in the connector itself, the information that `mysql` is used together with an overview of existing vulnerabilities in this application is deemed valuable by the operators.

[2]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3700

VII. Threats to Validity

There is a risk of interviewer bias as a result of the fact that the author of the research is asking the questions. This risk is mitigated by the fact that interviewees may potentially use the proposed solution, therefore they will be inclined to give honest responses because their future way of working may be influenced by the outcome of the current study.

The selection of technical consultants coming from a single company limits the generalizability. In order to generalize the results beyond the research context, replication of the experiment using subjects from multiple organizations is needed.

Another threat is the fact that we have used only Maven-based Java systems. Our findings can be generalized to comparable technologies (Ruby when using Gems, .NET when using NuGet). These technologies also require dependencies declared in specific files before build time, and the way in which dependencies are declared is also similar – we input application names, which can be extracted and matched with CPEs.

Due to tooling limitations, the data we worked with may be unreliable due to false positives or false negatives. Where appropriate, this threat was removed through manual adjustment.

VIII. Related Work

Our research extends prior work on software product quality monitoring by Bijlsma et al. [4] which defined a process for detecting interesting events in streams of quality attribute metric values. However, instead of looking at metric values we define an interesting event to be the use of a component with a known security vulnerability.

Mitropoulos et al. [14] and Saini et al. [17] used the FindBugs static analysis tool to investigate the security of Java components to create bug catalogs and bug evolution trend reports [13]. Their work is mainly focusses on finding security vulnerabilities inside a Java component, while our work focusses on the detection of the usage of components with known security vulnerabilities.

IX. Conclusion

This paper presents a case-study in which we employed an automated vulnerability scanning tool in the process of external quality assurance. The tool itself is evaluated, as well as the usefulness of the tool inside the context of our host company. In short, this paper makes the following contributions:

- We empirically show that using components with known security vulnerabilities is common in practice
- We provide a description of a process to handle alerts addressing security concerns
- We evaluated the usefulness of vulnerability alerts in the context of external software product quality monitoring with positive results

In our view, future work revolves mainly around replicating our study in the same setting for confirming the findings, but also in a different setting, in order to understand the impact of the context on the conclusions drawn. Another element of future work relevant for our study are extensions to the contributions: more precise tooling, the handling of transitive dependencies, more and varied types of software systems, more heterogeneous interview subjects.

References

[1] CVE identifiers. http://cve.mitre.org/cve/identifiers/. Last visited 2014-01-19.

[2] OWASP - Using Components with Known Vulnerabilities. https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities. Last visited 2014-01-19.

[3] OWASP Top 10 Homepage. https://www.owasp.org/index.php/Top_10_2013-Table_of_Contents. Last visited 2014-01-19.

[4] D. Bijlsma, J. P. Correia, and J. Visser. Automatic event detection for software product quality monitoring. *Eighth International Conference on the Quality of Information and Communications Technology (QUATIC), IEEE*, pages 30–37, 2012.

[5] R. Bodenheim. Evaluation of the ability of the shodan search engine to identify internet-facing industrial control devices. *International Journal of Critical Infrastructure Protection*, pages 114–123, 2014.

[6] A. Buttner and N. Ziring. Common platform enumeration (cpe) specification. http://cpe.mitre.org/files/cpe-specification_2.1.pdf. Last visited 2014-04-30.

[7] M. Cadariu. Tracking vulnerable components in software systems. Master's thesis, Delft University of Technology, 2014.

[8] Shodan Search Engine. Jetty 6.1.1 keyword search on shodan. http://www.shodanhq.com/search?q=jetty+6.1.1. Last visited 2014-04-30.

[9] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 53–62, 1999.

[10] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. *Software Audit and Metrics*, pages 118–128, 2004.

[11] T. Kuipers, J. Visser, and G. de Vries. Monitoring the quality of outsourced software. *Proceedings of the International Workshop on Tools for Managing Globally Distributed Software Development (TOMAG 2007)*, pages 3–12, 2007.

[12] MITRE. Known vulnerability entry for jetty. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4610. Last visited 2014-04-30.

[13] D. Mitropoulos, G. Gousios, and D. Spinellis. Measuring the occurrence of security-related bugs through software evolution. In *PCI 2012: Proceedings of 16th Panhellenic Conference on Informatics (PCI 2012)*, pages 117–122, 2012.

[14] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis. The Bug Catalog of the Maven Ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 372–375, 2014.

[15] OWASP. Owasp dependency check home page. https://www.owasp.org/index.php/OWASP_Dependency_Check. Last visited 2014-03-13.

[16] S. Raemaekers, A. van Deursen, R. Schuppenies, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. *Proceedings of the Tenth International Workshop on Mining Software Repositories*, 2013.

[17] V. Saini, H. Sajnani, J. Ossher, and C. V. Lopes. A dataset for maven artifacts and bug patterns found in them. *In Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419, 2014.

[18] A. Van Deursen and T. Kuipers. Source-based software risk assessment. *Proceedings of the International Conference on Software Maintenance*, pages 385–388, 2003.

[19] J. Williams and A. Dabirsiaghi. The unfortunate reality of insecure libraries. *Aspect Security, Inc.*, March 2012.

[20] H. Xu, J. Heijmans, and J. Visser. A practical model for rating software security. In *Software Security and Reliability-Companion*, pages 231–232, 2013.