Empirical Study of Using Vulnerable Libraries in the Architecture of Chrome

Bladimir Baez and Hussein Talib
Department of Software Engineering
Rochester Institute of Technology
134 Lomb Memorial Drive
1+585-475-7829
(bb5100, hat6622)@rit.edu

I. INTRODUCTION

Nowadays, the growth of Internet has brought the beginning of a new era of technologies. This is the case of Internet browsers that have gained more popularity than in the past and have evolved tremendously over time. So, there is a diverse quantity of browsers available for the users such as Internet Explorer, Mozilla Firefox, Safari, Opera, Google Chrome, and others. Not only is there an increase in browser usage but also the security issue comes into play and Internet browsers can be affected. Indeed, any vulnerability may lead to leave enough room for attackers to take advantage of. Among different ways in which a browser can be vulnerable, there are the third-party extensions that, as mentioned in a previous work [10], may have security risks if they are not used in a proper way.

Speaking of Open Source Software (OSS) libraries, there is a tendency to provide more freedom so that the code can be reusable for others. Studies have demonstrated that OSS may include libraries that can contain some vulnerability signs. In some cases, a single vulnerable library component may not have a strong impact in the software.

Google Chrome is a web browser that was developed by the Google Inc. in 2008. Throughout the years, it has gained much popularity. Actually, this web browser is available for computers that use different types of operative systems such as Windows, MacOS, Linux; and devices that have Android, and iOS. On the other hand, the Chromium project is the open-source browser project that allows users to contribute at the source code level. The principal objective of Chromium is to build the more stable, safer, and faster web experience to the users. In its official website anyone can look for design documents, architecture overviews, testing information, and more. In order to avoid confusion, it is important to emphasize that Google Chrome is not open source itself. What makes it accessible, however, is the Chromium project in which user can access to the code, contribute, report a bug to be fixed, and so on.

In our study, the goal was to conduct an empirical research in the architecture of Chrome taking into account the usage of vulnerable libraries in the Chrome's architecture. To find any existing vulnerabilities, we used a dependency check tool called OWASP. In addition, we downloaded all the Google Chrome releases in order to be able to analyze the versions. This tool helped us identify all potential vulnerabilities that each version contains.

We implemented a model that runs on the Chromium project to make sure the vulnerability libraries that are defined in the National Vulnerability Database (NVD) contain the correct description. By choosing different Chrome versions, we used the tool mentioned to check any vulnerability in the database. We get the XML files that contain a list of vulnerabilities. Then, we determine whether a library has vulnerabilities or not.

We selected Google Chrome as the project to be studied because it is one of the most popular OSS. Also, there are plenty of resources available such as design documents, architecture overviews, testing information that can be facilitated through its official platform.

II. RELATED WORK

In the work [9], authors have proposed an approach to determine whether a vulnerability in the libraries can affect a particular project or not. If so, they suggest including a security patch in the software itself so the problems can be fixed. But, sometimes software developers do not keep in mind when a security patch can be applied. It may no affect when the system is in development or deploy mode. If it is applied when the software is in development mode, there will not be any serious risks. But when the security patch is included when a software is deployed and released to the users, there is a risk that the system has downtime whenever there is an update or change. To avoid this, it is recommended to make updates and include the security patch before the software is deployed.

One of the most popular projects, the Chromium Project for instance, may have vulnerabilities. Such vulnerabilities may be due to a bug found in the software, or any other factors that can be involved. However, in a previous work [4], researchers have pointed out that there is a weak correlation between bugs and vulnerabilities. They explained that there is a possibility that a bug can be injected without any intention before its release and it will not necessarily result in a vulnerability in the system. Instead, the authors mentioned that any mistake made by developers in the source code may lead to a vulnerability afterwards so that the attackers can take advantage of if they realize the flaw in the software.

Another factor regarding vulnerability issues is the fact that browser extensions, also know as "plugins", can contribute to more security leaks. For example, in Google Chrome there is a huge amount of extensions that facilitates user interactions on the web. One problem is that whenever a new extension is introduced, the risk of vulnerability becomes even higher compared to the ones that have been for a while.

It is important to take into account the purpose of some extensions. There are many that can be used as a way to introduce malware to the system. So, it makes it difficult to developers to address the external or internal factors. In addition, the compatibility of those extensions lead to another confusion due to the complexity that are associated with server-side templating languages. In 2009, according to a previous work [5], Google Chrome released an extension platform that can avoid and reduce extension vulnerabilities. The mechanisms included in the platform are *Privilege separation*, *Isolated worlds*, and *Permissions*.

In this paper [3], the authors present the security architecture of Chromium. It is said that the browser is split into two protection domains. There are the browser kernel, which is responsible for managing the operating system, and the rendering engine that runs with the protected privileges. The chromium's architecture is composed by several tasks or functions. First, it is the rendering engine. This component renders the web content by calling the DOM API and providing default behaviors. Then, the browser kernel which handles the different user operations through the browser such as saved passwords, bookmarks, cookies, and so on. Next, there is the process granularity, in which every instance is handle separately from the rendering engine. In order to display trusted content, a main exception called Web Inspector is integrated. And plug-ins that are components outside the sandbox and run in a separate host process. Authors in this work stressed that one of the challenges in evaluating the security of the Chromium's architecture corresponds to the fact that even though there are bugs in the implementation, its goal is to provide security. According to a survey, 67.4% of vulnerabilities would have occurred in the rendering engine if present in Chromium. In addition, 70.4% of the most severe vulnerabilities could be tackle.

The goal in [4] is to focus on architecture security risk analysis. When taking into account the security issues in the software at the beginning, it helps in discovering security risks, areas that security requirements are being affected, and any security-related flaws in the system at hand. Many efforts on security risk analysis depend on a set of techniques that are hard-coded or implemented in the analysis tool. Main problems are the absence of automated tools for analyzing those system architectures, lack of architecture evaluation criteria, and a limitation of the software capabilities details. They come up with a successful architecture security analysis schema that takes the important details of a system attack possible scenario. For the key entry they use "attack signature" which lists several variants that shows whether an architecture is vulnerable to an attack. So, Object Constraint Language (OCL) is used to get those signatures. By applying the idea, it was possible to apply the scenarios at source code, design and architecture levels. It is not convenient that security metrics are specified as ratio and percentage metrics since this allows misleading figures for the actual system security.

Similarly to the work [4], authors in [1] propose a hybrid approach that is about a model to identify software security risks in the early stages of the software development cycle. The model is based on a structural components factor and propagates any possible risk effects on a security to a higher level. In addition, they show the security risk evaluation model which is part of the SEF framework, which has some limitations when the risk-based assessment stage is in process. On the other hand, one of the main objectives is prepare software engineering with a metric that is able to quantify the security provided in the design phase in a given software system that lacks of security trust. However, further validation is suggested. So, for future work they plan to follow observations from industrial case studies to prove how practical is the risk model and the viability of the security trust indicator.

In this work [2], by mining the Issue Tracking System, authors try to present a generalized framework that characterize software maintenance projects that are stable using participation patterns. It is said that the vast majority of open source projects tend to fail because of low participation of contributors. From the Google Chromium Issue Tracking System, the conduct experiments on four years of data from 2009 until 2013. It was showed that at least every three months the 33% of contributors quit their roles in contributing in the open source systems. In addition, there is an increase in the workload in the contributors overtime. For this reason, contributors tend to participate even less, to the point that they decide not to continue participating at all.

III. Data Collection & Analysis

In order to be able to analyze all the Google Chrome versions, we downloaded them from the official Chrome project: *https://chromium.googlesource.com/chromium/src/+refs*. Since each version contained many sub-releases, we decided to choose only the latest from each one. For instance, from the versions "56.0.2912.2", "56.0.2912.1", and "56.0.2912.0" we only took into account the latest stable version which is "56.0.2912.2" from all labeled version 56, in this case.

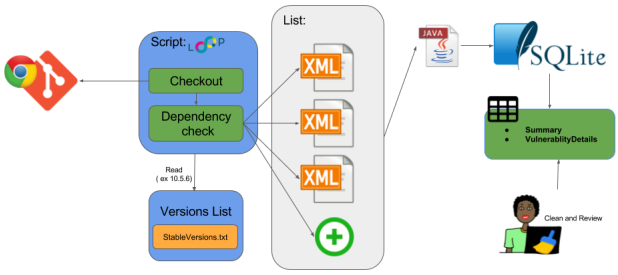

Fig. 1. Represents the approach followed.

The OWASP dependency-check tool helps identifying project dependencies and any vulnerable components, mainly third-party libraries. For the goal of our study, we focused on the libraries of the Chrome versions.

The way the dependency-check tool works, it first scans a project and tries to assign a CPE, which is described as the "Common Platform Enumeration", to each entity. It is important to know that there is much of heuristics involved.

When CPEs are assigned, it looks for vulnerabilities (CVEs) assigned to those CPEs. A scan is written to a result file. The tool supports multiple formats. The most common ones are the HTML (which is for direct reading) and XML (this one for further processing) formats. Teams use the XML output in order to process multiple reports of multiple projects and assign them to particular teams. (One team is usually responsible for multiple projects). At the time of this research project was conducted, the current version of the dependency-check tool was 1.4.4.

In this study, once the libraries were scanned by using the check-dependency tool, the number of dependencies encountered was 150, (147 of them being unique). Speaking of vulnerable dependencies, a total of 11 were found. On the other hand, regarding vulnerabilities, we found 714.

For each version, the tool detected the number of dependencies, number of dependencies scanned, and the number of vulnerabilities found. Also, we identified all third-party libraries that were contained in the Chrome versions. It was organized by its proper ID (which is the CPE number assigned), its name, and its description.

For example, there is a library from our generated dataset which summarizes the library vulnerabilities:

ID: "CVE-2016-6153"

ThirdPartyName: "sqlite3.dll"

ThirdPartyDesc: "os_unix.c in SQLite before 3.13.0 improperly implements the temporary directory search algorithm, which might allow local users to obtain sensitive information, cause a denial of service (application crash), or have unspecified other impact by leveraging use of the current working directory for temporary files."

Based on that, we identified any possible false positives that may appear in our dataset collected in order to make sure that it is a real problem. As you notice, the ID contains the identifier assigned to the specific "Common Vulnerabilities and Exposures". The *ThirdPartyName* represents the name of the library in which the vulnerability was detected. Then, *ThirdPartyDesc*, which contains the detailed description about the library vulnerability that was analyzed.

For our study, we used the dependency-check tool which extracts the list name of the stable Google Chrome versions from a text file. Then, it generates all the necessary XML files for each vulnerability. Once all XML files are generated, the most important tags are extracted such as its *ID*, *Name*, *Description*, and so on. Using Java, we generated the proper tables into a dataset for the SQLite database.

In order to make the process faster, we used a shell script that reads each Chrome version from the text file and generates the proper XML files so that the data can be stored in the database. Figure 2, shown below, is a snippet of the Shell script that was used for that purpose.

```
while read version_chrome

do

echo "checkout version: ${version_chrome}";

cd /chromium/src;

sudo git checkout "tags/${version_chrome}" -b "s${version_chrome}" ;

cd ../;

cd ../;

 echo "start dependency-check for version: ${version_chrome}";

dependency-check --project Testing --out . --scan  chromium/src --format XML;

mv dependency-check-report.xml "${version_chrome}.xml";

done < StableVersions.txt
```

Fig. 2. Shell script that reads each Chrome version names.

When the process is automated, it prevents any possible human errors that might occur while collecting the data in multiple versions. We, finally, reviewed and cleaned the code in order to have a readable report of the dataset. So, for the library detected, it was possible to find the library version.

Version	Library	Library version on last update
29.0.1548.1	Sqlite3.dll	3.5.4.1
29.0.1548.1	php5ts	5.2.6
29.0.1548.1	CygPCRE-0.dll	(No version)

As shown in the table above, there was a problem while trying to detect the correct file version. For example, the library *Sqlite3.dll* had 3.5.4.1 version on its last update when the Chrome version was above 29.0. So in those cases, the Google pull did not work well with *Sqlite3.dll* so we had to do the process manually.

IV. Results

We found a total of 15 vulnerable libraries. In total, there were 8,104 vulnerabilities encountered in those libraries.

The most vulnerable library detected was *OpenSSL.exe*, which appears to have 1,944 vulnerabilities among all the Chrome versions. Next we have *php-cgi.exe* and *php5ts.dll* with 1,674 vulnerabilities each. The, *LightTPD.exe* having a number of 1,026 vulnerabilities. As shown in figure 3.

Meanwhile, libraries such as *pom.xml*, *optipng.exe*, *gcm.jar*, were the less vulnerable among all Chrome versions according to the results.

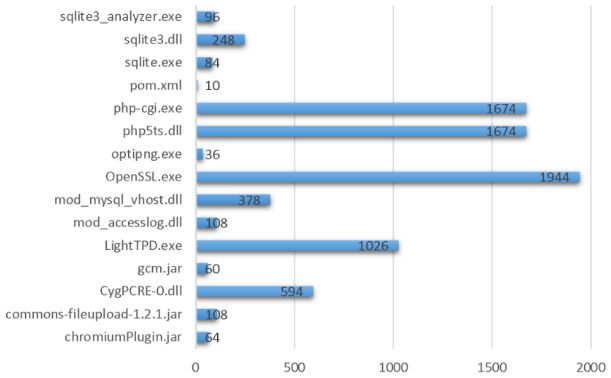

Fig. 3. Detected vulnerable libraries.

As it can be noticed in figure 4, we can clearly see that the first versions of Chrome such as 3.0, 4.0, and 5.0 were the most vulnerable than the subsequent versions. On the other hand, when coming to versions 15 through 24, the number of vulnerabilities are less compared to all the Chrome versions analyzed.

It is interesting to see how Chrome evolves over time. When a new version is released, the number of vulnerabilities decreases. However, it is not the case in all versions. If we take a look a the chart, after version 28 the numbers are increasing. It is mostly because they keep adding even more components, features in the newer versions that leads to the libraries to be more vulnerable in this case even though the older Chrome versions uses less components.

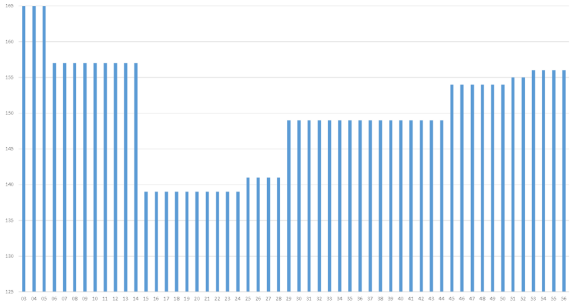

Fig. 4. Vulnerable libraries frequency in each Google Chrome version.

In figure 5, for each Chrome version and each vulnerable library, true and false detection are illustrated. If we take a look at version 5.0, it illustrates the different libraries found to be vulnerable and then the number of true and false positives.

A single bar represents the Google Chrome version, and within that bar, it contains all the vulnerable libraries found and the true/false positives. As an example, for *sqlite3_analyzer.exe*, the number of false and true positives are represented using different colors in order to be able to identify results with clarity. In this case, the light green for the true positives, and light blue for false positives on that particular library.

It is shown that the *commons-fileupload-1.2.1.jar* library is one of the most libraries validated to be vulnerable. Similarly, it is the case for the PHP libraries, such as *php-cgi.exe* and *php5ts.dll*. Since there are space and size constraints, the graph will be attached separately in order to catch every single illustrated detail.

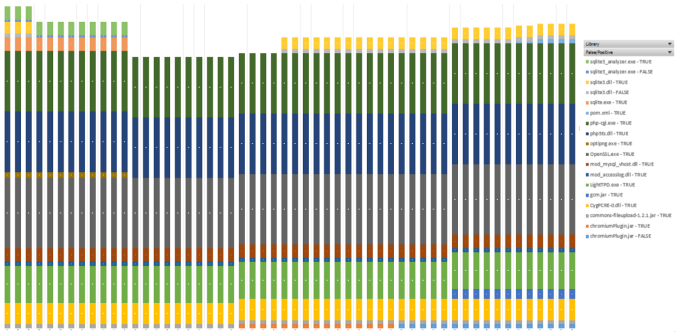

Fig. 5. Libraries with the number of true and false positives in each version.

The number of true positives detected is 7,999. However, for false positives we have 105. In this illustration (figure 6), libraries such as *sqlite3_analyzer.exe*, *sqlite3.dll*, and *chromiumPlugin.jar* have 12, 62, and 34 as false positives respectively. Meanwhile, the vulnerable library that has the highest number of true positives is *OpenSSL.exe*, followed by the PHP libraries (*php-cgi.exe*, *php5ts.dll*) which has 1,674 each. Then, the library *LightTPD.exe* in which the frequency of true positives was 1,026. Finally, the other libraries that have below 600.

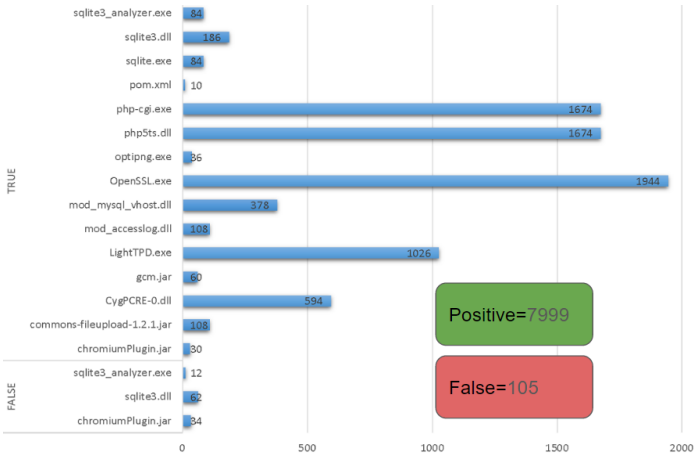

Fig. 6. False positives on vulnerable libraries.

Among all versions, in the last six months the vulnerability increases because the National Vulnerability Database (NVD) was not updated with the new types of vulnerabilities that were detected, as seen in figure 7.

In previous years, for those vulnerabilities present at the time, it was possible to validate whether it was a true positive or not. That is the reason why some new types of vulnerabilities are impossible to track on the NVD.

In this chart, figure 7, for each Chrome version the total number of vulnerabilities in all libraries is represented by two bars; one for the true positives and the other for false positives, and so on.

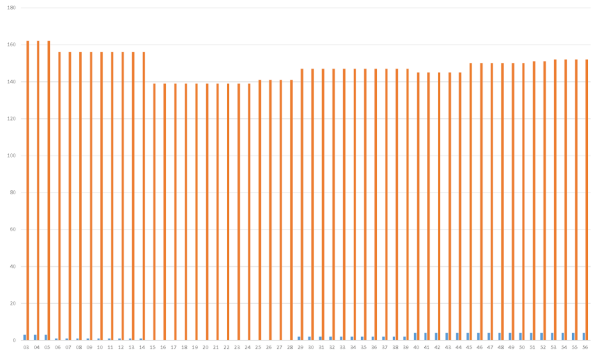

Fig. 7. True and false positives in all versions for all vulnerable libraries.

In figure 8, there is a representation of only libraries that were validated as false positives. As it can be seen, there were few libraries that were identified to be false positives. Those libraries are *sqlite3.dll*, *sqlite3_analyzer.exe*, and *chromiumPlugin.jar* even though in some versions they were identified as true positives.

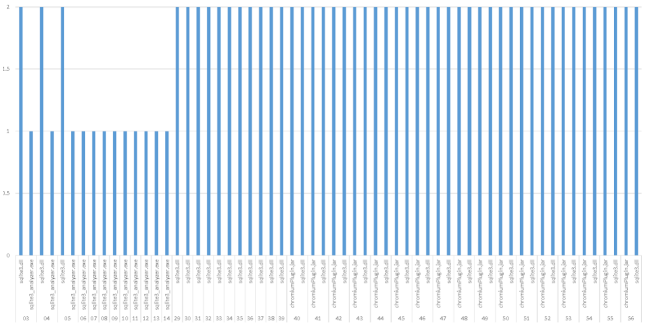

Fig. 8. Representation of all vulnerable libraries as false positives.

Regarding negative vulnerabilities that are considered as false positives, the recent versions of Chrome appear to have more false positives, as illustrated in figure 9.

When referring to negative vulnerability, we take into consideration the total count of false positives for each version, as well as the vulnerable libraries that were detected. In this case, the SQLite libraries (*sqlite3_analyzer.exe*, *sqlite3.dll*) and the one from Google (*chromiumPlugin.jar*). As it have been seen before on versions that have a large count of true positives, the latest versions are having the high number of false positives as well due to the fact that more components, features are used in more recent versions.

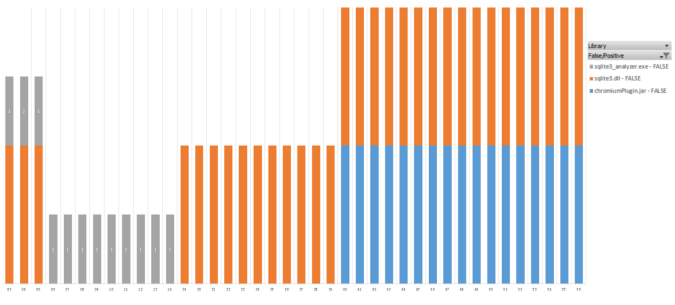

Fig. 9. Total count of negative vulnerabilities per version.

As presented in figure 10, we could detect that the new version have more vulnerabilities because they used more components as well. Some vulnerabilities were fixed by the time because they were part of Google. Others were not fixed since Google does not have control of them and they are third party libraries. So, each single bar contains the vulnerable libraries highlighted with different colors along with the total number of cases for each libraries that are vulnerable.

Throughout all versions, there have been libraries that have been present as vulnerable since a while. In most cases, those libraries begin to be absent in subsequent versions. There is the case of the *sqlite.exe* which only appears in the first 14 Chrome versions. On the other hand, libraries such as *gcm.jar*, was not present until the Chrome version No. 45 was released to the market, as seen in figure 10.

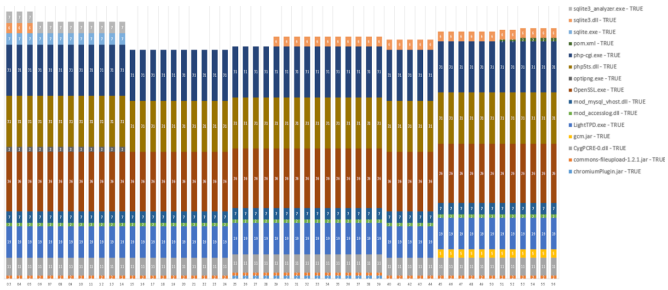

Fig. 10. Total count of positive vulnerabilities per version.

V. Challenges

While working in the research project, we faced many challenges at the beginning that led us come up with appropriate solutions to the problems that appeared. For instance, one of them was:

1- When running the script, we were not able to identify the version for some of the libraries found. In order to tackle this problem we had to update our script and run it again so that we could find and identify the versions.

2- At first, it was not possible to detect false positives in some libraries such as *Sqlite3.dll*, *php5ts*. But, after verifying each version we could find any false positives. In *Sqlite3.dll*, the version detected was 3.5.1; for *php5ts*, was 52.6, and so on.

3- We had never seen the OWASP dependency-check tool before. So, it took some time to be familiarized with the tool by reading the appropriate documentation in order to be able to use it properly.

VI. Conclusion

Some library vulnerabilities were fixed at time because they were published by Google (e.g. Sqlite.exe) while other vulnerabilities were not. It is because the others are third-party libraries from other companies and they are were not controlled by the Chromium team.

Another point is that the dependency-check tool cannot recognize new updates of some libraries such as *sqlite3.dll* that was treated as Sqlite.

In the study, the percent of correct description of vulnerability library is as follows:

- Total number of files (7,996 out of 8,104), which represents the 98%.

- Total number of libraries (12 out of 15), representing the 80%.

For the Chrome clone, it can be said that the library clone was manually verified. Finally, it is important to mention that the percentages presented above cannot be considered as the same for projects that uses new or different libraries.

All in all, the dependency-check tool have an excellent precision in finding the vulnerable libraries in a project, at least for the Chromium Project.

We have included a final report that includes the false-positives and true-positives, the vulnerability name, its ID, as well as a detail vulnerability description along with all the charts from this document.

REFERENCES

[1] A. Alkussayer and W. H. Allen, "Security risk analysis of software architecture based on AHP," *7th International Conference on Networked Computing*, Gyeongsangbuk-do, 2011, pp. 60-67.

[2] A. Rastogi and A. Sureka, "What Community Contribution Pattern Says about Stability of Software Project?," 21st Asia-Pacific Software Engineering Conference, Jeju, 2014, pp. 31-34.

[3] Barth, A.; Jackson, C.; Reis, C. "The security architecture of the Chromium browser." 2010-11-18)[2011-7-12]. http://seclah.stanfordedu/websec/chromium (2008).

[4] Camilo, F.; Meneely, A.; Nagappan, M. Automated Software Architecture Security Risk Analysis using Formalized Signatures. A Study of the Chromium Project. *12th Working Conference on Mining Software Repositories*, pp. 269-279 (2015).

[5] Carlini, N; Felt, A.P.; Wagner, D. An Evaluation of the Google Chrome Extension Security Architecture. *Security'12 Proceedings of the 21st USENIX conference on Security symposium*, (2012).

[6] Chromium Project, https://www.chromium.org/, 2016.

[7] M. Almorsy, J. Grundy and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," *35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, 2013, pp. 662-671.

[8] OWASP, Dependency Check tool https://www.owasp.org/. 2016

[9] Plate, H.; Ponta, S.E.; Sabetta, A. Impact Assessment for Vulnerabilities in Open-Source Software Libraries. *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 411-420 (2015).

[10] Swamy, N.; Guha, A.; Fredrikson, M.; Livshits, B. "Verified Security for Browser Extensions," *2011 IEEE Symposium on Security and Privacy*, Berkeley, CA, 2011, pp. 115-130.