INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Daniel Askeli

DATA COLLECTION FOR SOFTWARE SECURITY ANALYSIS

Master's Thesis
Degree Programme in Computer Science and Engineering
April 2016

Askeli D. (2015) Data collection for software security analysis. University of Oulu, Computer Science and Engineering. Master's thesis, 51 p.

ABSTRACT

There is a vast amount of data available on the Internet originating from multiple sources. Combining different sources is interesting as it can offer more information than separate sources. In addition current trends favoring open source projects and open information provide an interesting setting for security analysis.

However in order to utilize the data it needs to be harvested. In this work implementation of a document oriented time series data collection framework is presented. It provides features that make data collection easier compared to previously existing solutions. The framework is then used to collect data from two popular open source projects and relevant vulnerability data sources. The data is used to determine where in the source code the vulnerabilities locate and the locations are visualized. Results suggest that there is value to be gained from combining data sources.

Keywords: Data collection framework, Open Source Intelligence, software security

Askeli D. (2015) Tiedonkeruu ohjelmistojen tietoturva-analyysiin. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 51 s.

TIIVISTELMÄ

Internet tarjoaa valtavan määrän vapaasti saatavilla olevia datalähteitä. Näiden datalähteiden yhdistäminen on mielenkiintoista, sillä siten voidaan saada enemmän tietoa kuin erillisistä tietolähteistä. Lisäksi nykyiset avoimen lähdekoodin projekteja suosivat suuntaukset antavat mielenkiintoisen kehyksen tietoturva-analyysille.

Jotta dataa voidaan käyttää, pitää se kuitenkin ensin kerätä. Tässä työssä esitetään dokumenttisuuntautunut aikasarjadatan keräämiseen tarkoitettu ohjelmistokehys. Kehys sisältää ominaisuuksia, jotka tekevät datan keräämisestä helpompaa verrattuna aikaisempiin ohjelmistoratkaisuihin. Kehystä käytetään datan keräämiseen kahdesta suositusta avoimen lähdekoodin projektista ja niihin liittyvistä haavoittuvuusdatalähteistä. Kerättyä dataa käytetään haavoittuvuuksien paikan selvittämiseen, minkä jälkeen ne visualisoidaan. Tulokset osoittavat, että tietolähteitä yhdistämällä voidaan saada lisäarvoa tietoturva-analyysissa.

Avainsanat: Tiedonkeräysohjelmistokehys, Open Source Intelligence, tietoturva

TABLE OF CONTENTS

FOREWORD

I would like to thank my thesis supervisor Prof. Juha Röning for his guidance and patience during the writing process. I would also like to show my gratitude to all the folks at Oulu University Secure Programming Group, my fiancée Hilla, family, and friends for their continued encouragement and support. Also Thomas Schaberreiter and Christian Wieser deserve special thanks for their invaluable help and advice throughout the process of writing this thesis.

Oulu, Finland May 31, 2016

Daniel Askeli

ABBREVIATIONS

CPE	Common Platform Enumeration
CSV	Comma-Separated Values
CVE	Common Vulnerabilities and Exposures
CVE-ID	CVE Identifier
CVSS	Common Vulnerability Scoring System
ES6	ECMAScript 6th Edition
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
NVD	National Vulnerability Database
OSINT	Open Source Intelligence
RFC	Request For Comments
UDP	User Datagram Protocol
VCC	Vulnerability Contributing Commit
VCS	Version Control System
XML	Extensible Markup Language

1. INTRODUCTION

There is plenty of openly available data on the Internet originating from many different kinds of sources. The sources vary from unstructured ones like mailing lists and social media, which may require extra effort to harvest and analyse, to databases that provide information in easy to use, structured format. There is a trend of applications using these publicly available sources to extract and enhance information [1, 2, 3, 4, 5]. In addition, an another form of open information, open source software has gained ground among the regular users. This is most prominently displayed by the huge market shares held by open source software in the web browser and mobile operating system markets.

On the other hand in our ever increasing virtual presences, we trust software with our personal information and money, making it alluring for adversaries to try to get their hands on the information. This emphasizes putting effort into discovering and fixing vulnerabilities in the software to raise the barrier for creating exploits. Examples of vulnerability discovery methods are static code analysis, fuzz testing and manual analysis done by security experts. Static code analysis tools analyze source code and attempt to highlight possible issues. This is done without running the program and can be easily integrated to the development cycle of the project. Fuzzing is largely automatic, efficient, and excels at finding bugs caused by incorrectly handling input. In contrast manual analysis is labor intensive, but might reveal vulnerabilities that could not be found by other means. Researchers have also been experimenting with different machine learning based vulnerability discovery methods. However, those methods have not been able to gain footing in the vulnerability discovery scene [6, 7, 8].

The data collection framework implemented in this work was motivated by an interest in collecting data from multiple sources. By combining sources it is possible to study the relations between sources and get a greater picture of the information than it is possible from a single data source. The need for time series collection arises especially when the data source does not hold history information. In that case the data collection must be done periodically if history data is needed.

To address these needs the implementation of a document oriented time series data collection framework is presented in this work. Compared to existing solutions the implemented framework offers additional features that makes the collection easier by assisting in scheduling and filling in missing metadata.

Chromium and Android projects were selected for a case study based on combination of a few factors. First and foremost the projects needed to be open sourced or the development information available would be very limited. Secondly the size, security impact, and availability of vulnerability reports were assessed. All of those contribute to the amount of security reports the project has. Lastly Git was the favored version control system (VCS) due to its popularity among open source projects. As such the methods used in this work are easier to apply to other projects, which use Git. Chromium and Android fulfill these criteria well.

The framework is used to collect data from the projects' development activity and vulnerability information. The combining of sources allowed distinguishing the security related code changes in the repository. This was used to map the code locations touched by security issues and the results were visualized. In addition, a method for automatically determining the changes that caused the issues was experimented with.

The research question this thesis aims to answer is “How to leverage the openly available data sources for security analysis of a software project?” The process of tackling this question can be split into three parts. First step is to select the software projects to inspect and to identify the corresponding data sources. Section 2.4 introduces the selected projects and data sources. After that the used technologies, the data collection framework, data collection, and a method used for mapping vulnerabilities to code locations are presented in Chapter 3. Chapter 4 will present results of data collection, visualizations, and analysis done. Chapter 5 discusses the reliability of used methods and some ideas for future work.

2. LITERATURE REVIEW AND RELATED TECHNOLOGIES

The objective of this work was to collect time series data from open data sources to be used in software security analysis. The existing research and the state of the art in Open Source Intelligence (OSINT) and vulnerability discovery will be presented. In addition, some existing time series data collection solutions will be introduced.

To get tools for tackling the research question literature of different applications utilizing open information and the state of the art vulnerability discovery methods were studied. A few different open data collection applications are discussed in section 2.1. In section 2.2 the related work on vulnerability discovery is presented. Sections 2.4.2 and 2.4.3 will introduce the open source projects selected for the case study. The security significance of each project is discussed and the identified vulnerability sources for the projects are presented.

2.1. Open Source Intelligence

Ministry of Defence of Finland defines OSINT as "Acquiring information from public sources such as literature, maps, print media, public documents and websites" [9]. As such the OSINT predates Internet, but the emergence of Internet has accelerated the speed at which data is generated and made the data easy to access globally. Especially the vast amount of user generated data from for example social media, has accelerated the amount of data generated each day. A common problem in utilizing OSINT is that in many sources the data is in unstructured format and might require natural language processing methods to extract the information, which in itself is a source of error. An example of such methods is to analyze messages in social media to extract indicators of status or sentiment of the author.

The usage of OSINT has been proved to be valuable in real world applications for example in getting situational information or to analyse trends. Sail Labs Media Mining System is an example of a system which makes use of freely available information. It aims to allow accurate situational analysis of crisis locations by analyzing different relevant data feeds. It gathers information from multiple sources including television, radio and various Internet sources and uses data mining techniques to extract information about the content and also metadata indexes this information to allow analysts to query the dataset.[1]

In [2], Aramaki et al. present a system that analyzes Twitter messages to find influenza-related contents to early detect influenza outbreaks. Authors also propose methods for filtering out messages which seem to mention disease, but whose author does not become ill in reality.

Open source software is one form of OSINT. The VCSs used in software projects are a rich information source of the development of the projects. Studies using machine learning and visual methods to analyze open source software show that it is possible to add value for projects management by extracting metrics from developer activity and predicting the development [3, 4, 5] or using the development history to trace back which change introduced a vulnerability [6, 7, 10].

Weicheng et al. inspected the relation between developer activity and major version releases. Models for source code dependency and developer activity were presented

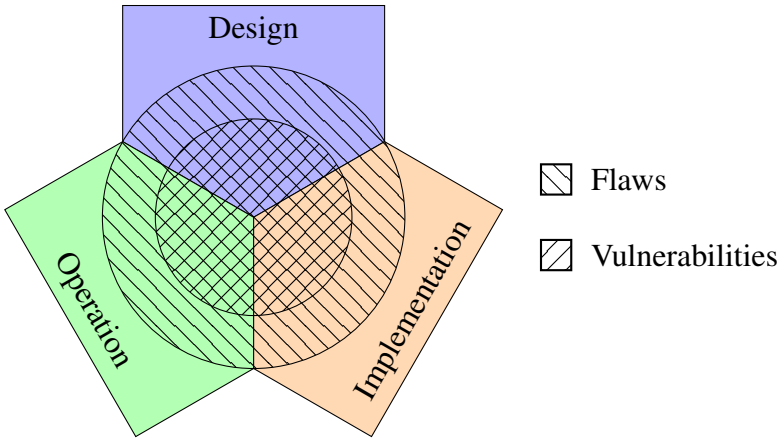

Figure 1. Locations of flaws and vulnerabilities in a system.

and applied to a case study of eight Github projects. Authors observed that changes in files predict future changes in dependable files. [3]

Ben et al. studied behavior patterns of the contributors of the open source projects and found a linear relationship between developers' early and later contribution. More over they found that contributors who started by contributing to a code with lots of recent changes tend to have smaller overall contribution. [4]

In [5] Gousios et al. presented a way to evaluate developer contribution. In addition to measurements from the source code repository they used the mailing list, wiki, and Internet Relay Chat activity as data sources in the study.

2.2. Vulnerability Discovery

Internet Engineering Task Force's (IETF) Request For Comments (RFC) 4949 defines vulnerability as "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy" [11]. Firstly the definition requires that a flaw can be exploited in order to be classified as a vulnerability. This means that after the finding of a flaw, that looks like a vulnerability, its context must be evaluated to determine if the flaw can be exploited. Proving the exploitability can be done by developing an exploit for the flaw or by analysis that determines the flaw as probably exploitable. However, proving that a flaw cannot be exploited is hard as even if the analysis was perfect a change in environment or a discovery of a new exploit method could make the flaw exploitable in the future.

Secondly the definition splits vulnerabilities into three types by the location of a flaw in the development cycle. Figure 1 shows the three different phases in a lifespan of a software and illustrates that vulnerabilities are a subset of flaws. Vulnerabilities caused by flaws in the implementation can also be referred to as security bugs.

2011 CWE/SANS Top 25 Most Dangerous Software Errors lists common causes of software vulnerabilities [12]. It also suggests prevention and mitigation methods for each type of errors. The listing offers multiple ways of avoiding the errors in different parts of the development. The listing ranks "Improper Neutralization of Special Elements used in an SQL Command" as the most dangerous software error. The listing

suggest a variety of mitigations for this error. Reflecting the mitigations to Figure 1 gives an idea of the concerns in the different parts of the lifespan of the software. For example it suggests to choose well vetted components when designing the software. In implementation it is suggested to assume that all input is malicious. Furthermore it suggests an operational mitigation, a firewall, which detects these types of attacks.

There are many different methods to approach the vulnerability discovery but no single best method for discovering all types of vulnerabilities. Most frequently used methods in practice are fuzz testing and manual expert analysis. In general fuzzing means feeding the targeted system randomized and possibly malformed inputs in order to trigger unexpected behavior in the system thus exposing the errors in the program.

However, fully randomized inputs are likely to be rejected at an early stage in the execution. To remedy this there are different methods for more intelligent fuzzing. For example Viide et al. developed system for interfering models from training files to assist fuzzing [13]. The benefit of fuzzing is that it is easily automatized and the testing can be done by harnessing an army of computers to test the system. The manual expert analysis is somewhat opposite of fuzzing as it is labor intensive, however it leverages the experience of the analyst and might reveal complex vulnerabilities that are not detected by fuzzing.

Software security is related to software robustness as both are interested in flaws that lead to unintended functionality. Software robustness is concerned about flaws in the program that affect the stability of the program. In contrast software security is interested in flaws that can possibly be exploited to for example redirect control or disclose information, in other words vulnerabilities. The difference in these settings is that bugs are much more common and only bugs that can be exploited are vulnerabilities. It has also been found that usually only a small portion of the code base is vulnerable. Therefore directing the available resources to most probable vulnerability sources is a good way to increase the effectiveness of the vulnerability discovery. A common method to approach automatic software vulnerability detection has been borrowing methods from software reliability modeling [8].

In [14] Alhazmi and Malayia experimented with the performances of static methods of estimating the number of vulnerabilities from the vulnerability data of the predecessor of the system, Alhazmi-Malaiya Logistic model and Linear Vulnerability Discovery model were evaluated by fitting models to three projects. Testing showed that Linear Vulnerability Discovery model performs well in systems that are yet to reach saturation in the vulnerability finding. The scope of the study was more managerial as the aim was to estimate the total number of vulnerabilities found in the software. Their work did not make any attempt to locate the vulnerable parts of the software.

In [6] Nagappan et al. studied correlation between code complexity measurements and post-release defects. Many object oriented measurements are presented such as number of classes, coupling of the classes and inheritance depth. After eliminating multicollinearity by principal component analysis predictors were constructed. Testing showed that these predictors were able to predict post-release defect prone modules from projects defect history in a random split test. They also observed that applying predicator constructed for one project to other was not successful in general case, but can be useful with similar projects.

In [7] Neuhaus et al. presented a system which maps vulnerabilities to software components and identifies patterns in the vulnerable components. Feature patterns were

then used to predict vulnerable components in the code base. Authors hypothesized that vulnerabilities are caused by the domain of imported libraries and used functions. The case study on Mozilla showed promising results as they were able to predict half of the vulnerable components with a two thirds of positive predictions actually containing vulnerabilities. These results hint that some parts of the source code are more prone to vulnerabilities and therefore methods helping to identify the vulnerable parts could benefit the efficiency of vulnerability discovery methods.

In [15] Zhang et al. used vulnerability data available in NVD to constructed models for predicting the time until the next zero-day vulnerability for a product is discovered. Authors found the data from NVD insufficient for reliable prediction, as it often contained inaccuracies or missing data. In addition, the tendency of vulnerabilities to affect also the older versions of the software made the prediction harder. Authors also noted that vendors have different reporting habits that affect the release date of vulnerability reports. This hides the real discovery dates of the vulnerabilities making accurate predictions impossible.

In [16] Glanz et al. argued that vulnerability reports in NVD define affected software versions in a way that lacks structure and is not exhaustive. They proposed an approach to enhancing the vulnerability data by adding information of vulnerable and fixed versions in a structured machine readable form. Authors also propose a method for filling this data out automatically for the pre-existing entries in the database.

In [8] Shin and Williams studied whether fault prediction models can be used for vulnerability discovery. They found that the fault prediction model gave similar recall and accuracy as the vulnerability prediction model for Firefox 2.0. Authors found recall satisfying but they propose that further work is required to improve the accuracy of models. They also hypothesized that fault prediction is more accurate than vulnerability detection because faulty files are much more common than vulnerable files. Hypothesis was tested by randomly removing faulty files from test set till the proportion was similar to vulnerable files. Test showed that in such conditions accuracy of fault prediction dropped to the level of vulnerability discovery thus proving that result is related to the needle in haystack condition.

In [10] Fonseca and Vieira studied causes of software vulnerabilities in six web applications. The study showed that majority of the cross-site scripting and Structured Query Language injection vulnerabilities are caused by eleven fault types of the 62 defined in [17]. They also define an extended version of the missing function call fault defined in [17], which allows the use of the return value of the missing function later in the code, to account for vulnerabilities caused by for example missing sanitation of the variables. The missing function call extended fault type was found to be the cause in 76 % of all the vulnerabilities.

In [18] Brumley et al. presented a proof of concept work which demonstrates that it is feasible to automatically generate exploits from software patches in just minutes. Authors argued that because of their results everyone who has received a patch to a vulnerability, should be assumed to be able to generate an exploit to the patched vulnerability. For this reason authors criticize staggered patching schemes where users receive the patch at different times, because it gives a time frame where an attacker who receives the patch early can generate an exploit and attack users who have not yet received the patch. Results of this work indicate the value of the software patches intrinsic property of disclosing the locations of the errors they aim to fix. By identifying

the fix it is possible to work back to the vulnerability present in the previous software version.

In [19] Avgerinos et al. presented an automatic end-to-end exploit generation system, thus demonstrating that such system is feasible. Authors also argued that source code level analysis is insufficient for detecting vulnerabilities because bug found in source code must be analyzed also on a binary level to determine if it is exploitable and thus a vulnerability. Authors argued that their work shows that exploit development which has been thought to require highly trained attacker is no longer true. However, automatic exploit generation can also be used on the defense by training the defense systems with the generated exploits.

In summary the studies using machine learning for vulnerability discovery present some common challenges [6, 7, 8]. Discovered vulnerabilities are relatively rare. This leads to datasets which are too small for developing robust machine learning models. On the other hand many studies construct the training data by hand, which limits the scalability. In addition, the studies in [15, 16] point out some shortcomings of NVD as a data source. However, these studies give a good indication of the kind of data usually used in vulnerability discovery methods. Automatic method for mapping vulnerabilities to source code locations is presented in this work, but the results are not accurate enough to be used as a basis for vulnerability discovery. However, if the automatic mapping procedure would be accurate enough it could be used to generate training datasets for machine learning. Finally the two automatic exploit generation studies highlight the lowered threshold for generating exploits [18, 19]. This makes discovering and fixing vulnerabilities even more important as the cost of developing exploits can not be assumed to be high anymore.

2.3. Existing Times Series Data Collection Solutions

Time series database is a system which is optimized for handling time series data, which is basically arrays of numbers. Traditional examples of time series data are stock prices and temperature graphs. However, this concept of storing only numbers is very limiting when capturing more complex phenomena. In this work document oriented time series database is used to refer to a data collection system which captures abstract data in a way which emphasizes the temporal aspect of the data.

The designing of a document oriented time series data collection system, which collects data from multiple sources, requires considering a few things. To emphasize the temporal aspect of the data, each data point needs to include the time of the collection. In addition, in order to distinguish between different sources metadata identifying the data source is preferable. Another feature to make the data collection more flexible is to prefer document-oriented database design instead of more traditional relational databases as the structure of the collected data might not be known beforehand or the data is by nature amorphous. Some of the other data collection frameworks matching these considerations are introduced below.

Cube is an open source document oriented time series data collection and analysis system implemented on Node.js and it uses MongoDB as the database back end [20]. The main design of Cube is that it collects “events” which are JavaScript Object Notation (JSON) objects containing fields for type, timestamp, and data, which is arbitrary

JSON object containing the event data. The Cube system consist of three components: Emitters, Collector and Evaluator. Emitters send the events to collectors and Evaluator is used to querying and post-hoc analysis of the collected events. Collector supports sending events send through User Datagram Protocol (UDP), Hypertext Transfer Protocol (HTTP), WebSockets and Emitter supports queries via HTTP and WebSockets.

Seriesly is an open source document oriented time series database written in Go [21]. Like Cube it stores data as JSON objects enriched with timestamp, however no type metadata is included by default. Data collection is done using HTTP to send the JSON object to the database and querying the data is done using HTTP. Seriesly also offers option to do some aggregation on the data at the query time.

The Cube and Seriesly focus solely on storing of collected JSON objects and leave the actual collecting, sending and scheduling of the collection to the responsibility of the user. The system presented in this work offers easy scheduling of the collection and simpler collection scripts as no networking by the collector script is required. Chapter 5 provides further discussion between the framework implemented in this work and pre-existing solutions.

2.4. Open Source Projects and Data Sources

This section introduces the OSINT data sources used in this work including general vulnerability data and the selected open source projects and the project specific data sources. Firstly the standard terminology used when communicating about vulnerabilities is introduced as well as the general vulnerability data sources. In sections 2.4.2 and 2.4.3 Chromium and Android open source projects and the selection of the projects are explained. Also the vulnerability data used in each project is discussed.

2.4.1. Common Vulnerability Data Sources

Common Vulnerabilities and Exposures (CVE), Common Platform Enumeration (CPE), Common Vulnerability Scoring System (CVSS) are industry standards for defining vulnerability critical information and to establish common names for referring to entities between separate organizations [22, 23, 24]. Following paragraphs will go in more detail on the usage of the concepts and what kind of information they hold. National Vulnerability Database (NVD) is also discussed as it is commonly used source for the CVE listing with additional analysis provided for each entry [25].

CVE is a listing of publicly known information security vulnerabilities and exposures. CVE Identifiers (CVE-ID) contain CVE Identifier number (e.g. CVE-2014-0160), which includes the year of assignment and unique four to seven digit unique identifier for that year, a short description and possible references. Purpose of CVE is to provide names for vulnerabilities and exposures which can be used to commonly reference them between different actors. [22]

In the same vein as CVE, CPE standardizes naming used in the industry. CPE defines rules to construct uniform resource identification for different platforms by their relevant components. CPE has three main categories for platforms defined by their level namely application, operation system and hardware. For example

```
cpe:2.3:a:google:chrome:42.0.2311.107:*:*:*:*:android:*:*
```

is the CPE for a version of Chrome browser running on Android. More precisely the used CPE version is defined in the beginning, “a” denotes that it is an application, vendor name is Google, product name is Chrome, version is 42.0.2311.107, and Android is the architecture surrounding the product. [23]

CVSS is a standard for determining a severity score for a vulnerability by assessing its characteristics. The score is constructed by assessing for a vulnerability a base score which can be then modified by temporal and environmental impact characteristics. The base score ranges from 0.0 to 10.0 and it considers the immutable characteristics of the vulnerability, such as the possible impact and the difficulty of an exploit. Temporal modifier considers available exploits, mitigation measures and the confidence of the vulnerability report. Environmental modifier allows users to adjust the score by taking into account how their own environment might increase or decrease the severity of the vulnerability. [24]

NVD provides a database and analysis for CVEs. In addition to hosting the CVE listing, NVD aims to assign each CVE with a CVSS base score. NVD publishes the vulnerability data in Extensible Markup Language (XML) format and also provides two data feeds one for vulnerabilities and the other for vulnerabilities whose analysis has been finished [25].

There are also other interesting data sources that were not used in this work. CVE Details is a vulnerability database which provides the CVE listing combined with the data from NVD and in addition adds data of know exploits for the vulnerabilities [26]. Open Sourced Vulnerability Database is a project collecting vulnerability data from open information sources such as security mailing lists and vendor websites and claims to have enriched information about CVEs and to include vulnerabilities which have not been assigned a CVE [27].

2.4.2. Chromium

Chromium Project maintains the source code of the web browsers which is, with minor modifications and combined with Google products, published as the Google Chrome web browser [28]. It can be argued that web browsers are one of the most security relevant software as they are used in online banking, and social media sites. At the same time ever increasing amount of applications are being implemented in the browser. Moreover, a browser can easily be exposed to malicious web pages for example by a careless click of a link, making the browser the only protective layer between the user and the attacker. As such the security of browsers is of utmost importance which makes them an interesting target for a study.

Chromium Project is a good case study target for several reasons. Chromium Project is open-source which makes it possible to analyze the source code repository. Chromium uses Git as its VCS, which was one of the interests in the study as the Git is widely popular among open-source projects. Before Chromium project migrated its VCS to Git in 2013 [29] it was using Apache Subversion for version control. However, in the migration the development history was ported into Git. In addition, the issue tracker of the project allows filtering issues categorized as security related by the

developers and the commit practices of the project allow easy mapping from issues to commit fixes. In addition, Chromium Project is adequately sized to contain reasonably amount of vulnerabilities.

Vulnerability Information

The issue tracker of the project was used as the data source for vulnerability information. The changes in the code base of the project often refer to the related issues, but the nature of the issues is not easy to determine from the change. The issue tracker on the other hand offers rich information about individual issues. The categorization of the issues can be used to identify security related issues, which in turn can be used to find the security related code changes. In addition, the data in the issue tracker is structured which makes it easy to use.

The issue tracker was the only vulnerability data source, which was feasible to implement with a high enough automation in the context of this work.

2.4.3. Android

Android is an open source operation system based on Linux kernel for mobile devices developed by Open Handset Alliance led by Google. It is the most popular mobile operating system reportedly holding 82.8% of the mobile market [30]. The project was made open source in 2008 [31], allowing the vendors develop their own customized versions of the operating system for their products.

The operating system is the software in control of the device and thus good security is important to raise the bar for the attacker to take over the device. Nowadays most of the people are carrying mobile devices with the same capacity as personal computers, hold the personal information of the user, and pack a bunch of sensors that can be used to spy on the user if the device gets compromised. Mobile devices also have many different connections and by their mobile nature are exposed to many unknown devices making the attack surface much larger than on a regular desktop computer.

Android source code repository consists of multitude of Git repositories that hold the different projects utilized in the operating system. Repo is a tool build for managing the multiple Git repositories of the Android repository and to automate some parts of the Android development workflow [32]. The Android source code repository is huge taking over sixty gigabytes of hard drive space as it holds the long development histories of multiple projects. More detailed explanation of Git and Repo will be provided in the section 3.4.

Vulnerability Information

As in the case of Chromium the mapping of software changes associated with vulnerabilities rely on commit messages mentioning the related identifiers. In practice development history will be searched for mentions of CVE-IDs and found references will then be enriched with the related data from querying the NVD.

Like Chromium, Android has an issue tracker but the conventions of the project for classifying issues differed from Chromium so that the tracker had only a handful of

security related issues and even those were of poor quality. However, during the process of making this thesis a severe vulnerability in the multimedia framework library Stagefright was discovered [33] and issues related to it also made appearance in the issue tracker of Android. This could mean that issue tracker would become a valuable information source for Android vulnerabilities in future. However, the issue tracker was not used as a data source for Android in this work, because of the shortcomings described above.

3. IMPLEMENTATION

The aim of the implemented data collection framework is to provide a system which allows the user to setup document oriented time series data collection by providing a collector and minimal configuration. The scheduling, adding meta data, and communicating with the database are taken care of by the framework so collector is only responsible of generating the data point which makes starting data collection easier than in the existing time series data collection solutions presented in the Section 2.3.

After the implementation of the framework, collectors for harvesting data from source code repositories of the Chromium and Android, and selected vulnerability data sources were implemented. The data collection framework was configured to use the collectors to setup a data collection. Moreover, a visualisation application was created to make use of the collected data providing a heatmap visualisation of the structure, activity and vulnerabilities of the project.

In this chapter the details of the implementation of the framework and case studies as well as the technologies used are introduced. The framework including the collector coordinator and simple database are implemented in JavaScript running on Node.js [34]. Scripting of the actual collectors is done in Bash shell scripts and Python scripts. The visualisation is implemented as a simple web application utilizing a popular data visualization framework D3.js [35]. The development was done on a Linux Mint 17.2 operating system. In addition, the framework was deployed on a virtual machine running Debian operating system.

3.1. JavaScript Object Notation

JSON is a commonly used human readable data format which is as the name suggests build upon the object notation of JavaScript [36]. However, JSON itself does not have any bindings to JavaScript and thus it can be used with any programming language. In this work JSON is used as the data format in the interfaces between components as well as in storing the data. In addition, it is used for storing and defining configuration settings for the components.

JSON has two basic structures: object which is an unordered collection of the key-value pairs and an array which is ordered collections of values. Value can be an object, array, string, number, “true”, “false”, or “null”. Figure 2 demonstrates the JSON syntax and different value types. Objects are delimited by curly brackets and arrays by square brackets. Values and key-value pairs are separated by commas and key and value by colon. White spaces between elements are not required but are useful to help readability. In the example different value types are demonstrated as well as the fact that arrays and objects can contain values of any type. The example presents also nesting of objects and arrays which easily enables presenting hierarchical data.

The benefits of JSON are that it is human readable, simple, and flexible. The flexibility shows up in the somewhat self-defining nature of the JSON objects as all the fields are named and no schema knowledge of the data is required when using the objects. However, these benefits come at the cost of the storage efficiency of the format as the keys and syntax include redundant data especially in the case where the values the object holds are known beforehand. For example interesting pieces of information in

a JSON object `{"name":"Bob", "age":30}` are a string, “Bob” and the integer “30” and all else is extraneous when we assume that the type of the contents are known from the context, however the JSON notation gives us an idea what this data is about without any external knowledge.

3.2. Node.js

Node.js and JavaScript were chosen for the implementation of the framework, because of their asynchronous event driven nature, which lends itself easily for coordinating subprocesses and is easily exposed to a browser based application. Node.js is build upon V8 JavaScript engine of Google Chrome.

The implementation was written in ECMAScript6 (ES6) [37] and transpiled to JavaScript using `babel`[38], which is a tool for transpiling ES6 code to JavaScript to allow the code to be ran on platforms that do not support ES6. ES6 brings multiple handy features to JavaScript such as arrow-functions, strict-mode, generators and promises. Even though those features have since been implemented into V8, `babel` was needed in the implementation during the work, as the features used were present at the start of the work.

The framework uses two nonstandard Node.js libraries `ws` and `node-schedule`. `ws` is a simple websocket library which is used for implementing a websocket interface for a database and in the communication between the database and the visualization web application. `node-schedule` is a library providing cron-style scheduling for tasks.

`node-schedule` is a Node.js module implementing cron-style scheduling with an extension of possibility of defining run intervals down to a second precision versus the minute granularity found in traditional Cron. Cron is a popular and widely adopted time-base scheduling tool for Unix-like operating systems used to run commands periodically. Figure 3 illustrates fields and their value ranges in the Cron-style time format used in `node-schedule`. Cron uses originally five fields: minutes, hours, days of the month, months and days of the week. `node-schedule` uses extended notation which includes an option of defining six fields instead of five in which case the leftmost field is interpreted as seconds. The fields are separated by white space. Each field can have “*” as matching all values, a range of values defined with hyphen (e.g. “1-5”)

```
1  {
2      "string": "Strings are delimited by quotation (\") marks.",
3      "numbers" :
4          {
5              "integer": 42,
6              "decimal": -3.14159,
7              "scientific": 1.234E9
8          },
9      "literals": [true, false, null],
10     "arrays": ["a", 1, ["b", 2], {"c": 3}]
11 }
```

Figure 2. A JSON object demonstrating different value types and nesting.

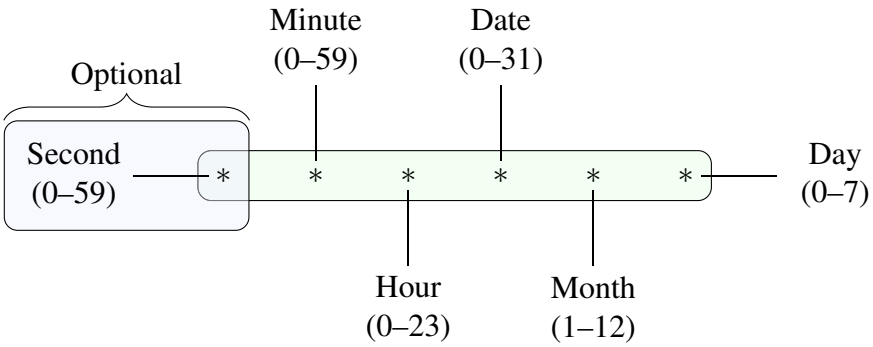

Figure 3. Cron-style format used in node-schedule including optional second precision.

or a list of values separated by commas (e.g. “1,4,8”). In addition, a desired interval within a field can be denoted in the style of “1-20/5” meaning the same as “1,6,13,19”. The scheduled task is ran when the time matches the time string. For example “15 30 4 * * 3,7” defines six fields and it means that the task is to be ran each third and seventh day of the week, 04:30:15 in the morning. The “* * * * * *” in Figure 3 means that the task is to be ran every second.

3.3. Scripting

As the collectors used in this work are fairly simple and are not performance-critical they can be simply implemented using general scripting.

Python scripts are used for the more complex parsing tasks as parsing tree structures from a file listing and for parsing JSON objects from the Git output. Main reasons for choosing Python was its wide availability and expressive power. The biggest drawback of Python is its poor performance compared to many other languages. However, this was not a problem in this work as the parts implemented in Python were not performance-critical.

Bash shell scripts are used to “clue” together different programs and Unix utilities. One convenient feature of bash scripting is the ability to easily “pipe” output from one program to the input of another. For example the command:

```
git ls-tree -r --name-only HEAD | sed "s:^:\$REPO_PATH:"
```

redirects the directory listing from `git ls-tree` to a Unix utility `sed` which inserts the contents of `$REPO_PATH` into the beginning of each line. Also the tracing of fixes to the blamed commits presented in section 3.8 was written as a Bash shell script to automate the process.

3.4. Git and Repo

Git is a popular distributed VCS which is being used by many large companies and projects such as Google, Facebook, Microsoft and Linux kernel [39]. More importantly

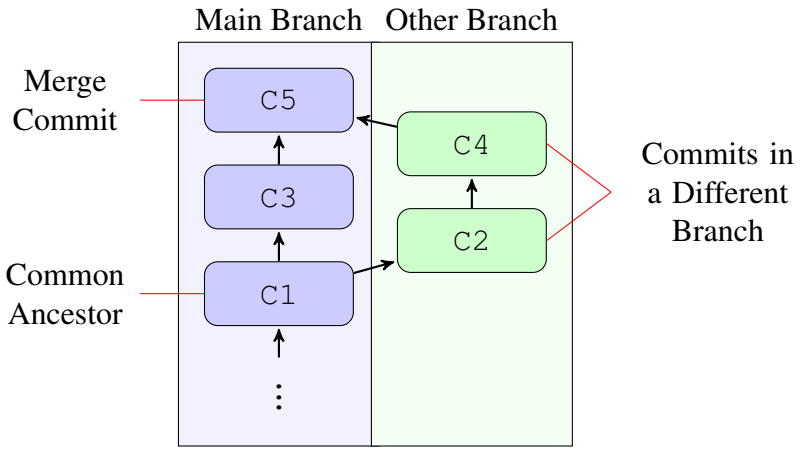

Figure 4. A simple Git branch and merge scenario.

it is used in the projects selected for this study. Understanding the basic Git concepts is also important as they are heavily used in the implementation of the collectors.

Git being distributed means that every user who clones a repository actually gets a local copy of the whole project. This offers performance improvement when compared to centralized VCSs on a remote server as most operations can be done locally thus eliminating the latencies introduced when communicating with remote server. [40]

A commit is a basic unit of work stored in a Git repository. When file changes are committed Git stores a snapshot of the project in an intelligent manner. Git keeps track of every commit made to the project which allows easily to revisit or revert to any earlier version of the project or more complex functionality such as tracing back who made a specified code change. [40]

Branching is a common VCS feature which allows grouping up related work. Different workflows using branches can for example have long running branches for release and development versions of a project or having short lived branch which implements a new feature or fixes an issue. In Figure 4 commit `C1` is branched using `git branch` tool to work on something while the work on the main branch continues in `C3`. The commit `C4` finishes what was being worked on and `git merge` tool is used to join together the development histories of the branches generating the merge commit `C5` thus introducing the work done in the other branch to the main branch. It is important to notice that chronological history of a branch can seemingly change as merging can introduce commits that are older than the last commit in the branch. In the case of Figure 4 before the merge commit `C5` the chronological history from new to old would read as `C3`, `C1` when after the merge the history would be `C5`, `C4`, `C3`, `C2`, `C1` introducing the commit `C2` between `C3` and `C1`.

Repo is a tool used in managing the Android repository. Repo is build on Git to simplify working with the repository. Repo provides functionality to download and update the Android source code repository as well as to work on the repository and to publish modifications to Androids review system. In this work Repo is used for updating the repository and for executing commands in each Git repository contained in the Android repository.

To get an idea of different operations that can be performed by Git the following rundown describes the utility of the most important Git tools used in the implementation:

- `git log`
  A tool for displaying the commit history of the repository. The most important parameters include options for filtering commits by matching contents against a regular expression or to limit the displayed history by a time range.

- `git blame`
  A tool for inspecting which revision last modified given parts of the source code. As the name suggests it can be used to put the blame on person who introduced for example vulnerable code to the repository.

- `git checkout`
  A tool which updates the files in the working directory to match the project state as defined by a branch or a commit. When given a commit as a parameter files are updated to correspond the project state after the commit.

- `git ls-tree`
  A tool that lists the contents of the Git tree object, which can be used to get file listing at a given commit for example instead of using `git checkout` on the said commit and listing files in the working directory.

- `repo forall`
  A tool that runs a given command on all Git repositories in the repository and gives some environment variables to the running command such as `REPO_PATH` which gives the path of the current project relative to the root of the repository.

These tools are used for inspecting the repository, except the `repo forall` which is just utility for running commands in all the repositories. In addition, Git provides tools for working with the repositories. However, only inspecting of the repositories is done in this work and therefore those tools, which modify the repositories, are not used.

3.4.1. Web application

The visualization web application in this work is written in CoffeeScript [41] which transpiles into JavaScript. CoffeeScript was chosen as it alleviates some inconvenient features of JavaScript and provides more convenient notation for loops and comprehensions. Overall its syntax is less cluttered and it makes the code more expressive and understandable. The web application uses two popular open source JavaScript libraries jQuery [42] and D3.js [35]. jQuery provides functionality which simplifies developing browser applications. The D3.js is used for rendering the visuals. In addition, it provides bunch of useful utilities such as date ranges and color interpolation. The libraries also support all the modern browsers, thus removing incompatibility problems.

3.4.2. CVE-search

The `cve-search` tool [43] is used to import CVE and CPE information from NVD into a local database as well as keeping the information up to date by updating the database. Amongst other tools it contains a tool for searching the database by CVE, CPE or free text and provides output in multiple formats including JSON. The benefit of having a local database opposed to directly using the NVD is that with a local database we avoid spilling information about the queries, that can be considered to be delicate information, to the third parties.

3.5. Data Collection Framework

The data collection framework was developed to enable easy to setup time series data collection of abstract data. At the simplest the framework allows the users to plug their data point collector scripts to the framework and with minimal configuration start the collection. Framework also allows data to be collected in a batch run for example for initialization purposes or when more than one data points are collected at once. In following sections architecture and data formats are discussed more closely.

3.5.1. Architecture

The design of the frameworks is indented to be simple and modular, so the parts can be easily replaced. In Figure 5 the general architecture of the data collection framework is presented. The controller maintains the scheduling and dispatching of the collectors. The collectors handle fetching and preprocessing data and then hand over the data points to the database. Database naturally holds the collected data and can be queried in order to access the collected data.

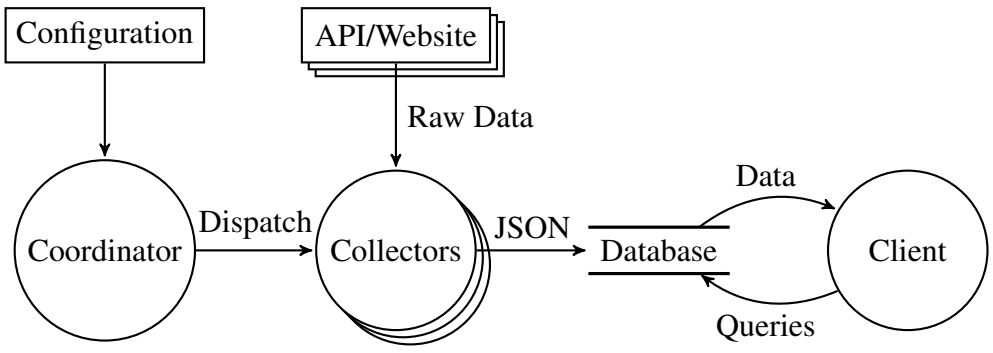

Figure 5. The architecture for the data collection system.

Figure 6 shows a swim lane activity diagram illustrating the control flow between the components that are associated with the scheduled data collection process. At the scheduled time coordinator dispatches a collector wrapper which handles the running of the collector. Once the collector finishes the collector wrapper validates the output of the collector and adds the required metadata. Once the wrapper has prepared the data

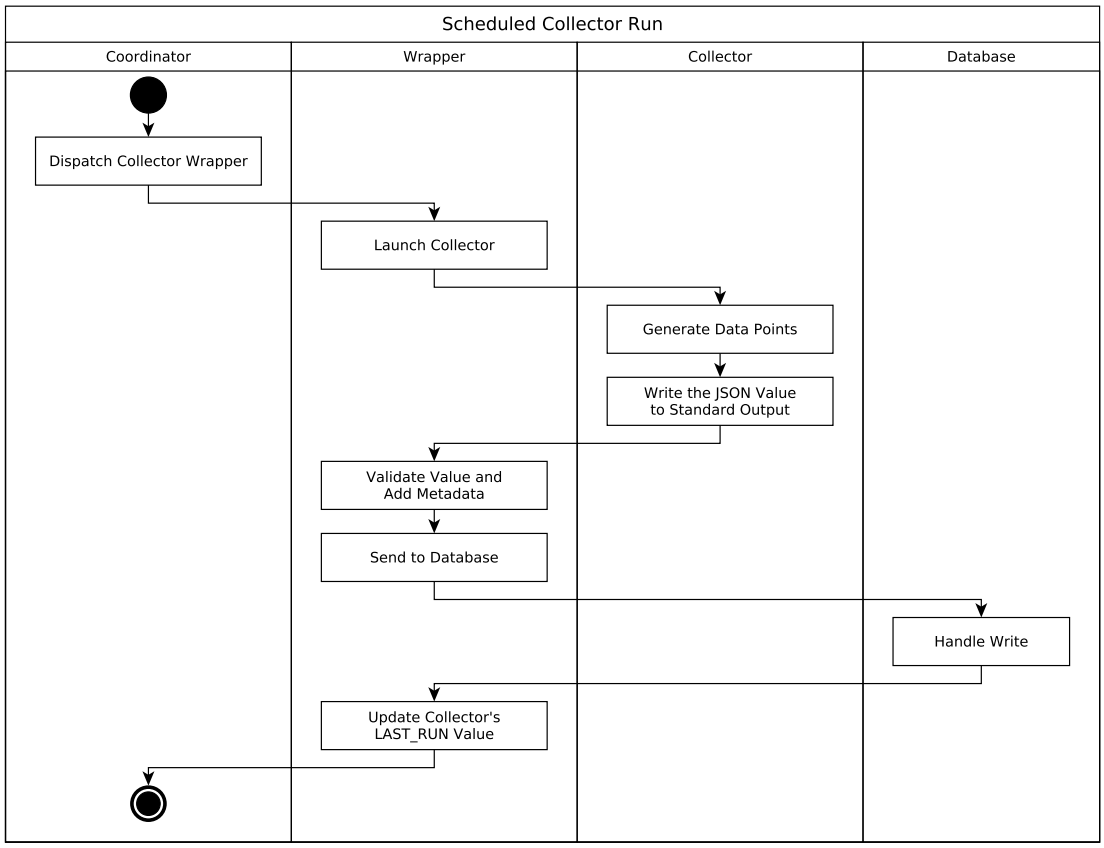

Figure 6. An activity diagram showing the process of scheduled collection event.

points with metadata they are sent to the database which stores the data. After that the value for the `LAST_RUN` attribute of the collector is updated and the collection ends. As can be seen in the figure, the collector is only responsible for providing the data point and everything else is taken care of by the coordinator, wrapper, and database, which are provided by the framework.

3.5.2. Data Formats

The data formats use JSON. The main reasons for choosing JSON as the data format for interfaces were that in addition to the advantages mentioned earlier it is also well known and widely supported. These things together make the JSON a flexible and transparent data format and as such it was chosen to be used in the interfaces.

The format used to store data points is the same as the format of an event in the Cube [20] and is shown in Figure 7. The “time” key stores the timestamp of the data point down to a second precision in the ISO-8601 format. The “type” key is added by the collector wrapper and contains the identifier of the collector which provided the data point. Finally the “data” key contains the JSON value, often an object, provided by the collector script.

To access the collected data the database implementation allows simple queries in the form of a filter which defines the desired time range and the type of the objects. All keys are optional and leaving out a key means that objects will not be filtered by

```
{
    "type": "collector identifier",
    "time": "1970-01-01T00:00:00+0000"
    "data": {
        ...
    }
}
```

Figure 7. An example of the data format. The “data” key contains an abstract JSON object.

```
{
    "query": "query identifier",
    "since": "2015-01-01 00:00:00",
    "until": "2015-01-01 23:59:59",
    "type": "collector identifier"
}
```

Figure 8. The database query filter is a simple JSON-object.

that property. For example not specifying the type returns objects of all types. Figure 8 presents an example of a query object. The query result is a JSON array of objects that match the filter and if no objects are matched an empty array is returned. Optionally the client can specify a query identifier with the key “query” which is echoed in corresponding query response to allow the client to link responses to the sent queries if multiple queries are performed simultaneously.

3.5.3. Configuration

The configuration for the different parts of the framework is done by text files that are given as parameters when the framework is started. The benefits of the text file based configuration is manifold as it makes modifying the configuration fairly easy, simplifies having different configurations, and makes the parsing of the configuration easy to implement.

The configuration of the collector coordinator and the database also basically use the JSON notation as they are JavaScript modules exporting the configuration as an object. The `require` method of the Node.js is used to import the configuration and it accepts either JavaScript module or a JSON file. This makes it possible to determine the configuration in either format. However, using the JavaScript format as presented in following examples has the added benefit of the syntax allowing JavaScript style comments in the configuration file.

Figure 9 shows an example of the systems configuration file. As can be seen the configuration file is a JavaScript module which exports object named “config”. The configuration defines the location of the database and possibly multiple collectors. The collectors are defined by a list of objects each defining the “type” of the collector, command to run the collector with possible arguments and run-interval in Cron-style notation. Also an additional collector configuration allows defining environmental vari-

```
var config = {
    "database": {
        "host": "localhost",
        "port": 7654
    },

    // Configuration comment
    "collectors": [
        {
            "type": "10s-collector",
            "cmd": "collector_script",
            "args": ["arg1", "arg2"],
            "interval": "*/10 * * * * *",
            // lastrun_option = {start,finish,last_item}
            "lastrun_option": "last_item",
            "env": {
                "ENVVAR": "foo"
            }

        },
        {
            "type": "monday-collector",
            "cmd": "python",
            "args": ["../collectors/harvest-monday.py"],
            "interval": "0 0 * * Mon",
            "lastrun_option": "last_item",
        }
    ]
}
module.exports = config;
```

Figure 9. An example of the configuration file of the data collector coordinator.

ables and the method of updating `LAST_RUN` variable for the collector. The update methods "start" and "finish" refer to the respective times of the run of the collector. Moreover, the "last run" method tells the coordinator to use the latest timestamp found in the data points the collector produced on that run. The coordinator uses a JSON file to store the `LAST_RUN` values for the collectors across restarts.

Database configuration is also done by a simple JavaScript module as for the collector coordinator. As can be seen in Figure 10, the configuration defines where the writer and the query sockets of the database are to be bound to and which directory relative to the configure file is used for the database.

3.5.4. Collectors

The collectors are provided by the user and their purpose is to harvest the data points from their data source. Collectors are allowed to collect multiple data points in a single run. This is accomplished by the framework passing the last time the collector was ran as an environmental variable named `LAST_RUN` to give the collector means to determine how many data points it should gather. The collector must print the collected

```
var config = {
   writer: {
     host: "localhost",
     port: 9200
   },
   query: {
     host: "localhost",
     port: 9201
   },
   database_directory: "../database"
};

module.exports = config;
```

Figure 10. An example of the configuration file of the database.

data point as a valid JSON value to the standard output, from which the framework then reads the value and handles the rest of the storing procedure.

Figure 11 shows more detailed diagram of the data flow taking place in collecting a data point. The collectors in Figure 5 are composed of a collector wrapper and a collector script as shown in Figure 11. The collector wrapper is part of the framework and it handles the interaction with the database. The collector wrapper dispatches the collector script which is provided by the user and then waits for the collector script to complete. After the script completes, wrapper checks if the scripts output is a valid JSON object and if the check passes, adds metadata to the data point and hands them over to the database.

3.6. Data Collection

The data collection was done by implementing the collector scripts for the desired data points and then configuring the framework to run the collection using the scripts. In the continuous collection the source code repositories and the vulnerability database are updated daily before collecting the data points. The source code repositories of the projects are used to collect weekly file structure snapshots, daily lines changed on the file level and commits mentioning CVEs or interesting issue identifiers. The weekly snapshots are collected to have an approximation of the structure of the project at each point in time and also the size of each file. In addition, the data of the activity of the project is collected by composing daily added and deleted lines according to Git on a file level.

To capture the structure of the project both snapshots and changes are saved as a hierarchical structure in JSON format. Figure 12 illustrates the hierarchical structure of the daily changes objects. The directory and file names are stored in the “name” field. The directories have a field named “children” which contains a list of objects denoting the contents of the directory. The files on the other hand have fields for the number of added and deleted lines for that day or a dash denoting a change in a binary file. In the daily changes only the files which have been changed are reported which greatly limits the size of the objects compared to snapshot objects as most of the code

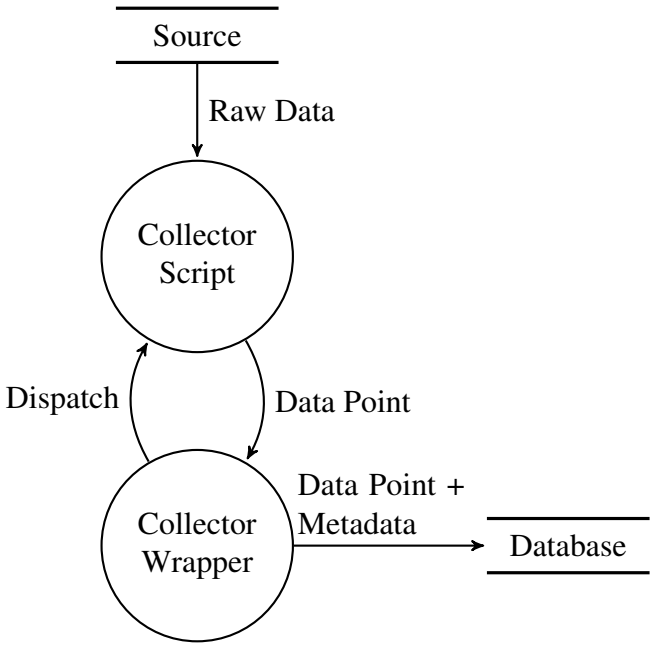

Figure 11. Architecture for collecting a single data point.

base is left untouched by the changes done in any single day. A snapshot contains the full file structure of the project for the time of taking the snapshot and is very similar to the structure of a daily changes object. The only difference is that files have a field for the size in bytes instead of the changes.

As Android is a massive project the weekly snapshots in the format above are over twenty megabytes in size. This can be argued to break the desired human readability of the data format because files of this size become very cumbersome to inspect by hand and the complex hierarchy is hard to grasp. Initially the snapshot objects were stored in the database as such. However, the large objects resulted in long response times for the queries due to the experimental setup. The lack of indexing in the database implementation results in reading and parsing all objects in a time frame in order to perform the queries. This makes the processing of queries slower the larger the entries in the database are in size, which in the case of the snapshots was noticeably slow. As a workaround the objects were moved to their own files and the file locations were stored in the database instead of the raw objects. This reduced the size of the database files down to a couple kilobytes per day which greatly improved the query times. The daily changes are much more maintainable topping one megabyte in size however they too were moved to separate files. The future work includes properly resolving the problems caused by the large objects by replacing the experimental database with a modern database backend.

The harvesting of weekly snapshots and the daily changes were implemented as a Bash scripts because most of the heavy lifting is done by the Repo and Git tools. A custom Python tool was implemented for parsing the file listings into the hierarchical JSON format shown in Figure 12. In short Repo and Git were used to harvest the data from the source code repositories and the data was parsed into JSON object by

```
{
    "name": "root",
    "children": [
        {
            "name":"fileA",
            "added": 100,
            "deleted": 80
        },
        {
            "name":"directory",
            "children": [
                {
                    "name":"fileB",
                    "added": 4,
                    "deleted": 3
                }
            ]
        }
    ]
}
```

Figure 12. Daily changes are stored in a hierarchical JSON structure.

the Python tool. Finally a timestamp was added to the generated JSON object as the harvesting was done retrospectively.

The weekly snapshots were created using the `repo forall` to get the file listing for each Git repository in the Android project from a specified time. As the Git repositories maintain the complete history of the project getting snapshots is possible retrospectively. The snapshots were constructed for the state of the file structure as at the start of each Monday. This was done by using `git log` to identify the last commit before each Monday and then pointing `git ls-tree` to that commit to receive the file hierarchy and sizes as was the state of the project after that commit. As `git ls-tree` shows only the filenames related to the root of that Git repository, each path were then prefixed with the `$REPO_PATH` environmental variable provided by the Repo tool to acquire the listing related to the root of the Android project.

The daily changes were also done retrospectively but for each day in the selected history range. Similarly to the file hierarchy snapshots `repo forall` was used to run the command on all Git repositories and the file names were prefixed to get the paths relative to Android root. The changes for each day were constructed by getting the last commit for the day and the last commit for the preceding day and using `git diff --numstat` to the commits to get a machine readable listing of the number of added and deleted lines for each file changed during that day. If there have not been any commits during the day the last commit for the day and the preceding day is the same thus resulting to zero differences.

Collecting the commits mentioning CVE-ID in the case of Android is done daily after updating the source code repository. The commits after the last run are filtered for the CVE-ID and matching commits are parsed to a JSON object and stored into the database. In addition, the merge commits are inspected for the development history

that they introduced to the main branch as explained in the Section 3.4. The mere filtering of the commit history by timestamp would ignore the addition of those commits.

As mentioned in Section 2.4.2, the vulnerability information for Chromium is collected from the issue tracker of the project. The issue tracker can be viewed via a web interface, but it also provides its contents for download in a comma-separated values (CSV) format. The CSV format is a simple text based structured format where entries are listed line by line and the values of the entries are separated by commas as the name suggests. The tracker entries contain information about issues' categorization, priority, modification time, owner, and summary, as well as an unique identifier for each issue. The category tags are used to identify security related issues. After a manual inspection of security related issues a set of tags which were common for the interesting issues were selected. Following is a list of the selected tags with short explanations for why they were selected.

- **Status: Fixed**
  Only consider fixed issues as these are most likely the valid issues, which have resulted in some action.
- **Type: Bug**
  Limit selected issues to actual bugs. Other issue types such as features are not interesting in the context of security bugs.
- **Category: Security**
  Assuming security related issues are related to vulnerabilities.
- **not Category: Security-UX**
  UX stands for user experience and the issues in this category are security issues mostly related to the shortcomings of the user interface. The decision to leave these out is based on their tendency to be related to wording problems or lack of information provided to the user, hence probably not caused by actual programming flaws.

The issue tracker gives an option to filter which entries are downloaded therefore only the issues with the selected tags were downloaded. The filtering tags can be appended as a query string into the download address of the issue tracker CSV, which enables fetching and updating the listing easily with a script.

After the filtering the resulting CSV file contains only the security related issues. Issue identifiers are parsed from the file and then used to identify the related code changes. Other pieces of information from the CSV file were not used in the implementation as the categorization of the issues was the most interesting attribute.

The data in the issue tracker of the Android project was found to be of a poor quality. Therefore, the vulnerability information used for Android consist of the CVE-IDs mentioned in the commits. In addition, the commits are combined with the relevant data from the NVD using the `cve-search` tool.

3.7. Data Visualization

Data visualization is implemented as a browser application. This queries the websocket interface of the database to access the collected data. The visualization is written in CoffeeScript and transpiled into JavaScript which is executed by the browser.

The graphical user interface of the visualization application consists of the treemap view, date selection slider and a scope selection button set. The scope selection determines the time frame around the selected date to be displayed. The date selection slider is used to select the desired time to be inspected. After a date is selected, the application queries the database for the required data and prepares the visualization. Visualization is displayed on a treemap layout. The treemap layout divides the given area between the selected elements and can be used to illustrate multiple properties by modifying the size and the color of the areas. The view can also be used to traverse the file hierarchy by clicking the elements, much like one would do in a file browser, to expose the hierarchical structure of the project and to allow more precise inspection.

The implemented data visualization combines the structure of the project and activity and vulnerability information to a heat map presentation. As a base the structure of the project is visualized using treemap layout which is rendered using the D3.js which calculates the layout in a data driven fashion. The layout displays each element as a rectangle with a size related to the logarithm of the size of the element on the disk. The treemap layout is capable of displaying multiple levels of hierarchy simultaneously. However, showing two levels of hierarchy at once was found to be too cluttered and therefore single level of the hierarchy was chosen. In addition, one level of hierarchy is more intuitive to understand as it shares similarities with the graphical file browsers in modern operating systems.

To give an overview of the data, colors are used to indicate the development activity and the vulnerability data relating to source code locations. File locations with no activity and no vulnerabilities are displayed as gray. If there has been activity in the code locations the corresponding rectangles are made yellower the more changes they have been subject to. The last level of color coding is adding red to the color if there has been security issue fixes in the locations.

The visualization application does all the necessary combining of the different data objects in order to construct the visualization. However, displaying a file structure over a period of time is not an accurate presentation as the file structures tend to change over time. If the selected time frame happens to include multiple file structure snapshots the visualization compromises and constructs a union which includes all the files in all the snapshots. In the cases where two snapshots have a file and a directory with clashing names the file is discarded in favor of including the directory. As the file structure snapshots also include the size of the files on the disk when combining snapshots with same files the latest size of the file is retained. The daily changes are combined in similar way to snapshots but added and deleted lines for files are combined by summing instead.

Figure 13 illustrates the steps that are required to generate the visualization after the date range to inspect has been selected. Firstly the date range is calculated and the corresponding queries are sent to the database concurrently. As the responses arrive from the database the processing continues in the case of snapshot and change objects by fetching the file containing the actual object as the database holds only the file lo-

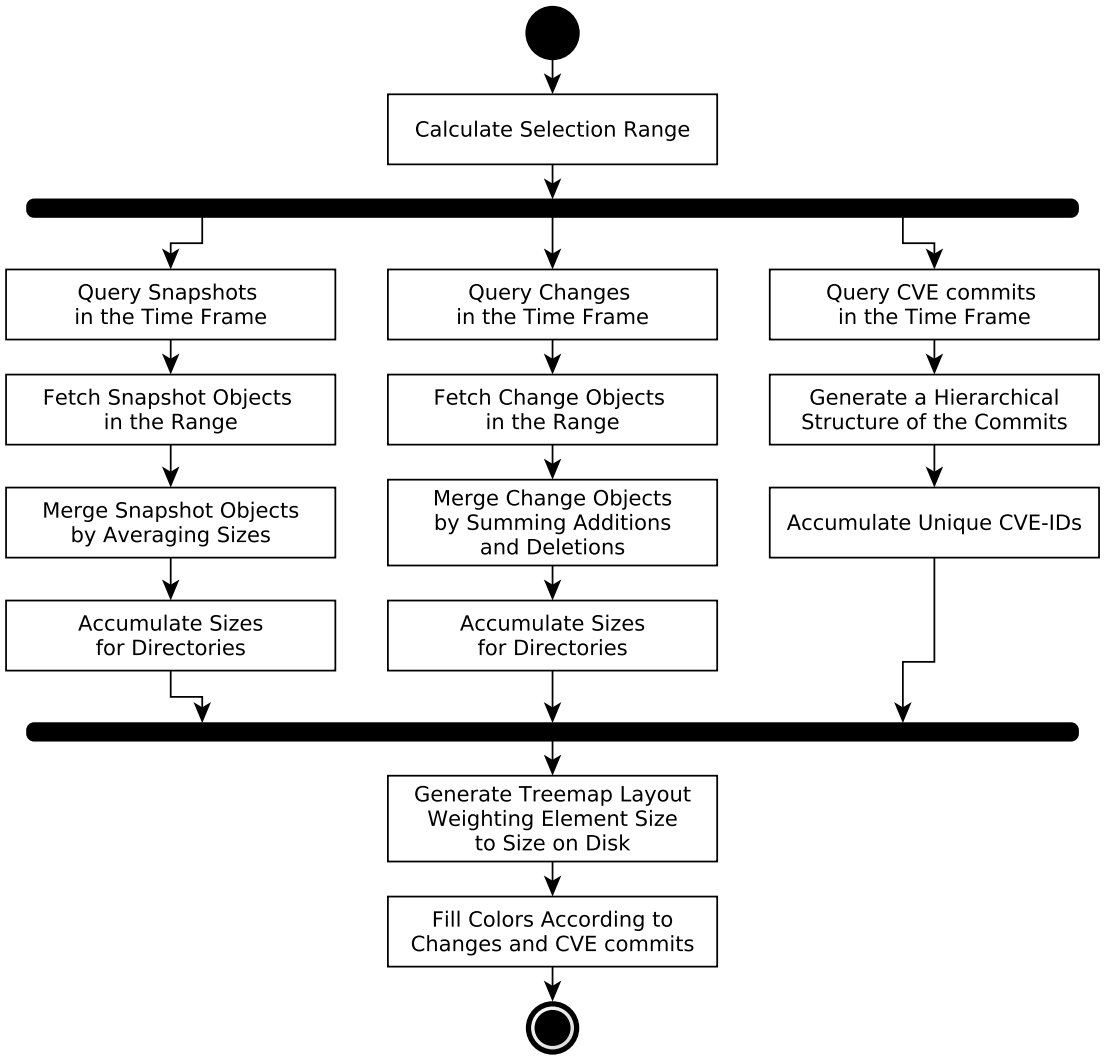

Figure 13. The activity diagram demonstrating the generation of the visualization.

cations of the objects. The CVE commit objects are not stored in hierarchical structure so a tree is constructed using the file paths the commits touch to fit the data with the snapshots. The time range includes multiple objects of each type so the next step is to combine the objects intelligently in order to display them in a single visualization. At this point all the attributes are stored only for the leaf nodes of the trees, which represent files. The values must be accumulated to the intermediate nodes representing the directories for rendering the visualization at all levels. Once all this preparation is done for the received data the layout is calculated and the colors are filled in.

3.8. Mapping Vulnerabilities to Code Changes

Mapping vulnerabilities to code changes is a process where the code changes responsible for the vulnerabilities are determined after the vulnerability has been found. Narrowing down a precise cause for a vulnerability is challenging and would require expert analysis of every vulnerability. For this reason a looser automatic method is used in the

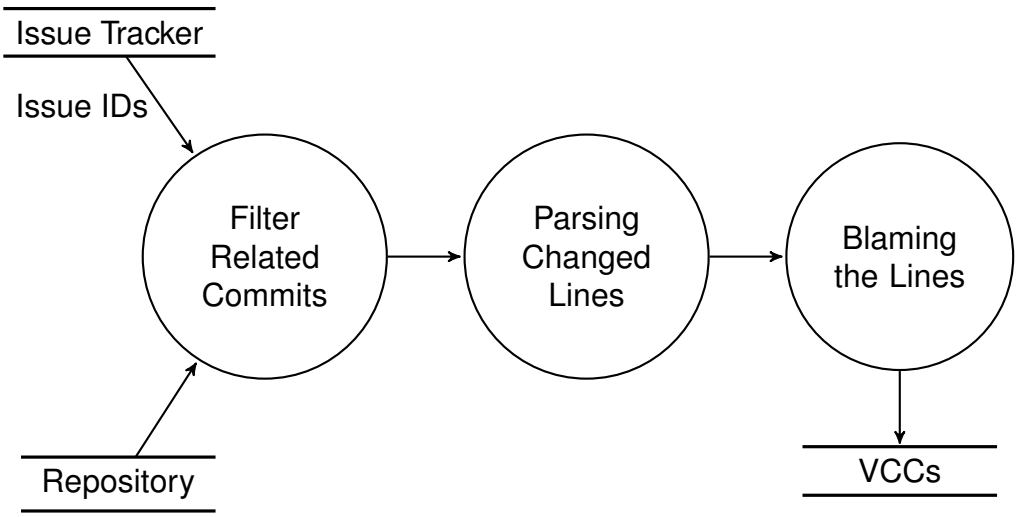

Figure 14. The work flow of mapping vulnerability issue to VCCs.

case study. The changes in commits related to issues are traced back to the commits that last modified concerned parts of the source code. In this context a vulnerability contributing commit (VCC) is any commit whose non-comment code changes are modified by the commit related to an issue.

The architecture used for mapping vulnerabilities to VCCs is illustrated in Figure 14. The issue identifier from the issue tracker is used to identify the commits assumed to introduce fixes to the related issue. The Algorithm 1 presents the procedure used to identify the VCCs for a given issue. The input is an issue identifier received from the issue tracker and the output is the commits blamed for introducing the issue. Following is a more precise description of how the major steps were implemented.

input : An issue identifier id
output: A list of blamed commits C

1 $F \leftarrow$ search commits mentioning id
2 **foreach** *fix commit* f *in list of commits* F **do**
3 $\quad p \leftarrow$ get f's patch
4 $\quad d \leftarrow$ parse deleted code lines from the p ignoring comments and blank lines
5 $\quad b \leftarrow$ blame each line in d to get a list of commits
6 $\quad C \leftarrow C + b$
7 **end**

Algorithm 1: The procedure for mapping an issue to the assumed causing commits

First step in Algorithm 1 is to identify the commits related to the issue which is done by using `git log` using its command line parameter "`--grep`" to filter commits whose message contains the identification number of the issue. There is many to many relation between issues and commits as a single commit can touch on multiple issues and an issue may need multiple commits before being resolved.

Third step concerns getting the patch of the given commit which can be acquired using `git show`. The patch shows line by line the differences in the file before and after the commit. Even though the added and removed lines are shown in the patch,

the fourth step is to use only the removed lines for the tracing the related commits with following justifications. Patches used are line-based so even a change of one character results in a patch which removes the old line and adds the new line with the change. This means that the added and removed lines would trace back to the same origin and thus inspecting only the removed line is justified. Another case is when the patch only adds lines to a specified part, this can mean that something was missing from the original commit in this context, but the scope to "blame" is ambiguous. For example if a commit adds a test for a variable to a file with a long history, there is no clear way to determine which commit should have included in the test in the first place. In attempt to increase the signal to noise ratio blank and comment lines were ignored. The actual parsing of the deleted lines and filtering commits and blanks from the commit patches generated by Git were implemented as a Python script.

Fifth step is actually to trace back the selected lines to their introduction to the code base which is done by using the `git blame` tool which can be given a set of source code lines as an input and it traces the commit history of the repository back and finds commits which introduced each line to the repository. By using `git blame` on lines parsed in previous step a commit to many commits relation is acquired. Commits which only add lines to the repository result in zero related commits as only the removed lines are traced back.

By using this method we can automatically find the commits assumed to cause an issue. The results and discussion of this method are presented in the following chapters.

4. RESULTS

In this chapter the results of the data collection, visualisation and tracing back the fixes to commits are discussed. First an overview of the collected data is given followed by the visualisation done. Finally the results of the tracing fix commits to VCCs are presented.

4.1. Collected Data

The data for Android and Chromium was collected about a general project status from source code repository and vulnerability data from other sources combined with linked source code changes.

The Android Open Source Project was first published in 2008 but the project makes use of multiple long lived projects which vastly predate the Git. In the repository multiple projects had their first commit dated to year 1970 which has probably been some kind of default when projects were first ported to a VCS. However, the earliest changes to the repositories are dated to year 1995 so there is also a lot of history from the time before the Android.

The following example data objects in Figures 15, 18, and 19 have been formated by adding line feeds and indentation to make them easier to read. The original data objects are on a single line with no extraneous spaces to save space. If one wants to inspect the file there are tools for automatically intending the object to aid with readability. As can be seen in the Figures 18 and 19 the objects are deeply nested in order to present the hierarchical structure of the project.

Figure 15 is an example of the data collected by the collector which collects commits that mention CVEs. The metadata contains the type identifier of the collector, “android-cve”, and the timestamp that was parsed from the commit. The “data” holds the information parsed from the commit including the identifier, author, and description message of the commit. In addition, the “changes” holds a list of summaries of the changed files including the relative path to the Android projects root and number of added and deleted lines. In this case the commit changed only one file thus the array in “changes” contains only one element. Finally the “cves” contains an array of the mentioned CVEs.

We can add to the value of the report in Figure 15 by searching for the CVE-ID “CVE-2015-1538” using the `cve-search` tool. The JSON formated result for the query is shown in the Figure 16. From the CVE entry we can see that the CVE has a CVSS of “10.0” which is the highest possible value indicating that the vulnerability in question is very severe. This seems strange when compared to the commit message found in Figure 15 which describes the commit fixing “benign overflows”. However, the CVE entry mentions the same file “SampleTable.cpp”, which is the same file the automatically found commit modified. The CVE entry also seems to have interesting data in the key “references” pointing to project specific websites related to the CVE. This data could also be incorporated to the vulnerability mapping procedure.

To get more information about the apparent conflict between the commit message in Figure 15 and the CVSS found in corresponding CVE entry in Figure 16 a manual inspection of the development history of the mentioned file “SampleTable.cpp”

```
{
  "type":"android-cve",
  "time":"2015-10-16T19:22:09.000Z",
  "data":
  {
    "author":"Dan Austin <danielaustin@google.com>",
    "commit":"ffd7950633edeb8e990770e4c8abb81bdcaa6f32",
    "message":"Fixed benign overflows triggered by tests CVE
       -2015-1538-1 andCVE-2015-1538-2 in CTS.\n\nBug: 25016754\
       nChange-Id: I0ceb2c799899015be6b37d5e94fe306d0037a8d2\n",
    "changes": [
      {
        "deleted":4, "added":9,
        "file":"frameworks/av/media/libstagefright/SampleTable.cpp"
      }
    ],
    "cves":["CVE-2015-1538"]
  }
}
```

Figure 15. An example of a collected Android repository commit mentioning a CVE-ID.

was done. From the development history it can be seen that the file had not been changed during the months before the commits displayed in Figure 17. The commits fix multiple problems in the code and are made by the same author who discovered the vulnerability, and thus can fairly certainly be assumed to be the real fix for the vulnerability. Moreover, when comparing the timestamps of the commits and the publishing date of the CVE in Figure 16, it can be seen that the commits precede the publishing by couple of weeks. However, it can also be seen that none of these commits mention the CVE-ID assigned for the vulnerability thus breaking the assumption used for filtering the interesting commits in the first place.

Figure 18 and Figure 19 show the examples of the general structure and the information stored in the weekly snapshots and the daily changes collected. As can be seen the file structure hierarchy has many levels and the objects are large. For this reason the objects have been truncated to show only a part of the lowest level of the hierarchy. However, the general structure is illustrated and the rest of the objects are structured in a similar way.

4.2. Mapping Vulnerabilities to Source Code Locations

Figure 21 shows the view of the implemented visualization which allows inspecting the project activity and vulnerability locations in different periods during the history of the project. There is a slider in the bottom of the view for selecting the time period and a selector for the time frame to be viewed with three options day, week and month. In the top of the view on a blue background the relative path to the root of the project is shown indicating the location of the currently shown directory. It also acts as a link for traversing back to the parent directory of the current directory. As only one level of

```
{
    "Modified": "2015-10-01T12:19:27.707-04:00",
    "Published": "2015-09-30T20:59:06.687-04:00",
    "access": {
        "authentication": "NONE",
        "complexity": "LOW",
        "vector": "NETWORK"
    },
    "cvss": 10.0,
    "cvss-time": "2015-10-01T09:54:50.467-04:00",
    "cwe": "CWE-189",
    "id": "CVE-2015-1538",
    "impact": {
        "availability": "COMPLETE",
        "confidentiality": "COMPLETE",
        "integrity": "COMPLETE"
    },
    "references": [
        "https://groups.google.com/forum/message/raw?msg=android-
            security-updates/Ugvu3fi6RQM/yzJvoTVrIQAJ",
        "https://android.googlesource.com/platform/frameworks/av
            /+/2434839bbd168469f80dd9a22f1328bc81046398"
    ],
    "summary": "Integer overflow in the SampleTable::
        setSampleToChunkParams function in SampleTable.cpp in
        libstagefright in Android before 5.1.1 LMY48I allows remote
        attackers to execute arbitrary code via crafted atoms in MP4
        data that trigger an unchecked multiplication, aka internal
        bug 20139950, a related issue to CVE-2015-4496.",
    "vulnerable_configuration": [
        "cpe:2.3:o:google:android:5.1"
    ],
    "vulnerable_configuration_cpe_2_2": [
        "cpe:/o:google:android:5.1"
    ]
}
```


Figure 16. An example of a JSON CVE entry produced by the `cve-search` tool.

```
commit c24607c29c96f939aed9e33bfa702b1dd79da4b7
Author: Joshua J. Drake <android-open-source@qoop.org>
Date:   Wed Apr 8 23:44:57 2015 -0500

    Fix integer overflow during MP4 atom processing

    A few sample table related FourCC values are handled by the
    setSampleToChunkParams function. An integer overflow exists within this
    function. Validate that mNumSampleToChunkOffets will not cause an integer
    overflow.

    Bug: 20139950
    Change-Id: I1972cc185fce5e058afa143ad5eabcc269ad324d

 media/libstagefright/SampleTable.cpp | 3 +++
 1 file changed, 3 insertions(+)

commit ad435371a4b95e16ceb49ab28efc04da8b3680e1
Author: Joshua J. Drake <android-open-source@qoop.org>
Date:   Wed Apr 8 23:31:25 2015 -0500

    Detect allocation failures and bail gracefully

    During the processing of several sample table related MP4 atoms, allocation
    sizes could be large enough cause a std::bad_alloc exception to be raised. This
    typically causes a crash (denial of service condition). Use std::nothrow to
    catch allocation failures and return gracefully.

    Bug: 20139950
    Change-Id: I03d3f01b24e5fe3fa38985914bcfa694ea3dc09e

 media/libstagefright/SampleTable.cpp | 21 ++++++++++++++++++-----
 1 file changed, 16 insertions(+), 5 deletions(-)

commit e2e812e58e8d2716b00d7d82db99b08d3afb4b32
Author: Joshua J. Drake <android-open-source@qoop.org>
Date:   Wed Apr 8 23:23:55 2015 -0500

    Fix several ineffective integer overflow checks

    Commit edd4a76 (which addressed bugs 15328708, 15342615, 15342751) added
    several integer overflow checks. Unfortunately, those checks fail to take into
    account integer promotion rules and are thus themselves subject to an integer
    overflow. Cast the sizeof() operator to a uint64_t to force promotion while
    multiplying.

    Bug: 20139950
    Change-Id: Ieb29a170edb805c722fc5658935f2390003e5260

 media/libstagefright/SampleTable.cpp | 6 +++---
 1 file changed, 3 insertions(+), 3 deletions(-)
```

Figure 17. The Git log output showing the fix commits for CVE-2015-1538.

```
{
  "time": "2015-12-07T00:00:00.000Z",
  "type": "android-snapshot",
  "data": {
    "name": "root",
    "children": [
      {
        "name": "development",
        "children": [
          {
            "name": "testrunner",
            "children": [
              { "name": "android_manifest.py", "size": 3747 },
              { "name": "am_instrument_parser.py", "size": 5683 },
              { "name": "adb_interface.py", "size": 19525 },
              { "name": "test_defs.xml", "size": 18865 },
              { "name": "errors.py", "size": 1340 },
              { "name": "test_defs.xsd", "size": 5721 },
              {
                "name": "coverage",
                "children": [
                  { "name": "coverage_targets.py", "size": 3890 },
                  { "name": "coverage.py", "size": 12887 },
                  { "name": "coverage_target.py", "size": 1221 },
                  { "name": "__init__.py", "size": 62 }
                ]
              },
              .
              .
              .
```

Figure 18. A truncated example of the project snapshot object showing hierarchy of a directory.

```
{
  "time": "2015-10-11T00:00:00.000Z",
  "type": "android-changes",
  "data": {
    "name": "root",
    "children": [
      {
        "name": "external",
        "children": [
          {
            "name": "libweave",
            "children": [
              {
                "name": "libweave",
                "children": [
                  {
                    "name": "include",
                    "children": [
                      {
                        "name": "weave",
                        "children": [
                          {
                            "name": "enum_to_string.h",
                            "added": 1,
                            "deleted": 1
                          },
                          {
                            "name": "provider",
                            "children": [
                              {
                                "name": "http_client.h",
                                "added": 8,
                                "deleted": 1
                              },
                              {
                                "name": "test",
                                "children": [
                                  {
                                    "name": "mock_http_client.h",
                                    "added": 7,
                                    "deleted": 13
                                  }
                                ]
                              }
                            ]
                          }
                        ]
                      }
                    ]
                  },
                  .
                  .
                  .
```


Figure 19. A truncated example of the daily changes object showing added and deleted lines for files for that day.

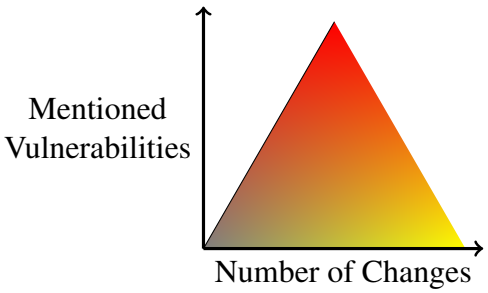

Figure 20. The visualization node coloring legend.

the file hierarchy is shown in the visualization, the user can traverse the hierarchy by clicking on the elements. Further information about each element on the screen can be viewed by hovering over them with the cursor.

The visualization shown in Figure 21 is the root directory of the Android source code repository. The view is a summary of changes and commits relating to vulnerabilities for a week from the beginning of August 2015. Figure 20 illustrates the change of the color of the node in relation to changes in the code base and number of unique vulnerabilities mentioned in the changes. The more changes a directory has had in it during the selected time period the yellower the corresponding rectangle is and the number of unique CVEs that have been mentioned in the commits applying changes to directories add red to the color of the rectangle. The size of the rectangle is logarithmically related to the approximation of the size of the directory on the disk. As can be seen in Figure 21 most of the top level directories have been worked on during the selected week but only one directory has changes mentioning a CVE-ID. More information of the directories can be obtained by hovering over the rectangle which displays a tool tip. The tool tip in Figure 21 shows the name of the directory, approximated size on the disk, the number of added and deleted lines and the CVE-IDs mentioned in the commits. For closer inspection the user can click the rectangle to traverse the hierarchy and display the contents of the selected directory.

Figure 22 shows the view after the “external”-directory was clicked in the Figure 21. The new view discloses the more precise location of the mentions of the CVE-IDs as well as the other contents of the selected directory in the same way as in the previous figure. There are also a lot of subdirectories which have had no activity during the selected time frame in Figure 22. To find the file affected by the changes mentioning CVE-ID the user could keep following the hierarchy by clicking the reddish directories until the directory where the affected files are located is reached.

4.3. Mapping Fixes to Vulnerability Contributing Commits

The automatic blaming were experimented with the source code repository of Chromium. The total of 283 commits referencing issues from issue tracker were traced back to commits introducing the removed lines in the fix using the procedure presented in Section 3.8.

Figure 23 shows the distribution of the number of commits blamed by each fix. As can be seen generally only few commits introduced the lines removed by any single

Figure 21. The visualization showing the root of the project.

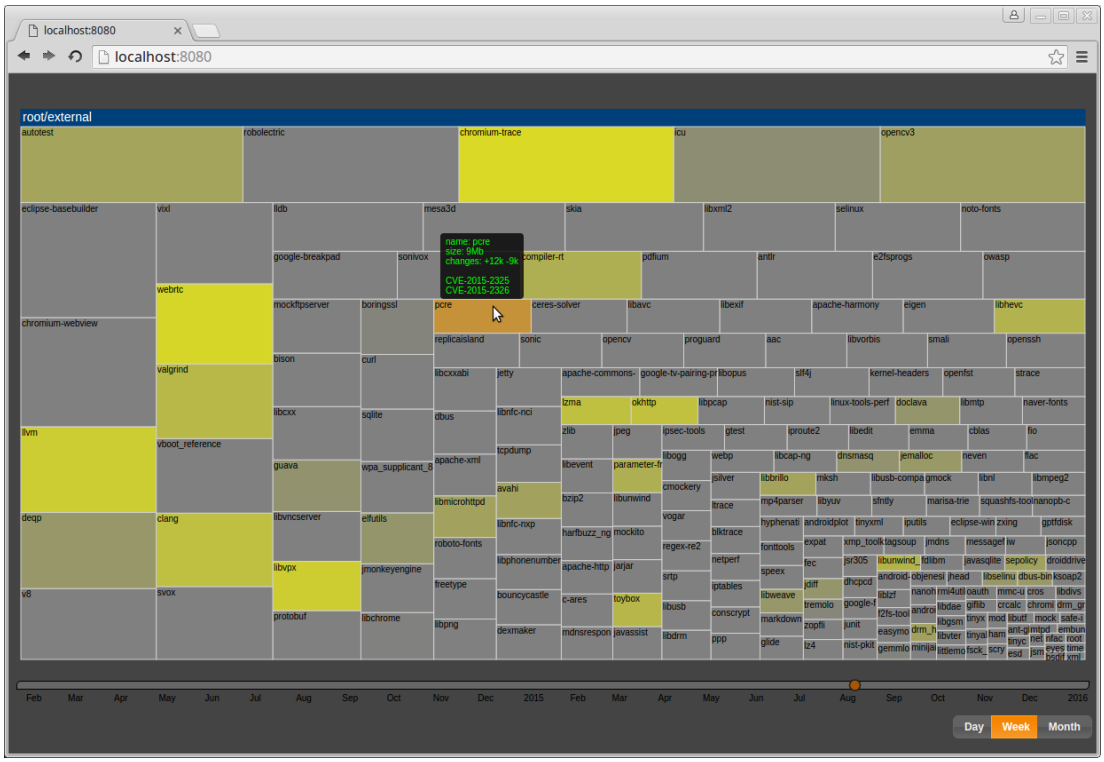

Figure 22. Traversing the file structure using the visualization.

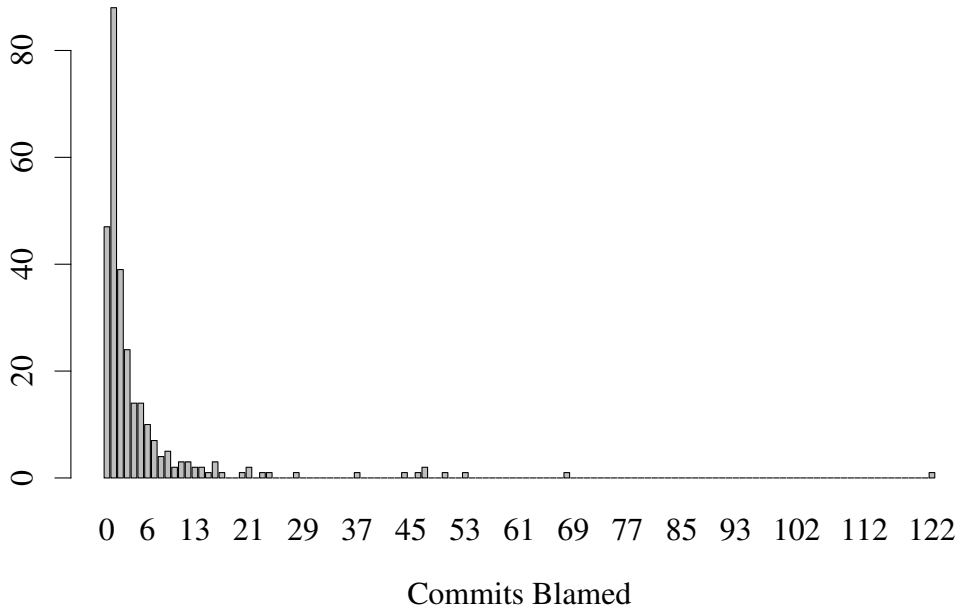

Figure 23. A bar plot showing the distribution in number of unique commits blamed for each fix.

fix commit. A relative high number of fix commits, 47, resulted in zero blamed commits which means the patch did not remove any lines, indicating that the problem was related to something, for example validation, missing. The mode of the dataset can be seen to be one, with frequency of 88, which is the most favorable result if the objective were to predict which commits are likely to introduce security issues as the effect of other commits could be ignored in a simple model. However, 52 % of the blames result in two or more commits. In addition, there are outliers which result in dozens of unique commits. This hints that issues are often the result of the combined effect of multiple commits.

5. DISCUSSION

In comparison to the two time series data collection frameworks introduced in the Section 2.3, Cube and Seriesly, the implemented framework provides features that make starting the data collection easier for the user. In addition to storing the data, the framework takes care of the scheduling of the collection and the communication with the database. A user only needs to implement the collector and configure the desired collection interval. The database implemented is a hierarchical directory structure with individual text files for each day. Because there is no indexing or other performance increasing features especially large objects slow down the operations. Because the data format used for collected data is the same as in the Cube, it could easily be used to replace the database back end. Doing so would provide the improved performance of the MongoDB and the more advanced queries and analysis. However, the database implementation presented in this work was chosen to enable simpler and more transparent access to the data.

The following arguments are made to assess the validity of the used vulnerability data. In the case of Chromium's issue tracker the issue categorization is validated by the developers of the project. Using only the issues that have already been resolved gives a set of issues, which have led to some action. As such it is reasonable to assume that the issue was real and the assigned categorization is valid. However, the security categorization is used broadly and the classification practices might vary so the categorization information is not guaranteed to be indicative of an actual vulnerability. In the case of Android only security issues which have been assigned an actual CVE, which is a strong indicator of an actual vulnerability, are used as data points. The validity of the vulnerability to source code mappings rely solely on the commits mentioning the specified identifiers. This means that mapping a vulnerability to a source code location fails if a commit fixing an issue does not mention the identifier as demonstrated in the Section 4.1. However, if a commit mentions an vulnerability identifier the link between the commit and the vulnerability is probable. In the vulnerability blaming procedure commit mentioning an issue identifier was assumed to be a fix for the commit, but there are also other reasons for developer to mention the identifier for example when implementing test or mitigation for the issue.

As the assumption that developers mention the related issues in the fix commits was demonstrated to be false, a more robust way of mapping vulnerabilities to fixes is required. As all the other parts of work presented in this thesis are based on the mapping of vulnerabilities to fixes developing a more sophisticated methods for it would be the most important work in the future. The pointers for seeking a better performing mapping procedure would be to correlate the dates in CVE entries and the changes in the code base before the publishing of the CVE. This would probably be efficient especially when vulnerabilities are discovered in components which are not under active development. Also in some cases the CVE entry contains the information of the location of the vulnerability which greatly helps locating the fix changes. Also the reference data in the CVE entries in NVD database could prove to be valuable information when available. An another way to possibly improve on the accuracy of finding fix commits would be to use natural language processing to finding indications of underlying vulnerability fix from the commit messages.

As the vulnerability reports are often subject to a disclosure timeline which is in Google's case 90 days or until a widely available fix is deployed with some exceptions [44], the issue tracker has incomplete visibility of the issues for the public which limits the uses of issue tracker as a vulnerability source. It was noticed during the work that there seems to be bots for automatically removing viewing restrictions on issues which have passed their deadline in case the developer did not manually lift the restriction once fix was deployed. However, the stability of the issue tracker as a data source can be questioned as the issue sets retrieved with the same parameters had some issues which were available when previously fetched but when fetched again on a later date disappeared from the received set and were not available for viewing in the web interface of the tracker. This could be remedied and more closely studied by keeping the local database of the publicly viewable issues of the issue tracker and frequently updating it and keeping track of the visibility of the issues.

As the Git stores the full development history of the project the collected data might be subject to noise from at least following sources. The reverting commits is a common procedure in the work flow of a project and it is not uncommon to see the same commit introduced and reverted multiple times. This might cause for example the daily changes numbers to be inflated and if the commit which gets reverted mentions a identifier which was looked for the same commit might get added to the mapping multiple times. An another source of noise is rebasing, which can be used to combine part of the development history into a single commit. This causes noise by possibly bunching up an security issue with non security related commits and thus covering the actual location of the vulnerability.

However, if there was a reliable automatic method for tracing back the causes for vulnerability after the disclosure of the vulnerability that data could be used to generate a dataset of commits that introduce vulnerabilities and this would open a way for many interesting analysis. Of course these analysis can be done today by manually tracing back the commits but an automatic method would allow much larger scale. The commits could be used to train a model for predicting the likelihood of a given commit or code change to introduce a vulnerability. After the model for VCC is created in addition to scanning new commits to assess their risk the whole history could be scanned similarly to possibly discover previously undiscovered vulnerabilities. Also characteristics not directly related to code could be analysed in wide scale. For example how does the time of the day and the week day affect the probability of faulty code or recognizing developers who are often involved in the introduction of vulnerabilities.

The visualization implemented provides a quick overview of the activity of the project and highlights the possible vulnerabilities that have been fixed at given time frames. This could be useful information for a vendor maintaining their own version of the Android operation system by prioritizing the integration of the vulnerability fixes. Also when composing a visualization over a longer time period the visualization might help to notice patterns like if certain components are more prone to vulnerabilities than others and as such helps to direct the testing effort. However, the visualization is limited by the missing vulnerability to location mappings and thus does not necessarily provide a truthful image of the state of the project.

Generally following the changes in a software project is interesting for the tester especially in a mature project whose vulnerability discovery has reached saturation. In this situation changes indicate a chance of software regressions. The recently modi-

fied locations could be combined with information about which tests have previously touched these parts of the code base. This could then be used to guide automated semi-targeted fuzz testing after changes happen in the code base.

As presented in Section 4.3 a large part of issues were traced back to single commit, so one way to use this result would be to use only the cases where there is one to one mapping between fixes and issues and use that as a dataset to train a model for determining the likelihood for a commit to introduce a vulnerability. However, in presented results there were only 88 fixes which blame single commit which is too small a sample set for constructing any relevant model. Another way to improve upon the procedure of casting blame would be to incorporate static code analysis tools to determine which lines are likely to contribute to a bug. On the other hand it could also be used to ignore lines that are safe in order to limit the number of lines blamed and thus reducing the number of unique commits the patch get traced back to. The used procedure ignores only comments and empty lines in the patch as they are not contributing to the functionality of the program with high certainty.

The future work most importantly includes identifying and incorporating additional open data sources. In addition, it contains improving methods for automatically mapping vulnerabilities to code changes assumed to fix them using proposed approaches. The visualization application can be optimized to better handle the data. Also implementing colorblind support to the visualization would make it more accessible. This could be done for example by displaying the number of mentioned issues or by providing additional color schemes. The future work concerning the data collection framework includes replacing the database backend with a more performant solution. Moreover, the framework should be utilized in different applications to evaluate the generality of the implementation.

6. CONCLUSION

The purpose of this study was to experiment with combining more than one data source for software security related analysis. Two popular and security critical open source projects, Chromium and Android, were selected for the analysis. Data related to the development of the projects was collected from the source code repositories. The data was then combined with the vulnerability related data from the additional data sources selected for the projects. Also a simple method for mapping the source code locations affected by the vulnerabilities was presented and the results were visualized. In addition, an automatic method for using a fix to determine the code changes that introduced the issue was experimented with.

Firstly interesting data sources for the projects were identified. For Chromium the issue tracker of the project was used as the source for vulnerability information. The issue categorizations found in the tracker were then used to identify the security fixes in the code base. The identified fixes were then used to experiment with the automatic procedure of finding the causes for the vulnerabilities. By using this method it was found that most of the issue fixes were traced to two or less commits, but some of the fixes resulted in dozens of commits being blamed.

In the Android case study NVD was used as the vulnerability data source and linking to fixes was done by searching for the CVE identifiers in the source code repository. The aim was to find and visualize where vulnerabilities had been found at different times and also enrich the data by using information extracted from the NVD.

In both case studies we assumed that the developers would mark up the identifiers for the problems in the commit messages of the fixes. However, manual analysis showed that assumption to be false for the Android as CVE fix was found which did not mention the CVE-ID. However, the ideas on improving the determining of the fix commits were proposed.

The mapping of the vulnerabilities to the fix locations was the basis for all the experiments done in this work. Because of this the perceived weaknesses of the mappings limit the accuracy of the experiments. In addition, the vulnerability reports are often hidden from the public till a widely available fix has been deployed. This adds delay to those data sources which makes continuous analysis challenging.

7. REFERENCES

[1] Backfried G, Schmidt C, Pfeiffer M, Quirchmayr G, Glanzer M & Rainer K (2012) Open source intelligence in disaster management. In: European Intelligence and Security Informatics Conference. Odense, Denmark, 254–258. IEEE Computer Society. DOI: http://doi.ieeecomputersociety.org/10.1109/EISIC.2012.42.

[2] Aramaki E, Maskawa S & Morita M (2011) Twitter catches the flu: Detecting influenza epidemics using twitter. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. Edinburgh, United Kingdom, 1568–1576. Association for Computational Linguistics. URL: http://dl.acm.org/citation.cfm?id=2145432.2145600.

[3] Weicheng Y, Beijun S & Ben X (2013) Mining GitHub: Why commit stops – exploring the relationship between developer's commit pattern and file version evolution. In: 2013 20th Asia-Pacific Software Engineering Conference. Bangkok, Thailand, volume 2, 165–169. DOI: http://dx.doi.org/10.1109/apsec.2013.133.

[4] Ben X, Beijun S & Weicheng Y (2013) Mining developer contribution in open source software using visualization techniques. In: Proceedings of the 2013 Third International Conference on Intelligent System Design and Engineering Applications. Hong Kong, 934–937. IEEE Computer Society. DOI: http://dx.doi.org/10.1109/isdea.2012.223.

[5] Gousios G, Kalliamvakou E & Spinellis D (2008) Measuring developer contribution from software repository data. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. Leipzig, Germany, 129–132. ACM. DOI: http://dx.doi.org/10.1145/1370750.1370781.

[6] Nagappan N, Ball T & Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering. Shanghai, China, 452–461. ACM. DOI: http://dx.doi.org/10.1145/1134285.1134349.

[7] Neuhaus S, Zimmermann T, Holler C & Zeller A (2007) Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. Alexandria, Virginia, USA, 529–540. ACM. DOI: http://dx.doi.org/10.1145/1315245.1315311.

[8] Shin Y & Williams L (2013) Can traditional fault prediction models be used for vulnerability prediction? Empirical Software Engineering 18(1): 25–59. DOI: http://dx.doi.org/10.1007/s10664-011-9190-8.

[9] Nordström H, Laitinen K, Lundelin M, Herrala J, Sjöblom J, Honkanen K & Nurminen M (2015) Guidelines for Developing Finnish Intelligence Legistlation URL: http://www.defmin.fi/files/3144/GUIDELINES_FOR_DEVELOPING_FINNISH_INTELLIGENCE_LEGISLATION.pdf. Accessed 25.1.2016.

[10] Fonseca J & Vieira M (2008) Mapping software faults with web security vulnerabilities. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC. Anchorage, Alaska, USA, 257–266. DOI: http://dx.doi.org/10.1109/dsn.2008.4630094.

[11] Shirey R (2007) Internet security glossary, version 2. RFC 4949, RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc4949.txt. Accessed 30.3.2016.

[12] Martin B, Brown M, Paller A & Kirby D (2011) 2011 CWE/SANS top 25 most dangerous software errors URL: http://cwe.mitre.org/top25/. Accessed 13.4.2016.

[13] Viide J, Helin A, Laakso M, Pietikäinen P, Seppänen M, Halunen K, Puuperä R & Röning J (2008) Experiences with model inference assisted fuzzing. In: Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies. San Jose, California, USA. USENIX Association. URL: https://www.usenix.org/legacy/event/woot08/tech/full_papers/viide/viide.pdf.

[14] Alhazmi O & Malaiya Y (2006) Prediction capabilities of vulnerability discovery models. Reliability and Maintainability Symposium: 86–91. DOI: http://dx.doi.org/10.1109/RAMS.2006.1677355.

[15] Zhang S, Caragea D & Ou X (2011) An empirical study on using the National Vulnerability Database to predict software vulnerabilities. In: Database and Expert Systems Applications: 22nd International Conference. Toulouse, France, 217–231. Springer. DOI: http://dx.doi.org/10.1007/978-3-642-23088-2_15.

[16] Glanz L, Schmidt S, Wollny S & Hermann B (2015) A vulnerability's lifetime: enhancing version information in CVE databases. In: Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business. Graz, Austria, 28:1–28:4. ACM. DOI: http://dx.doi.org/10.1145/2809563.2809612.

[17] Duraes JA & Madeira HS (2006) Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering 32(11): 849–867. DOI: http://dx.doi.org/10.1109/tse.2006.113.

[18] Brumley D, Poosankam P, Song D & Zheng J (2008) Automatic patch-based exploit generation is possible: Techniques and implications. In: IEEE Symposium on Security and Privacy. Oakland, California, USA, 143–157. IEEE. DOI: http://dx.doi.org/10.1109/sp.2008.17.

[19] Avgerinos T, Cha SK, Hao BLT & Brumley D (2011) AEG: Automatic exploit generation. In: 18th Annual Network and Distributed System Security Symposium. San Diego, California, USA. URL: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_5.pdf.

[20] Bostock M (2011) Cube: Time Series Visualization URL: https://corner.squareup.com/2011/09/cube.html. Accessed 25.1.2016.

[21] Sallings D (2012) Seriesly - document oriented time series database URL: http://dustin.sallings.org/2012/09/09/seriesly.html. Accessed 25.1.2016.

[22] Mitre (2015) Common Vulnerabilities and Exposures URL: https://cve.mitre.org/. Accessed 21.12.2015.

[23] Cheikes BA, Waltermire D & Scarfone K (2011) Common Platform Enumeration: naming specification version 2.3. NIST Interagency Report 7695 DOI: http://dx.doi.org/10.6028/nist.ir.7695.

[24] Mell P, Scarfone K & Romanosky S (2006) Common Vulnerability Scoring System. IEEE Security & Privacy 4(6): 85–89. DOI: http://dx.doi.org/10.1109/msp.2006.145.

[25] National Vulnerability Database (2015) URL: https://nvd.nist.gov/. Accessed 30.12.2015.

[26] CVE Details (2016) URL: https://www.cvedetails.com/. Accessed 10.1.2016.

[27] Open Sourced Vulnerability Database (2016) URL: http://osvdb.org/. Accessed 10.1.2016.

[28] Fette I (2008) Google Chrome, Chromium, and Google URL: https://blog.chromium.org/2008/10/google-chrome-chromium-and-google.html. Accessed 21.12.2015.

[29] Chromium Git Migration FAQ (2014) URL: https://www.chromium.org/developers/chromium-git-migration-faq. Accessed 10.1.2016.

[30] International Data Corporation (2015) Smartphone OS Market Share, 2015 Q2 URL: https://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed 10.1.2016.

[31] Android is now Open Source (2008) URL: http://android-developers.blogspot.fi/2008/10/android-is-now-open-source.html. Accessed 10.1.2016.

[32] Android Open Source Project (2016) URL: https://source.android.com/. Accessed 10.1.2016.

[33] Drake J (2015) Stagefright: Scary Code in the Heart of Android URL: https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf. Accessed 10.1.2016.

[34] Node.js (2015) URL: https://nodejs.org/. Accessed 21.12.2015.

[35] D3.js - Data-Driven Documents (2015) URL: https://d3js.org/. Accessed 21.12.2015.

[36] Bray T (2014) The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc7159.txt. Accessed 30.3.2016.

[37] ECMAScript 2015 Language Specification – ECMA-262 6th Edition (2015) URL: http://www.ecma-international.org/ecma-262/6.0/. Accessed 21.12.2015.

[38] Babel · The compiler for writing next generation JavaScript (2015) URL: https://babeljs.io/. Accessed 21.12.2015.

[39] Git (2015) URL: https://git-scm.com. Accessed 21.12.2015.

[40] Chacon S & Straub B (2014) Pro Git. Apress, 2nd edition. URL: https://git-scm.com/book/en/v2.

[41] CoffeeScript (2015) URL: http://coffeescript.org/. Accessed 21.12.2015.

[42] The jQuery Foundation (2015) URL: https://jquery.org/. Accessed 21.12.2015.

[43] CVE-search - a tool to perform local searches for known vulnerabilities (2015) URL: https://cve-search.github.io/cve-search/. Accessed 21.12.2015.

[44] Evans C, Hawkes B, Adkins H, Moore M, Zalewski M & Eschelbeck G (2015) Feedback and data-driven updates to Google's disclosure policy URL: http://googleprojectzero.blogspot.fi/2015/02/feedback-and-data-driven-updates-to.html. Accessed 17.1.2016.