Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

The Edit History of the National Vulnerability Database

and similar Vulnerability Databases

Mathias Karlsson

Master's Thesis MA-2012-06
February until August 2012
Advisor: Dr. Stephan Neuhaus
Supervisor: Prof. Dr. Bernhard Plattner

Acknowledgements

First I want to thank my tutor Dr. Stephan Neuhaus for his expertise and supportive advice as well as the large freedom and up front feedback he gave me during this thesis. I also want to thank Prof. Dr. Bernhard Plattner for his feedback in the intermediate reviews and presentations.

I also thank my mother enabling me to study all this time and providing the help whenever it was needed. Lastly, a special thank goes to my wife Jagoda Karlsson for her support and motivation during this thesis. I am honored to dedicate this thesis to them.

Abstract

When working with software vulnerabilities one has to deal with information of many different sources. These might relate to the same vulnerability but since every institution and vendor have their own naming conventions it might not be easy to connect them. This is where the Common Vulnerability Enumeration comes in. The institution running it issues standardized names to discovered vulnerabilities and also gives references to external resources associated with them.

The National Vulnerability Database is the vulnerability database based on the list of CVE entries. This database is often regarded as the ground truth when it comes to software vulnerabilities. To fulfill this task, the data of the published entries has to be stable, but it was discovered that some content experiences changes over time.

Therefore the primary goal of this thesis is to answer the question whether the National Vulnerability Database can be used as the ground truth despite its edits. The secondary tasks are to collect the data of a second vulnerability database for comparison, and develop the tools to build, analyze and compare the edit history.

In total, over a year of data of the National Vulnerability Database and nearly four months of data of the Open Source Vulnerability Database were analyzed. The results exhibit very similar patterns of when and which entries are edited. The majority of the changes are performed in data fields regarding vulnerable applications and references. These changes are not very substantial since the data concerning the vulnerability does not change. Additionally, the majority of the edits, non-substantial and substantial, are performed within the first month after the entry is released.

Concluding, the National Vulnerability Database can be used as the ground truth for not very fresh entries and the Open Source Vulnerability Database is a good source for additional information as well as a second opinion.

Contents

List of Figures

Chapter 1

Introduction

Working with software vulnerabilities means dealing with many different data sources, since there is no one database collecting all the information of each discovered software vulnerability. Finding the necessary information is also not very easy because every institution or vendor running a vulnerability database has its own way of naming and categorizing the vulnerability. This is where the Common Vulnerability and Exposure (CVE)[1] comes in. It offers a standardized name for software vulnerabilities and references to advisories in well-known vulnerability databases. A CVE Numbering Authority[2] can issue a new name to a newly discovered vulnerability. These CVE identifiers are widely adapted and greatly help finding information to a CVE registered vulnerability.

Even if the related information available can be retrieved much easier with the CVE identifier, the information that can be found in different sources have different formats or different names for the same information. This makes automatized processing very complicated. The National Vulnerability Database[3], which is based on the list of CVE identifiers, is an attempt to offer software vulnerability information in a standardized way.

The National Vulnerability Database does not collect all the available information from all available sources since many sources contain a lot of very specific vendor information. A vendor advisory usually describes what the specific effects of the vulnerability on the products are and how it can be resolved. Each National Vulnerability Database entry is, therefore, rather the least common denominator of the sources in a standardized format. Beside the CVE description and references, it also holds a threat rating, a list of vulnerable software and a Common Weakness Enumeration which categorizes the vulnerability.

Over the time this database turned into the de facto standard vulnerability database. It is widely used by vendors for their products. But it is also used by researchers to perform software vulnerability research and often regarded the ground truth. Related research topics are for example predicting software vulnerabilities [1, 2], relation of vulnerabilities to package dependencies [3], identifying trends [4] or studying the vulnerabilities of a specific product [5].

[1] http://cve.mitre.org
[2] http://cve.mitre.org/cve/cna
[3] http://nvd.nist.gov

To perform such research it is important that the data the research bases on is stable and reliable which is necessary for the results to be reproducible. Since the National Vulnerability Database is regarded the ground truth these properties are associated with it. But during research it was observed that changes can occur over time. The question is if these changes are so severe that the database cannot be used as ground truth anymore.

The goal of this thesis is to determine whether the National Vulnerability Database can be used as the ground truth despite its edits. Such an analysis was never done before.

This is done in three steps. First, the data has to be collected and archived frequently. Second, the edit history has to be created by extracting the differences between the successive revisions. Third the edit history has to be analyzed. In the Chapters 2 and 3 the data sources and techniques used as well as the questions that have to be answered are described. The Chapter 4 shows the results which answer the question raised in Chapter 3 and Chapter 5 presents the conclusion whether the National Vulnerability Database can be used as the ground truth for software vulnerabilities.

Chapter 2

Data Sources and Collection

In the first part of this chapter, we present the data sources we used and why we chose them. In the second part, we also present the data collection process.

2.1 Data Sources

The analysis of the edit history of the National Vulnerability Database will yield results. But without another reference it will be very hard to interpret these results. Therefore a second, independent data source is needed that is as similar as possible to the National Vulnerability Database to compare the obtained results. Otherwise it is unclear if the observed edits are severe changes or just necessary updates while maintaining such a database.

2.1.1 National Vulnerability Database

The National Vulnerability Database (NVD)[1] is a software vulnerability database that has been publicly available since 2005. The goal of the NVD is to provide a "repository of standards based vulnerability management data to enable automation of vulnerability management, security measurement and compliance"[2]. The NVD is often regarded as the ground truth for software vulnerabilities and is widely used by researchers and corporations. It contains over 50'000 vulnerabilities and claims to grows by of 13 entries per day, on average. The Computer Security Division of the National Institute of Standards and Technology (NIST)[3] maintains the NVD.

The data is based on the Common Vulnerabilities and Exposures (CVE)[4] entries published by MITRE[5]. Since 1999 MITRE catalogs software vulnerabilities and assigns common names to make it easier to share data since most vendors have their own naming systems which are not compatible. Such a CVE entry therefore only contains a short description and external references (e.g., other vulnerability databases, vendor advisories, etc.). A corresponding entry in the

[1] http://nvd.nist.gov
[2] http://nvd.nist.gov/about.cfm
[3] http://www.nist.gov
[4] http://cve.mitre.org
[5] http://www.mitre.org

NVD provides additional information such as impact rating, categorization of the vulnerability and a list of vulnerable software. The NVD database only contains entries associated with filled-out CVE entries; therefore a reserved CVE entry will only appear in the NVD after the information was disclosed (e.g., by the vendor when releasing the patch) and the CVE entry was updated. The NVD can be searched online and is also published daily as database export in XML format.

Until the beginning of 2012 no entry creation process was published. MITRE now publishes the process they use to create a CVE entry which consists of three stages[6].

First is the processing stage where in the first phase data is collected and converted into a standardized format. This is then matched to submission groups. In the refinement phase a content team member analyzes the submission group and decides whether a CVE entry should be created or not. If this is the case he creates the entry which is then reviewed by the CVE editor in the editing phase.

Second is the assignment stage where the CVE identifiers are assigned. This can be done in one of three ways: 1. they are defined by the content team, 2. they are reserved previously by an organization or individual, 3. they are created by the CVE editor to quickly create a CVE identifier.

The final stage is the publication which consists of a publication and a modification phase. In the publication phase the new CVE entry is added to the CVE website. This is followed by the final modification phase where the entry can be edited in simple ways such as clarifying the description or adding more references. Finally, it also mentions serious changes to the CVE list such as splitting an entry into multiple entries, combining entries and even deleting entries.

Both, MITRE and the Computer Security Devision of NIST, are sponsored by the Departement of Homeland Security of the U.S. government.

2.1.2 Open Source Vulnerability Database

The second data source should be as similar as possible in the number of vulnerabilities, in the goal it wants to achieve and in the information available per vulnerability. Many of the vendor based databases do not qualify in the goal aspect. They are often very user-oriented and cover problems associated with their own software and how it can be fixed. This even is true for security related vendors such as Secunia. They for example do not focus on the vulnerability rather than focusing on the vulnerable product, therefore it is possible that one CVE identifier points to multiple Secunia advisories. Furthermore it is in general very hard to gain access to vendor based repositories since they earn their money with this information. Other public databases have a very different goal (e.g., document exploits) or are simply not comparable in size.

[6] http://cve.mitre.org/cve/identifiers/build.html

The best match that could be found was the Open Source Vulnerability Database (OSVDB)[7] of the Open Security Foundation (OSF)[8]. The OSVDB was founded in August 2002 and thus predates the NVD. The project was started with the aim to provide "an independent and open source" vulnerability database with the goal "to provide accurate, detailed, current and unbiased technical information about all types of vulnerabilities." The project also wants to "promote greater, more open collaboration between companies and individuals, eliminate redundant works and reduce expenses inherent with development and maintenance of in-house vulnerability databases"[9].

The goals of both databases are very similar. Also the entry content is very comparable as well as the size, with over 80'000 entries the OSVDB is even larger than the NVD. When looking at the information given for each entry for most fields in the NVD a matching field in the OSVDB entry can be found.

The following fields of the NVD can be matched directly to fields in the OSVDB:

1. The vulnerable-software-list to the products.
2. The CVSS elements to the information in the CVSSv2 Score.
3. The references to the references.
4. The summary to the description.

No match can be found for:

1. The vulnerable configuration elements.
2. The published date since the OSVDB only states very roughly how long ago the entry was added.
3. The last modified date since the OSVDB only states very roughly how long ago the entry was edited.
4. The common weakness enumeration (CWE).
5. The OVAL scanner information.

On the other hand an OSVDB entry also can contain additional information that is not present in the NVD:

1. A timeline is given for each vulnerability covering critical dates such as the disclosure date, the exploit date or the patch availability date.
2. A classification gives a basic definition of the vulnerability by the parameters location (attack vector), attack type, impact, available solution type, exploit availability and by whom it was disclosed.
3. Further technical information regarding the vulnerability can be added.

[7] http://osvdb.org
[8] http://www.opensecurityfoundation.org
[9] http://www.osvdb.org/about

4. A solution to fix the vulnerability can be recommended.
5. A list of filters and tools that are able to scan for this vulnerability with their respective filter IDs might be present.
6. Sometimes also some manual-testing-notes that can be used to manually check whether a system is vulnerable or not, are published.

Regarding all this information, we are therefore confident that the OSVDB is the best choice to compare the results of the NVD with.

2.2 Data Collection

The data collection is the process of getting the data from the data source and archiving it in a version preserving manner. This section describes how the data is collected of the different data sources and subsequently stored for further processing.

2.2.1 NVD Data Collection

The data collection of the NVD was not part of this thesis since it was already previously implemented by Stephan Neuhaus but is still described here for completeness. The NVD is published daily as XML files on the NVD website. The entries are split into several files by the year, that is also part of the CVE identifier. The file of the year 2002 also contains the entries of the previous years. These files are downloaded daily automatically. In a second step the XML elements are sorted. This is performed to minimize syntactic changes without actual content changes. This then reduces the changes committed to the SVN repository which is done as a final step. The data is being collected since 12. July 2011.

The used sorting tool actually achieves its goal only partially. The idea is, that the elements at the same XML tree depth and the arguments within each element are sorted. The sorting tool manages to sort the arguments within one element. It does not manage to sort elements sharing the same XML tree depth by their tag name. It is therefore not surprising that more complicated sorting, such as the sorting of elements with the same tag name by argument values, by information within sub-elements or by the text node, is also not performed. This theoretically leaves a great potential for superficial changes that would be checked in to the SVN repository. In reality the data export is usually done by the same process every day, keeping the order the same as on the previous day and therefore should not be of serious concern.

2.2.2 OSVDB Data Collection

The initial idea was to collect the data from the OSVDB entries in the same way as it is done in the case of the NVD by simply downloading the available daily database export. But as of February 2012 the data exports are not available for download any more. This is because the whole website of the OSVDB is

undergoing a major reconstruction. The link to the SQL dumps shows an error page whereas the link to the XML exports shows an empty page.

As of this moment the website that previously offered the exports is non-existent. The new website on the other hand offers a beta version of an API that allows registered users to request the details of one entry by its identifier and receive the result in XML form. To fetch all the information through this API would be relatively easy if there would not be an access limit of 100 entries per day per registered user. We contacted the maintainers explaining our interests and intentions and asked for alternative ways to access the information of the database, like some kind of database export or removing the limitation of the API. In both cases the maintainers' answer was that they simply do not have the resources to process such a request, and therefore might be only pursued after a larger donation.

Since there is no equal alternative to the OSVDB the only option left was to download the information directly from the website which delayed the data collection significantly. To start with the data collection as fast as possible the first version downloaded and stored the complete HTML information retrieved for each entry. Due to the amount of data collected it was deemed necessary to split the resulting file into smaller files. The idea was to do this according to the NVD files, meaning that each entry would be stored in the file according the to the year associated with it. To be as consistent as possible with the NVD, each OSVDB entry was checked for a reference to a CVE identifier and if one was found the whole entry was stored in the annual file associated with this CVE identifier.

This seemed to be a good approach because it should have been relatively easy to find and compare entries between the NVD and OSVDB. But this caused entries without a reference to a CVE identifier to be dropped. At first this did not seem to be a problem since the goal was to compare the OSVDB to the NVD which only contains entries related to CVE identifiers. However when the reference to a CVE identifier is added after the entry is published, it causes problems because with this assumption the entry shows up the first time when the reference was added and not when the entry was actually added. It was therefore necessary to also store all the entries without a reference to CVE identifiers. Additionally the discovery that it is possible for OSVDB entries to also reference to multiple CVE identifiers proved that the assignment of OSVDB entries to year files was very arbitrary. It would have been better to store all the entries in one file or, if necessary, split into smaller files based on the OSVDB identifiers. At the moment this problem was discovered, it was already clear that the data of the OSVDB entries had to be preprocessed before creating the edit history. The problem can be addressed during preprocessing in order to not loose the already collected data.

Preprocessing is necessary because the downloaded and stored entry with the complete HTML document consists mainly of HTML code and very little useful information (e.g., vulnerability rating, description, etc.). Additionally, the already developed tool to create the edit history for the NVD is based on XML files. This tool could be used directly on the complete HTML version of the entry if it were a correctly formatted XHTML document but it would

take much longer and produce a lot of undesired output (i.e., changes in the HTML structure) since also the HTML code is compared. To prevent this from happening the useful information has to be extracted from each downloaded HTML OSVDB entry and stored in an XML file similar to the NVD export XML files. This process requires multiple steps.

First some general data like the download date are written to the output file. Then the data of the entries are processed. For each entry, initially all the data of the entry is read from the file that contains the complete entry in HTML form. In the next step unnecessary HTML elements such as java scripts or elements with non-unique attributes are removed. Next some special characters that cause problems during the processing are replaced by non-hazardous replacements. The interesting information can now be extracted. This is done by using XPath because it allows direct access to a specific XML element. The found information is written as XML elements to the output file. After the entry is processed for all specified information the data of the next entry is read. This is repeated until all the entries are processed. During the data extraction the information of all the entries is written to the same output file. This also resolves the problems caused by storing the OSVDB entries to multiple files based on the encountered CVE reference.

The download and the processing task turned out to be very time consuming. This is because the download is limited by the download delay whereas the information extraction is mainly limited by the processing performance. The initial versions performed their tasks in sequence and it could take up to a day to download and process the entries. The idea to resolve this was to download the entries in chunks which allows to start the information extraction while other downloads are still in progress.

Using the multi threading offered by Perl turned out to be very memory consuming and the system started to thrash. An implementation relying on separated scripts is much more reliable and works just as fine. With this the runtime was shortened significantly. The final version runs multiple downloads and extractions in parallel which reduces the time even more to just a few hours. This tool fetches and processes all the entries of the OSVDB every day and commits the resulting XML file to the SVN repository.

Chapter 3

Methods

In this chapter we present the ideas and tools used to create, analyze and compare the history of the vulnerability databases.

3.1 History Building

In this section we describe the methods and tools developed and used to build the history files. The goal of this task is to create an edit history of a file of which multiple revisions are available. The history's purpose is to document for each entry *what* was edited and *when*.

3.1.1 Differences between two XML Documents

Determining the difference between two XML documents is no easy task. Without any prior knowledge of the specific structure of the document every element has to be compared with any other element. In this case finding the differences between the two documents is essentially finding the minimum edits required to transform one document into the other one. Since the elements in a XML document can also be interpreted as a tree and each element is labeled with a tag, it is possible to reduce this problem to the the problem of the *edit distance on unordered labeled trees*. This problem is actually NP-complete [6, 7].

For small XML documents which are only a few kilobytes big, this comparison is performed fairly quickly since there are not that many elements to compare. But in case of much larger documents with several hundred thousand element this comparison can consume a significant amount of time.

3.1.2 Building the History of a versioned XML File

The edit history of a versioned file is basically the sum of the changes between two consecutive revisions over all the revision. The subversion repository already works in a similar manner since it only stores the differences between two versions of a file. It offers the function *svn diff* to display these differences between two given revisions.

But like the *diff* tool, the *svn diff* function of subversion cannot fulfill the requirements [1]. These tools are line-based which means that the result will tell us which lines are different in the "left" and the "right" file. This is simply not enough information since we have no idea to which entry that line actually belongs. Additionally, in an XML file the order of elements at the same depth of the XML tree or attributes of an element does not matter, but for the *diff* tools this order matters because they are not aware of the XML code and treat the line as string.

The next step was to look for tools that are specifically designed to compare XML files. The freely available tools do not seem to be designed for larger files and tend to take very long. This is probably due to their generic design which forces them to compare each element with every other element because they cannot rely on a specific document structure. Such information could be given to some tools with a Document Type Definition (DTD). But even without such information at least the order of the elements with the same XML tree depth will not affect the result. This is already much better but they still cannot satisfy the requirements since neither the *when* nor the *which entry* question are answered since the information is contained within certain attributes.

Due to these reasons it was decided that the best approach is to develop our own tool that extracts the difference between two NVD XML files and documents the edits according to the requirements. This tool cannot extract the difference between any pair of XML files but is aware of the specific structure, and therefore can extract the necessary information.

The basic idea is to compare each *entry* with the corresponding *entry* of the other revision. Each sub-element of the entry that matches completely in both revisions is removed from the document. After all the entries are processed all the elements that can be only found in the newer revision were added, all the elements that can be only found in the older revision were removed and finally all the elements that can be uniquely identified in both revisions of the entry but the text content does not match, are changed. This defines the *edit type*, the *when* can be extracted of the date attribute in the root element, the *where* is defined by the entry identifier and the *tag* by the currently processed element and finally the *what* by the content of the element. A detailed description can be found in Section 4.1.1.

With this information it is possible to create an edit entry that fulfills all the given requirements. This comparison has to be performed between every revision with its predecessor to create the complete history.

For the implementation a combination of a shell script, java and XML document object model (DOM) were selected. This choice was made because bash, java and DOM were already familiar to the author from previous projects and were most likely to lead to success. The performance is not of utmost importance, and therefore the expected significant memory requirements of DOM (eight to fifteen times the documents file size per document) does not matter.

The developed tool is able to build the complete history of all the NVD files

[1] Even though *svn diff* does not fulfill the necessary requirements it can still be used to get an idea of the maximum size of the edit history.

over all the revision within one to two days. This is done by distributing the workload on multiple machines. This is sufficient since it is not necessary to update the history every day.

3.1.3 Building the History of a large OSVDB XML file

The building of the OSVDB history took significantly longer than the NVD even with fewer revisions. This is mainly due to the fact that all the OSVDB entries are contained in one large file. This problem can be resolved by distributing the workload on multiple machines that each performs the comparisons of a set of revisions. After all the machines are done the results can be combined to the complete history. With this approach every time the history is built, it is built from scratch.

This is actually not necessary since the history of the revisions processed by the previous history build does not change. It is, therefore, only necessary to process the revisions that the previous history build did not process and append the result to the history of the previous run. With this approach it is possible to build the history of the OSVDB with as few changes as possible to the general history building process. As an added bonus, when the history building is done on a daily basis in such an incremental way, it not only keeps the runtime at a minimum but also offers always an up-to-date version.

3.2 History Analysis

After creating the histories one can start to analyze them. Initially it is important to get a good overview. In a second step topics of special interest are selected and investigated deeper. Such topics can be for example looking for special patterns or a certain behavior. In each step, first, questions have to be defined which then have to be answered by analyzing the data.

Since the different data sources are in the end processed by the same tool to create the history, each edit entry is written in the same unified format. Therefore, it is largely possible to use to the same tools to analyze the histories of the NVD and the OSVDB.

3.2.1 Preliminary Analysis

To create an overview the following questions were defined:

- How many edits are there per entry?
- On how many days was the entry edited?
- Are there any database entries with significantly more edits than others?
- How many edits are there per day, per month?
- How many entries were edited per day, per month?
- Are there any dates with significantly more edits than others?

- Which data (XML element) was edited?
- Are there certain days in the month with increased activity?
- How does it look when the largest edits are not present?
- When are new entries added?

To perform the different evaluations quickly another tool was developed. It not only takes care of all the arguments for the different operations performed on the history (one execution of the evaluation script is one operation) but also allows a certain degree of preprocessing. The tool is split into three components which are the operations script, the config file and the operations file.

In the config file the arguments for the script are defined which will be the same for all the operations (e.g., directory of the history, grep pattern for the history files, output directory, etc.).

The operations script reads the information of the config and operations file, preprocesses the history and then runs the evaluation script.

The operations file holds one or many operations which will be executed by the operations script one after another. Each operation is defined by a set of argument required to once run the evaluation script (e.g., grep pattern for history filtering, output file name, output format, etc.).

With this tool it is possible to easily create new operations or multiple variations of existing operations. Since all the operations can be defined in one operations file all the history evaluation files can be updated at once. This convenience comes at the cost of time since all the operations are performed even if only a few might be needed. This can be avoided by creating another operations file that only contains the wanted operations. After testing the edited operations can be added to the default operations file.

3.2.2 Targeted Analysis

During the targeted analysis certain selected topics are investigated in more detail. These can be specific patterns or a deeper analysis of a previously discovered phenomena. The following questions fall into this category:

- After how many days was the entry last edited?
- Is it possible to see the Microsoft patch day?
- Can we detect edit wars?
- How often are severities reassessed?
- What happened to deleted entries?
- Are the number of unique entry edits corresponding with the number of last modified date time edits?

These topics often require some very specific analysis which cannot be performed by the default history evaluation script and adding the functionality to it would only complicate it.

Depending on the problem the best approach is either to reduce the history to the interesting data and then inspect it manually or by writing a script that investigates the specific problem.

To allow the execution of these specific scripts by the operations tool described previously, it had to be more generalized since it was initially only designed to be used in junction with the default evaluation script.

The main change was that each operation also specifies the evaluation script that will be called to perform the analysis and that in the configuration file the directory path to all the evaluation scripts was added. With these changes it is now possible to execute many different operations at once.

3.2.3 Visualization and Interpretation

The evaluations write the results to text files. Normally these hold multiple numbers per line where each line stands either for a date or an identifier. These files can be imported to MATLAB for further processing and visualization.

MATLAB was chosen for this task because it is very versatile offering a wide range of tools and is familiar to the author which minimizes the required work.

The interpretation usually starts with finding patterns in the figure produced by the MATLAB script of the evaluation results. This usually raises follow-up questions which need more details. Relatively simple questions can be answered by checking the human-readable version of the evaluation result with a text editor since it contains additional information.

More complicated questions require manual inspection of the history or partial history. This can also be done directly by using a text editor. But when a specific type of edit or a specific entry has to be investigated, it is usually much easier to use *grep* to get the desired information. The result can be written to a file and inspected by the tool of choice or directly forwarded to another tool (e.g., word count).

Very complex questions that require multiple steps of processing are best solved by a specially designed script. This takes initially more time but subsequent evaluations are much faster.

3.3 History Comparison

The goal of the history comparison is to find out what the similarities and the differences between the databases are. We do this primarily to check whether the found patterns in the NVD are simply a byproduct of maintaining a vulnerability database. The differences might give some insight into how the data is processed or into the working pattern.

Comparing the results of the two vulnerability databases is mostly straightforward since the same questions are answered. This is done by comparing the general pattern as well as exceptional cases. During this process normally additional questions came up relating to similarities or discrepancies. Simple

questions could be answered by looking up details in the human-readable evaluation results whereas more complex ones require a specific analysis or manual inspection.

Another approach is to use a specific data set for which also information of another source exists to compare the vulnerability databases. This is shown on the example of a Mozilla patch day in Section 4.3.6. Mozilla is a good choice because they publish vulnerability information of their own products. With this additional information it is possible to compare the vulnerability databases to a third reference.

Chapter 4

Results

In this chapter we present the results by the described methods. Each section of Chapter 3 has a corresponding section in this chapter. The data used for the results is the version available on 24th of July 21:50 on revision 9635.

4.1 History Building

First, we present some general information about the histories of the NVD and the OSVDB to give a brief overview over the built histories. It is then followed by a discussion regarding how the different types of edits within an XML element can have a different impact on the history. This is important to keep in mind for the interpretation of the results in the following sections.

4.1.1 History Format

Since all the histories are created by the same tool, they share the same format. The history consists of multiple entries, each logging an edit in one XML element. A history entry has the following components:

1. The *edit type* describes the type of edit performed. The edit type *add* means that the whole element was added whereas *remove* means that the element was removed in the revision being compared. The edit type *change* means that only the information in the text node contained within the XML element was edited.

2. The *id* field specifies in which entry the edit took place.

3. The *tag* field specifies which XML element was edited. Since it is only the tag name, it does not have to be unique.

4. The *date* field specifies on which date the edit took place. The information is taken from the attribute of the XML root element that either holds the publish date of the database export or the download date of the entries.

5. The *data* field holds the complete XML element for the edit types *add* and *remove*. In case of the edit type *change* the field *data_old* holds the information of the XML element in the older revision and the field *data_new* holds the information of the XML element of the newer revision of the file.

Together these fields document *how* was in *which entry*, in *which element*, *when*, *what* information edited.

4.1.2 Overview of the NVD History

The first time the database exports were downloaded was on the 12th of July 2011. Therefore, all the entries present at that date show up the first time in the history which can be seen by the large number of adds of the type element. The export file for the entries of the year 2012 was added on the 29th of February 2012 containing the export published on the 28th of February.

Since the 12th of July 2011 a total of 4573 new entries were added to the database which is an average of 12.1 per day and 579770 of elements (the reference as a whole is treated as one element) were edited which gives an average of 1658.1 per day. When looking at the edit types 264302 elements were added, 170868 were removed and 144600 were changed.

This means that in general information more information is added than removed. This is gets even stronger when the *attribute-edit* effect is considered which is discussed in Section 4.1.4.

4.1.3 Overview of the OSVDB History

The database was the first time crawled on the 5th of April 2012 but only the entries with a reference to a CVE identifier. The entries without a reference to a CVE identifier were downloaded the first time on the 16th of April 2012. Therefore, all the entries present on the 5th of April 2012 with a CVE reference will show up the first time as added on that date whereas the entries without a CVE reference until 16th of April 2012 will show up the first time then.

Since the beginning of the download a total of 3151 new entries were added which is an average of 28.9 new entries per day and 30412 of elements were edited which gives an average of 279 edits per day. Regarding the edit types the majority are adds with 25327 followed by removes with 2726 and changes with 2359.

This means that information is mainly added to the OSVDB and rarely deleted. The removes nearly vanish when the *attribute-edit* effect is considered which is discussed in the following.

4.1.4 The Impact of XML elements in the History

The impact of different edits within an XML element is discussed in this section. This is necessary because the elements of the XML files given to the history building tool of the NVD and OSVDB are different, therefore the impact of

the elements on the history is also different. The relevant XML elements are discussed for each database separately at the end of this section.

- When an XML element is not present in the older revision but present in the newer revision one history entry will be created stating that an edit of type *add* occurred.
- When an XML element is present in the older revision but not present in the newer revision one history entry will be created stating that an edit of type *remove* occurred.
- When an XML element can be uniquely matched by its hierarchical position and attributes in the older and newer revision but the text node within the XML element of the older revision does not match the text node within the XML element the newer revision one history entry will be created stating that an edit of type *change* occurred.
- When the attribute values of an XML element are edited it is not possible anymore to find a match because the hierarchical position, the tag and the attributes are used to uniquely identify the XML element. These requirements are necessary since some elements can exist multiple times and only the value of one attribute is different. Therefore, whenever the value of an attribute of an XML element is changed two history entries will be created, one edit of type *remove* (the element found in the older revision) and one edit of type *add* (the element found in the newer revision). In the following this effect will be referred to as the *attribute-edit* effect.

The edit of an attribute value is problematic because it will create two entries in the history instead of one. This is especially problematic when elements are storing the data in an attribute value, which is more likely to change, instead of using the text field. But there can be several reasons why one might decide to do that. The elements of each vulnerability database for which this is the case are now discussed in detail.

Critical XML elements of a NVD entry

Since the XML schema of the database exports is given, the only way to control the resulting history is by changing the definition of *edit types*. Therefore, it had to be accepted that edits in some elements can cause an *attribute-edit* effect. The position of the element in the XML document tree is given in a XPath like manner.

- The element *cpe-lang:fact-ref* at position *//entry/vuln:vulnerability-configuration/cpe-lang:logical-test/cpe-lang:fact-ref* holds the name and version of an application that is vulnerable. The same information can actually also be found one of the *vuln:product* elements at *//entry/vuln:vulnerable-software-list/vuln:product*, and is, therefore, redundant. When the *cpe-lang:fact-ref* is edited also the matching *vuln:product* should be edited.

Therefore, the *attribute-edit* effect can be negated by simply observing the edits of the *vuln:products*.

- The element *vuln:cwe* at position *//entry/vuln:cwe* holds the common weakness enumeration as an attribute. After a CWE number is assigned to a vulnerability, it seems unlikely that it will be removed completely again, so the edits of type *remove* of this element can actually be seen as *changes*.

- The *vuln:references* elements at position *//entry/vuln:references* are special cases. This is because there can be more than one reference with identical attributes since the details of the reference are stored within the sub-elements. Only with its sub-elements the reference is actually uniquely identifiable, and therefore it makes more sense to treat the *vuln:references* elements with its sub-elements as one whole element and had to be treated as an exception in the history building tool. It normally also does not make sense to remove references except for the case when an entry is deprecated when all the information of the entry is removed and one reference points to the new entry. The only other source for *removes* of *vuln:references* are when the reference is edited.

Critical XML elements of an OSVDB entry

Since the data of the downloaded entry is extracted by a tool developed in this thesis it was possible to avoid the *attribute-edit* effect to a certain degree but not entirely. To avoid more exceptions in the history building tool such as the NVD references it was important to avoid elements that can only be uniquely identified with its sub-elements. A special case is the information contained within the *Manual Testing* field where some code is presented to manually test the system for the vulnerability. Since this is actual exploit code, the string can contain characters that can be interpreted as XML code. When the string is stored as an attribute value these characters will be encoded in a way that will not break the XML structure. Several elements, therefore, contain information within attributes that can trigger the *attribute-edit* effect.

- The element *product* at position *//entry/products/product* holds the information of one vulnerable application. The information on the website is actually in three fields which are the name of the vendor, the name of the product and the version. The *attribute-edit* effect should in general not happen for this element since the names and version should not change. These values should actually only be edited to fix errors.

- The element *reference* at position *//entry/references/reference* holds the information of one reference. To contain the information of one reference in one element, despite the fact that multiple parts (name, link, etc.) building it up, the parts had to be stored within several attributes. As in the NVD it normally does not make sense to remove references and not working web links are normally flagged as unreachable. Therefore,

the main source for *removes* of this element will probably caused by the *attribute-edit* effect.

- The element *filter* at position *//entry/filters/filter* references to a vulnerability scanner or filter. Similar to the reference multiple data fields build up the filter element, and therefore it makes much more sense to store the data of each field in one attribute. Also filter elements in general should not be removed since the scanner seems to be able to scan for that vulnerability and it can be expected that the majority of the removes of this element is caused by the *attribute-edit* effect.

- The element *credit* at position *//entry/credits/credit* is either a simple text field or a reference to the OSVDB internal credit list. This reference to the credit list comprises different parts. To maintain one single element the data parts are stored in attributes.

- As already mentioned before the *manual testing* field is a special case because the string can contain special characters (e.g., null character), characters that will be executed (e.g., carriage return) or XML reserved characters (e.g “<”, “>”, etc.) which will be interpreted as XML code if they are printed as text node. Some characters have to be replaced by a description (e.g., null character) whereas others will be encoded when stored as an attribute value.

4.2 History Analysis

In this section we present and discuss the results to the questions raised in Sections 3.2.1 and 3.2.2. To visualize the results MATLAB was used for plotting. The results are presented for each database separately in the following subsections.

4.2.1 NVD Analysis

In Section 4.1.2 some very general information about the edits of the National Vulnerability Database was given. In the following the NVD history will be analyzed in more detail.

Which data (XML element) was edited?

This is a good question to start with to get an idea what is edited. Figure 4.1 shows the number of edits per tag. The y-axis uses a logarithmic scale.

Most edits can be found in the element *cpe-lang:fact-ref* closely followed by *vuln:product* which both hold the product information of vulnerable applications. The amount of changes of both elements should be actually the same but as mentioned *cpe-lang:fact-ref* is affected by the *attribute-edit* effect which causes two edit entries in the history instead of one. Since the information of these two elements in general matches, *vuln:product* will be used as reference and *cpe-lang:fact-ref* is only consulted when necessary. It is not

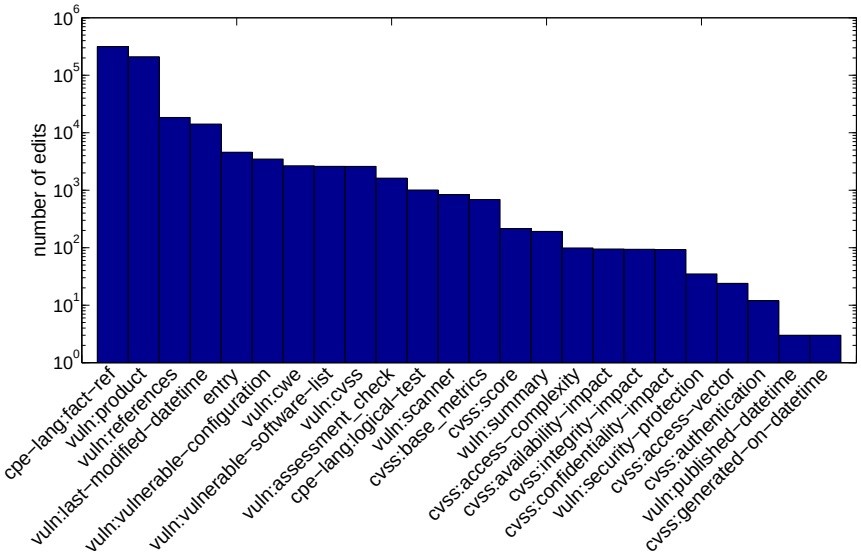


Figure 4.1: Tag edits in the NVD

surprising that these elements experience a lot of changes because more applications might be found to be vulnerable or that a certain version of an application is actually not vulnerable, and therefore removed from the list. There is also one special event that accounts for over 100'000 edits within the *vuln:product* element. It was decided that the common product enumeration of the Linux kernel should be changed from *cpe:/o:linux:linux_kernel:<version>* to *cpe:/o:linux:kernel:<version>* which required that every product in all the entries the Linux kernel was listed as vulnerable had to be modified.

It is also not surprising that the *vuln:references* are very high in the list because after a new entry is published other databases might pick it up, create their own advisory and the references in the NVD will be updated. Sometimes only the additional information of a reference is updated triggering an *attribute-edit* effect.

In the history are also many entries of the element *entry*. This is because every time a new entry is seen the first time is logged as such. The exceptions are three entries that were removed but shortly after readded. It is, therefore, safe to assume that these were export errors.

Whenever some data within an entry is edited the *vuln:last-modified-datetime* should be updated as well. It is, therefore, not surprising to find this element in the top five.

The elements *cpe-lang:logical-test* and *vuln:configuration* are super-elements of *cpe-lang:fact-ref* whereas *vuln:vulnerable-software-list* is the super-element of *vuln:product*. They can contain many elements that contain the information of one vulnerable product. It seems fairly common to add the whole list of vulnerable products short after the entry was first time published which caused many edits in these elements.

The element *vuln:cvss* is the super-element of *cvss:base_metrics*, *cvss:score*, *cvss:access-complexity*, *cvss:availability-impact*, *cvss:conficentiality-impact*, *cvss:-integrity-impact*, *cvss:access-vector*, *cvss:authentication* and *cvss:generated-on-datetime*. Whenever the whole CVSS rating is added later, an add of *vuln:cvss* appears in the history. This seems to be quite common whereas *removes* are very rare and only happen when the entry is deprecated. The sub-elements are never added or removed but sometimes modified.

The element *vuln:scanner* contains a reference to an open vulnerability and assessment language (OVAL) entry published by MITRE. During 2011 and the beginning of 2012 to about 800 entries this type of information were added but since 27th of January 2012 no changes were observed anymore. This indicates that the edits were updates to older entries and that current entries are directly published with this information.

The element *vuln:security-protection* describes the possible access type gained. It seems that this element is commonly used in older years reaching its maximum coverage in 2006 with over fifty percent. Since then it is declining and basically negligible in 2011 and 2012 because it is present in less than one percent of the entries.

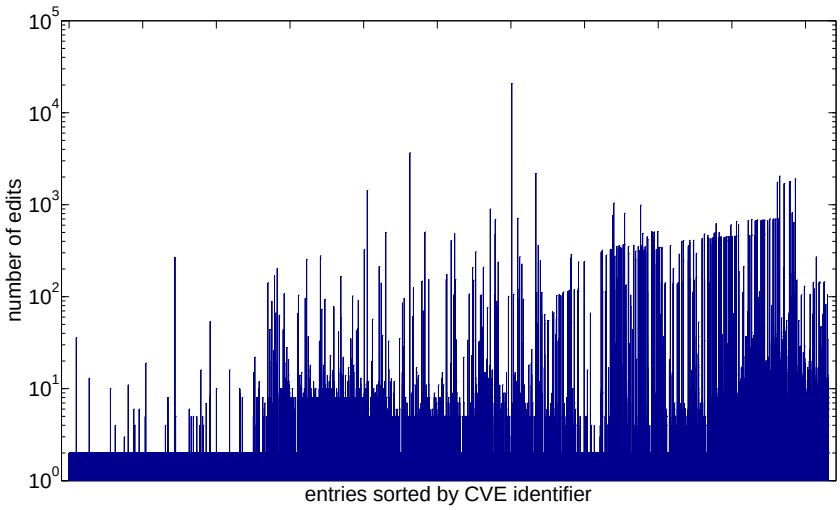
There are also two edits in *vuln:published-datetime* which can be safely assumed to be an error because this value should not be changed.

How many edits are there per entry?
On how many days was the entry edited?

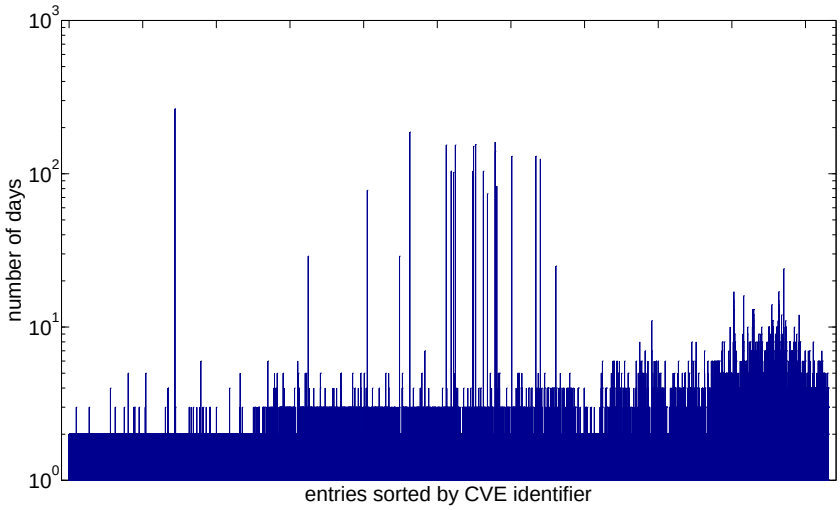
Each entry will appear at least once when it shows up the first time. The edits in the elements *cpe-lang:fact-ref*, *cpe-lang:logical-test* and *vuln:configuration* were dropped for these questions since the information is redundantly present in *vuln:vulnerable-software-list* and *vuln:product*. The only effect it would cause is that an edit of a product would show up as two or three edits instead of one, and therefore weigh more than it should.

The way the CVE identifiers are constructed allows them to be historically sorted since the identifier is by the year the vulnerability was discovered and a counter. Therefore, when plotted, the oldest entry will be on the left and the newest one on the right.

When thinking about the number of edits that an entry experiences one expects that older entries experience little to no edits whereas newer ones are changed more often because references, vulnerable applications might have to be added. The actual data is shown in Figure 4.2(a) with a logarithmic y-axis scale. This general expectation is correct but even some older entries experience many changes. The newer entries with many changes are often vulnerabilities that affect many applications or applications with many version (e.g., Google Chrome), and therefore adding a newly discovered vulnerable application causes many edits. This effect becomes less pronounced when we plot the number of days an entry was edited.



(a) Number of edits by entry



(b) Number of days the entry was edited

Figure 4.2: NVD edit statistics by CVE identifier

Figure 4.2(b) shows how many days the entry was edited. In general older entries are edited very seldom which also puts the observation of a large number of edits into a slightly different perspective meaning that edits happen rarely but if something is edited multiple elements might be changed.

A special case are the older entries that change up to several hundred times. These are likely to be errors since in general not much is edited but this is done very often. It can be for example that one day some products are added and the following day removed again. Another example is that the element *vuln:last-modified-datetime* is updated every day which leads to the conclusion that either a bad script updates the field wrongly or that there is actual data updated which is not published.

How many edits are there per day?
How many entries are edited per day?

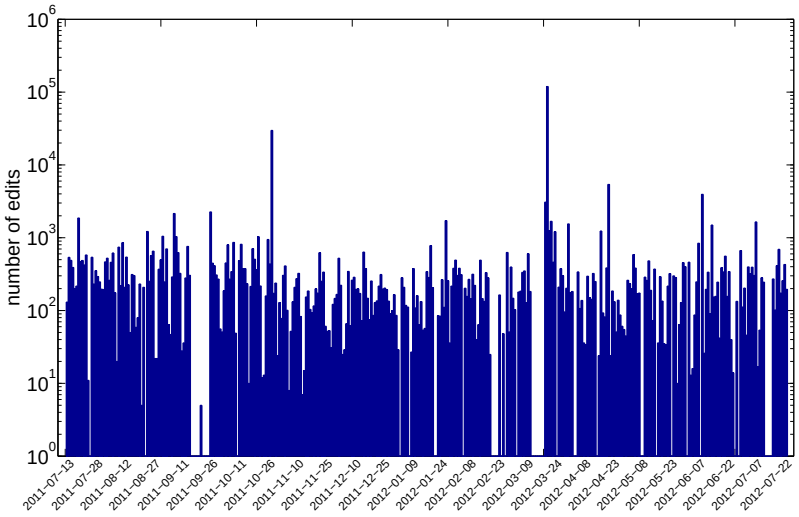
The edits in the elements *cpe-lang:fact-ref*, *cpe-lang:logical-test* and *vuln:configuration* were dropped for these questions since the information is redundantly present in *vuln:vulnerable-software-list* and *vuln:product*. The only effect it would cause is that an edit of a product would show up as two or three edits instead of one, and therefore weigh more than it should.

Figure 4.3(a) shows the number of edits per day over the time the database was downloaded. There are several days with zero edits. The reason for this can either be that no export was published or that the download failed. Sometimes there are even longer periods missing which indicates a more serious problem that stopped the download. These were caused by changed SSH keys or that the machine that should perform the download was down.

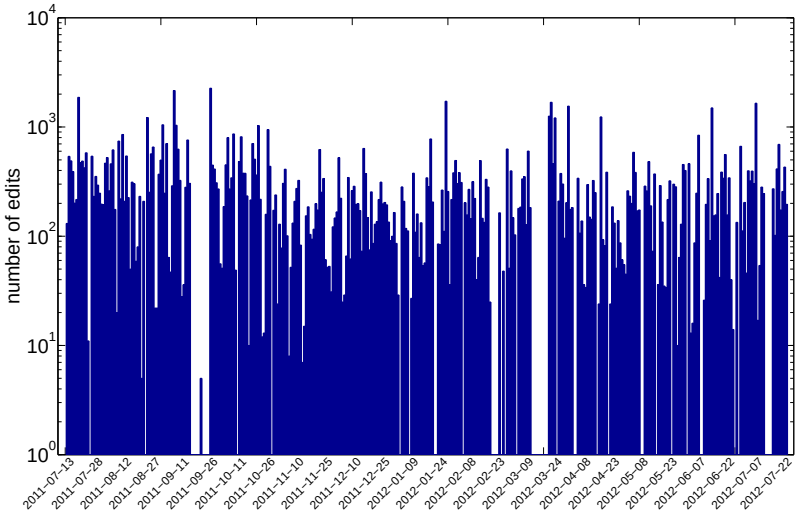
There are several days that stand out because they have many more edits than others. On the day with the most edits the Common Product Enumeration of the Linux kernel was changed. This caused that all the product information related to the Linux kernel had to be updated. In the days with the second, third and fifth most edits mainly Google Chrome is involved. Because it is released so frequently there are a few thousand versions. In cases when a vulnerability is published, to which Google Chrome is vulnerable, but not yet present in the vulnerable software list, the entry has to be updated. Since there are so many versions of Google Chrome this causes many edits. The fourth largest edit is caused by an update of several entries where the Opera browser version one to nine was added as vulnerable product.

These few days with a very large number of edits distort the results in further analysis and hence were removed in the upcoming datasets.

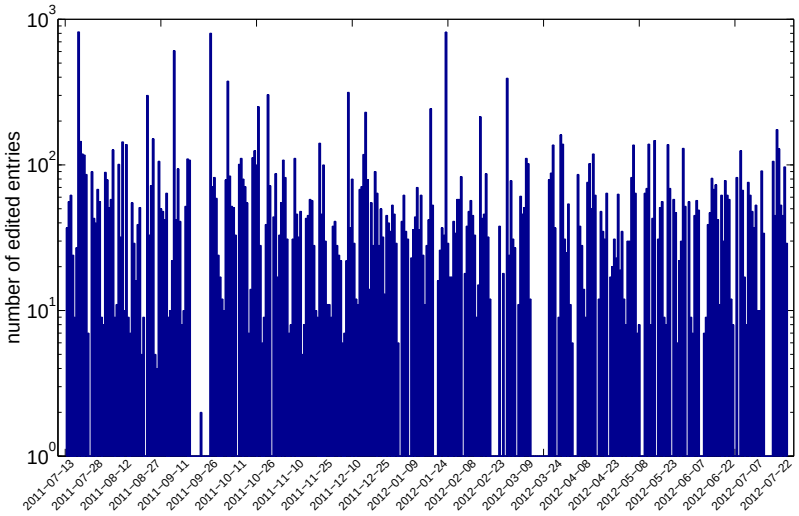
After removing the days with the five largest number of edits, the dataset is more even as can be seen in Figure 4.3(b). The number of edits of the majority of the days range from 100 to 1'000. When comparing the number of edited entries per day (Figure 4.3(c)) with Figure 4.3(b) the following can be deduced. Many edited entries on a day mostly cause a large number of edits on the same day. But many number of edits does not require that many entries were edited. This is because there are sometimes single entries which experience a large number of edited elements. Normally, these many edits is changed product



(a) Number of edits per day



(b) Number of edits per day without the top five edit days



(c) Number of entries edited per day without the top five edit days

Figure 4.3: NVD edit statistics per day

information in the vulnerable software list. In Figure 4.3(c) a repeating pattern can be found which consists of five days with a larger number of edited entries followed by two smaller ones. This lets suspect that this might be a weekly working pattern which is inspected further later in this Chapter.

Figure 4.4(a) shows the counts of the values of Figure 4.3(b) with a logarithmic x-axis. If there is a bar of height 3 at position 10 this means that 3 are three days with 10 edits. This graph supports the previous statement that the majority of the days have between 100 and 1000 edits and also days with 20 to 80 edits are also more likely to happen. But since the maximum of the y-axis is 4 and most of the bars have a height of 1, the distribution is very flat.

The picture is slightly different when looking at the count of figure 4.3(c) which can be seen in figure 4.4(b) with a logarithmic x-axis. If there is a bar of height five at position seven on the x-axis it means that there are five days where seven entries were edited. Here we find first a clear grouping around eight modified entries and a second concentration between 20 and 60. This shows that on most of the days the number of edited entries is either around eight or thirty and rarely more than one hundred entries are edited on one day.

These two groupings observed in both figures are two different kind of work days. The groups with less edits are mostly edits conducted on weekend days. The second groups with more edits and more edited entries are changes performed on work days.

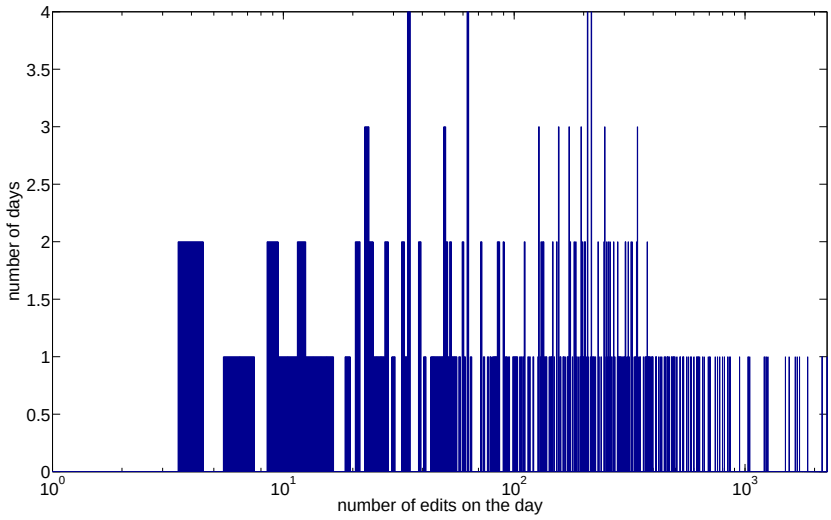
How many edits are there per month?
How many entries are edited per month?

The edits in the elements *cpe-lang:fact-ref*, *cpe-lang:logical-test* and *vuln:configuration* were dropped for these questions since the information is redundantly present in *vuln:vulnerable-software-list* and *vuln:product*. The only effect it would cause is that an edit of a product would show up as two or three edits instead of one, and therefore weigh more than it should. Additionally the days with the top five numbers of edits were removed to filter out extraordinary situations.

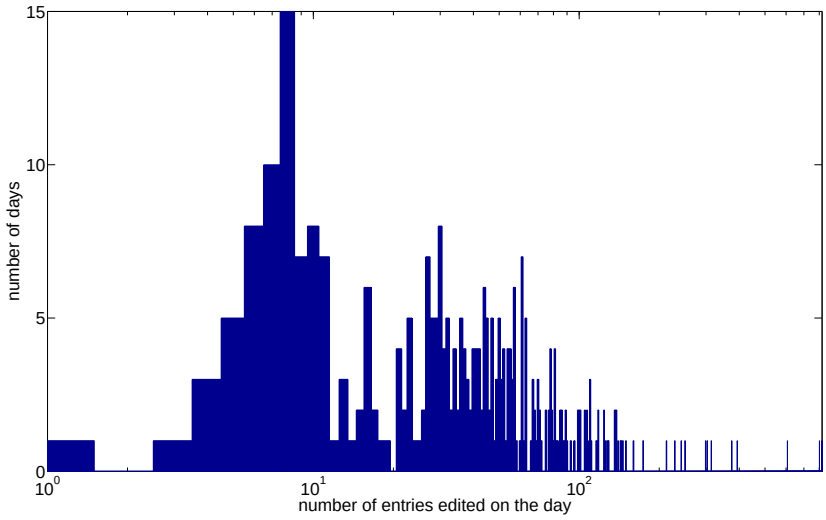
Figure 4.5(a) shows the number of edits per month whereas Figure 4.5(b) shows the number of edited entries over the time the database was downloaded.

Towards the end of the year the number of edits as well as the number of edited entries increases again slightly. This is probably due to finishing work towards the end of the year. This effect continues in January, reaching 1292 edited entries, which might be because vendors released information at the end of the year. During the following months the number of edited entries declines steadily reaching in April with 611 less than half of the peak in January and continues to stay within a range of 550 to 800 edited entries per month.

When looking at the number of edits, March with its many edits seems to be a special case since actually not that many entries were edited. But when looking at both figures together it turns out that August, March and June share the same characteristics, a large number of edits but fewer edited entries as compared to other months with a similar number of edits. This is because

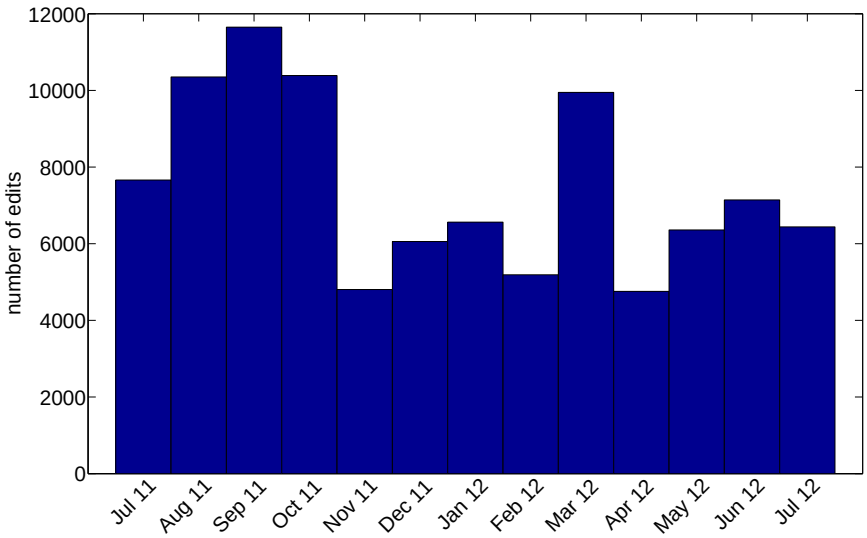


(a) Number of edits per day without the top five edit days

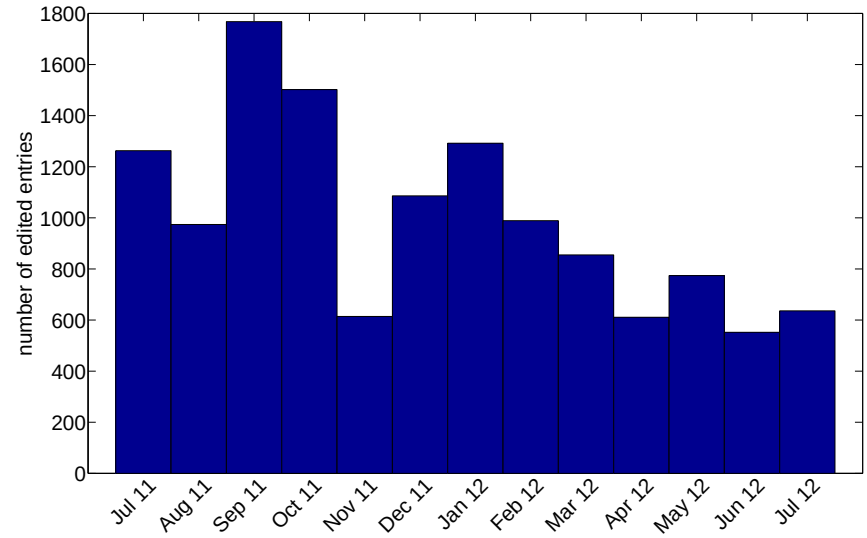


(b) Number of entries edited per day without the top five edit days

Figure 4.4: NVD count of the edit statistics per day



(a) Number of edits per month without the top five edit days



(b) Number of entries edited per month without the top five edit days

Figure 4.5: NVD edit statistics per month

very few, mostly older, entries have a large number of elements edited. This is usually done in the software list. This can be seen more clearly in Figure 4.6.

It shows the number of edits and the number of edited entries split up by the year contained within the CVE identifier. The years range from 1999 (dark blue on the left) to 2012 (dark red on the right). The orange bars correspond to edits in CVE identifiers issued in 2008. In Figure 4.6(a) the number of edits is very large whereas in 4.6(b) the number of edited entries for 2008 is very small. For the CVE entries of the year 2008 this actually is true for the whole period of time and especially in May 2012. On a closer look it turns out that the software list of very few entries is edited very frequently. This means that several products are added and a few days later removed again and added again and so on. Since this persists over a longer period of time it looks more like a script or export error than a real edit.

When looking at Figure 4.6(b) the following observations can be made. Until November 2011 still a lot of entries of older years were edited (colors blue to orange), which looks like that there a backlog had to be processed. As of November 2011 only very few older entries get edited and this is nearly always a change in the software list or references plus an update of the last modified date. Other elements are very rarely edited. This is also true for March and April where again entries of older years are edited.

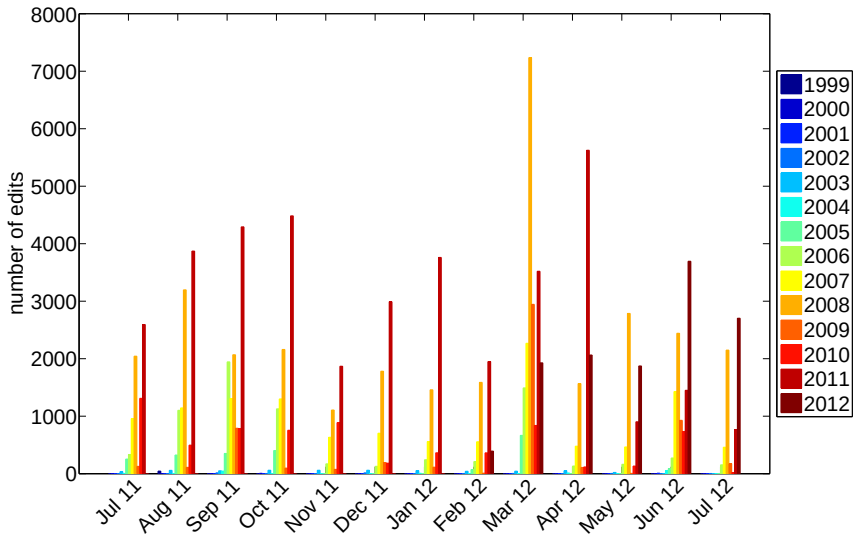
The number of edits in entries with the year 2011 in the identifier (dark red, first from the right 2011 or second from the right since Feb 2012) increases towards the end of the year and January 2012 and then drops sharply. This supports the conclusion that at the end of the year the open work is finished. Since probably also additional information is released by vendors this continues until January.

In February 2012 this drops sharply whereas the number of edits of new entries with the year 2012 increases. This might actually start already in January but the download of the export file for the year 2012 was added on the 29th of February, and therefore the edit history for these identifiers is missing until then.

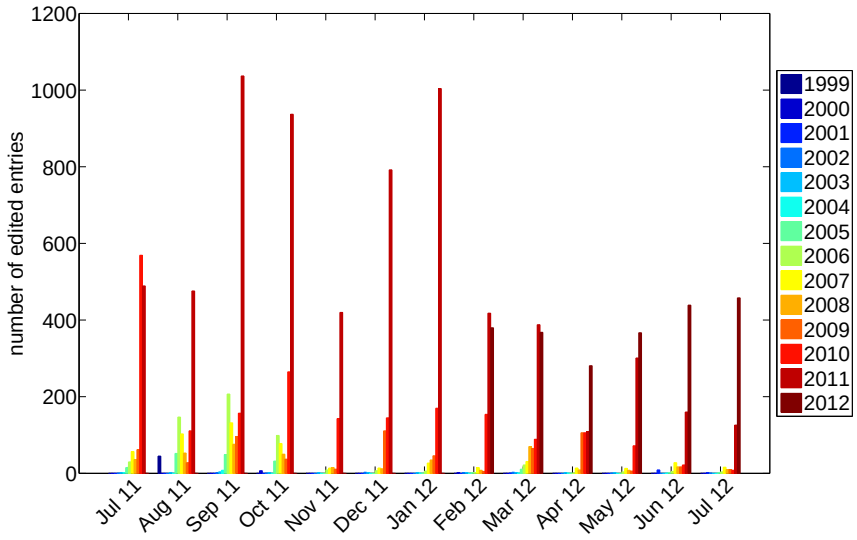
Are there certain days in the month with increased activity? How many entries are edited per week day?

For these questions we only look at the edited entries because the number of edits does not give additional information except that sometimes there are a few entries where many edits are performed at once.

Figure 4.7(a) shows the number of edited entries per day of the month (1st to 31st). Each day of the month represents the sum of the edited entries of the corresponding days in the month across all the months. It peaks on the 5th, 7th, 19th, 26th and 27th. The peak on the 26th exists partially due to a download error which caused the changes of several dates to be accumulated on that date. The 5th, 7th and 19th exist primarily because there is one day with edits in a vast number of entries. On the 27th also one such day exists but there are also frequently days with larger edits. Keeping all this in mind the monthly editing pattern is quite consistent with about 500 to 700 edited

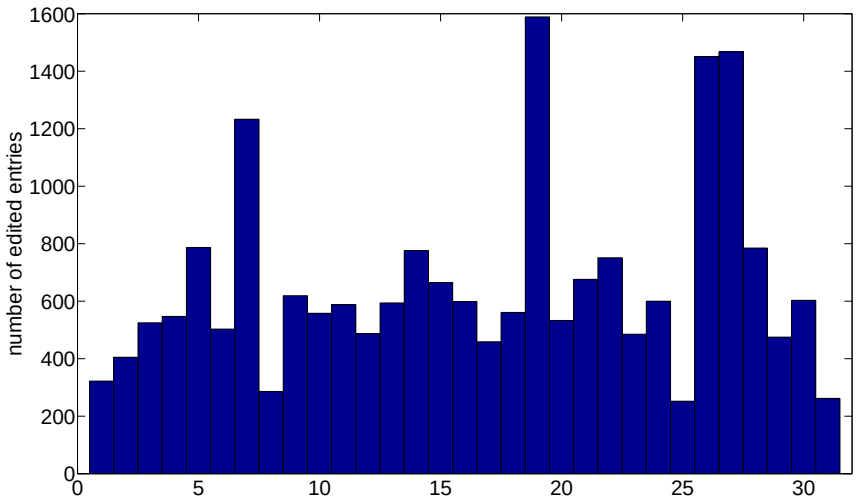


(a) Number of edits per month without the top five edit days

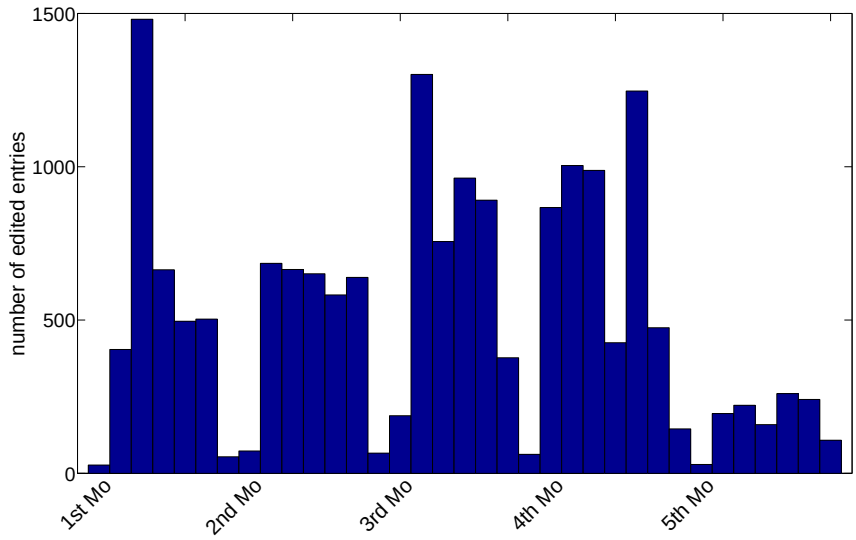


(b) Number of entries edited per month without the top five edit days

Figure 4.6: NVD edit statistics per month for each year separately



(a) Number of edited entries per day of the month without the top five edit days



(b) Number of edited entries per week day of the month without the top five edit days

Figure 4.7: NVD edit statistics within a month

entries per day over the last year. There are less edits at the first few days of the month, small concentrations around the 14th and 22nd, and there is also a slight increase towards the end of the month.

Figure 4.7(b) shows the edits within a month by week days starting with the first Monday on the left to the first Tuesday, first Wednesday and so on to the fifth Sunday on the right. The first Monday does not have to be the first day in the month since it can also start with any other week day and is, therefore, simply the first Monday encountered per month. The 5th week consists of the week days that fall on dates beyond the 28th of the month and for that reason are less common.

There is a clear weekly working pattern with 5 days of increased activity and two days of reduced activity. At first it looks like this reduced activity happens on Sunday and Monday but since the data is exported very early in the morning of the day this means that the edits are shifted by one day and the Monday actually represents the changes of the Sunday. When looking at the Mondays, representing the Sundays, the 4th Monday is the odd one out, which would imply that on the 4th Sunday there is increased activity but the 26th of September on which the accumulated data was checked in after the error falls on the 4th Monday and accounts for the majority of the edited entries.

The extreme peaks on the 1st Wednesday, 3rd Tuesday and 4th Friday are largely due to a large number of edits on one day. When cutting these out of the picture an increase of edits towards the end of the month can be observed. The edits made in the 5th week also partially represent edits made in the 4th week depending on how the dates lie which supports the observation. Also when looking at the 4th and 5th Sunday which together represent the 4th and 5th Saturday in the month have increased activity.

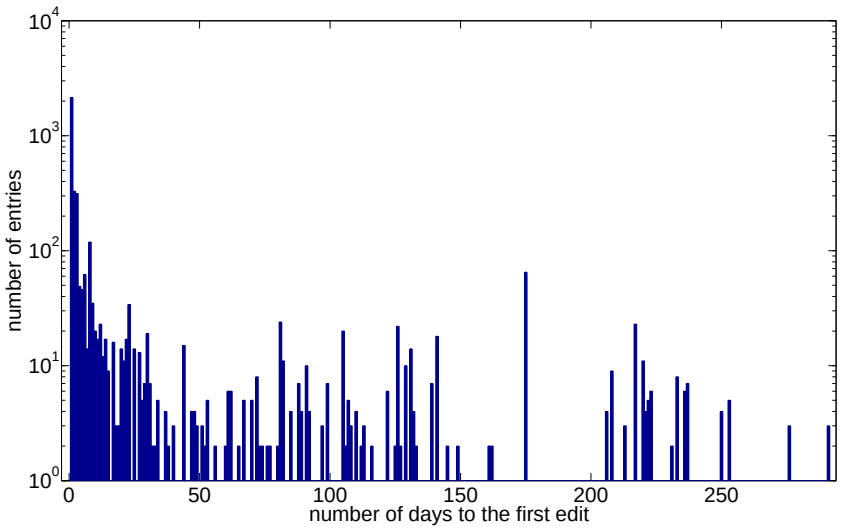
After how many days was the entry first edited?
After how many days was the entry last edited?

These are very interesting questions since the answers tell us how reliable the published information is regarding edits after a certain period of time. For entries, for which the original add date is unknown, no precise duration can be calculated, and therefore they are excluded from this analysis. This means edit durations are only calculated for entries for which the entry add was observed.

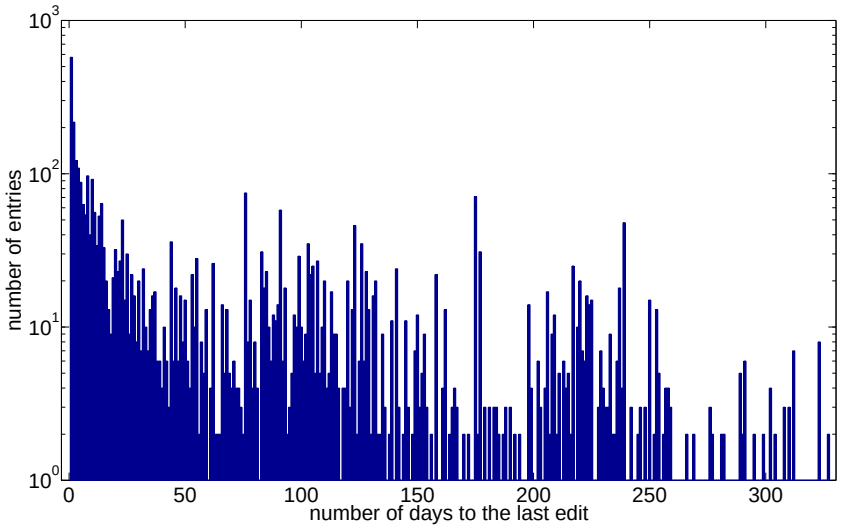
These are very interesting questions since the answers tell us how reliable the published information is regarding edits after a certain period of time. Entries for which the original entry add date is unknown no precise duration can be calculated and are, therefore, excluded. This means the edit durations are only calculated for entries for which the entry add was observed.

In Figure 4.8(a) each bar represents the number of entries that were edited the first time after the number of days equal to its position on the x-axis. The second Figure 4.8(b) shows the same counts for the last time the entry was edited. Their y-axis use a logarithmic scale. The entries that do not experience a change at all are excluded of this analysis. They make up 17.8% of the entries in data set and the remaining 82.2% experience an edit in the observed time.

A total of 89% of the edited entries have their first edit (Figure 4.8(a)) within



(a) Number of entries counted by the duration until the first edit



(b) Number of entries counted by the duration until the last edit

Figure 4.8: NVD count of entries to the first and last edit

30 days and 96% within the first 150 days. Only 0.2% of the entries experience the first edit later than 250 days after the entry was edited. For the most recent edit the picture changes (Figure 4.8(b)), but still 52.8% of the entries were edited only in the first 30 days and 85.1% within the first 150 days. Barely 1.8% of the edited entries were edited the last time after more than 250 days after the entry was added.

Most of these edits are not really substantial because they are mostly related to either the software list or the references. When *cpe-lang:fact-ref*, *cpe-lang:logical-test*, *vuln:configuration*, *vuln:vulnerable-software-list*, *vuln:product*, *vuln:scanner*, *vuln:references*, *vuln:last-modified-datetime*, *vuln:assessment_check* and *vuln:security-protection* are excluded the results change slightly. Over 41.4% experience no substantial changes at all which means that nearly 58.6% of the entries are edited.

The majority of the entries have their first substantial edit (4.9(a)) within 30 days after the entry was added. This group accounts for 99.3% percent of the entries. In less than 1% of the entries these elements are edited the first time later than 60 days after the entry as added.

The last substantial edit (4.9(b)) is for still over 98.4% conducted within the first 30 days. In still less than 1% of the entries experience an edit in substantial elements after more than 60 days.

Regarding substantial elements the entries are also much less likely to be edited multiple times as can be seen in figure 4.9(c).

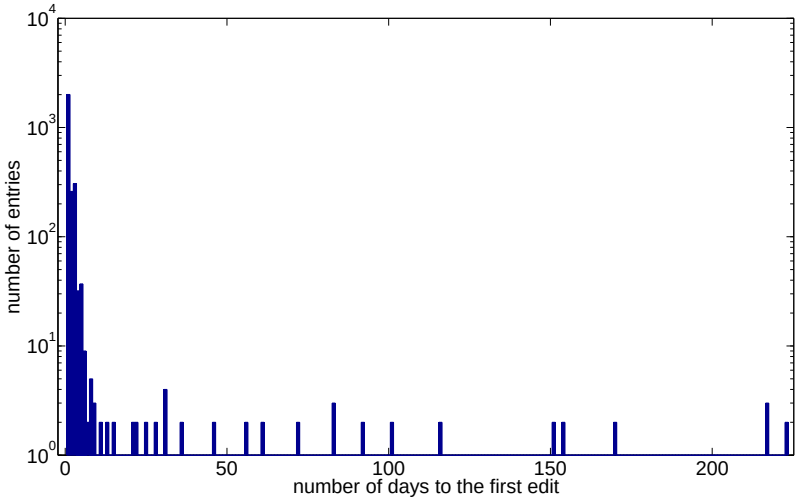
When are new entries added?

For this question only the newly added entries were considered. Figure 4.10(a) shows the number of newly added entries per month. The number in January 2012 is probably too low whereas February is too high. This is because the new file for the year 2012 was added on the 28th of February, which caused that all new entries, with CVE identifiers issued previously in 2012, show up the first time. In total 380 entries were added on that day. Some of these entries were probably added in January. If January some of the new entries of February are assigned to February, then there is a continuous groth of new entries per month, and only September, October and April do not fit in.

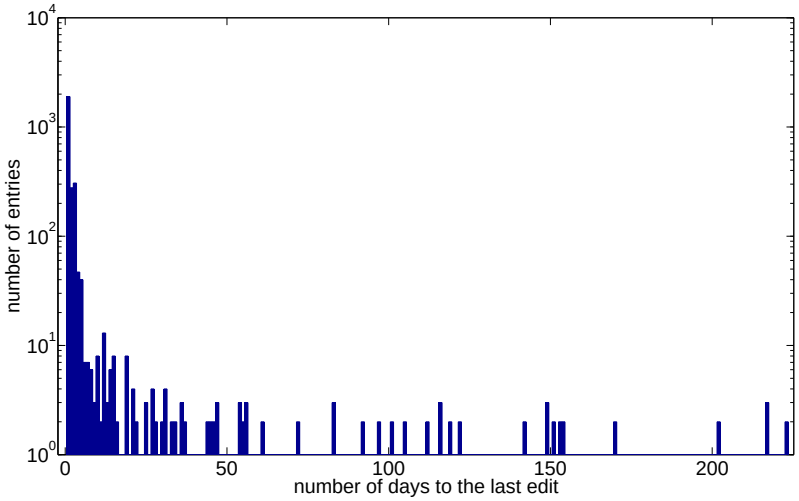
Looking at Figure 4.10(b) the number of newly added entries over the month does not paint a clear picture. It is important to note that the new files are released in the early hours of the day (local time), and therefore reflect the changes of the day before. So if many of changes show up on the 3rd or on a Tuesday this means the edits happened on 2nd or on Monday.

At first the 28th seems to be very significant but is actually not because then the 2012 file was initially added. This leaves the 26th, 19th, 9th, 3rd, 4th and 21st the most outstanding days in the month. The 30th and 31st can be merged into one day because together they represent the last day of the month. On the other hand the 1st, 7th, 16th and 24th have a very low publishing rate.

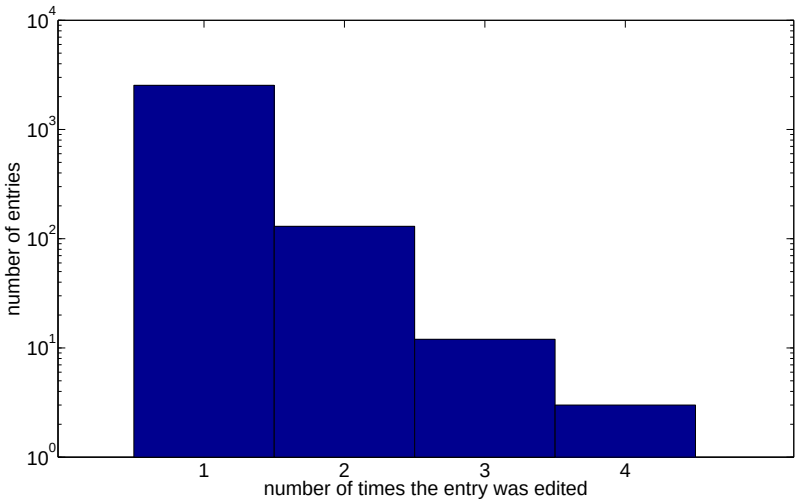
This gives us three main publishing periods in a month. First at the beginning of the month which are probably new entries based on information released by vendors at the end of the month. The easy to complete ones can be published



(a) Number of entries counted by the duration until the first edit

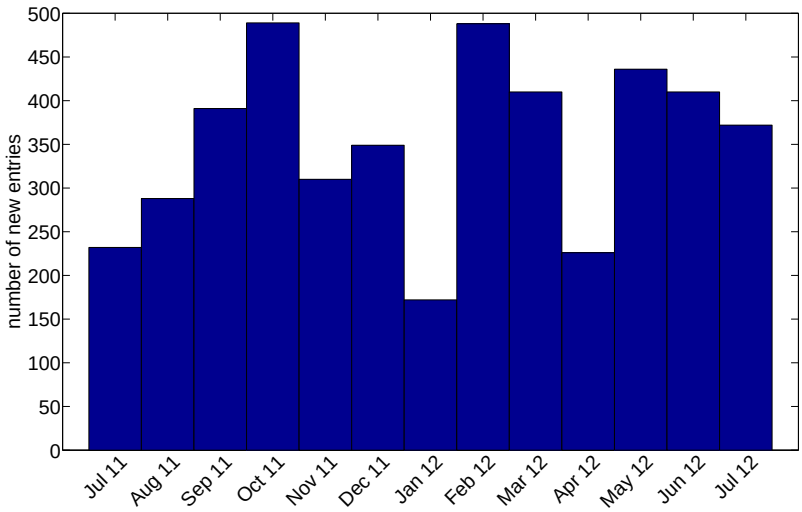


(b) Number of entries counted by the duration until the last edit

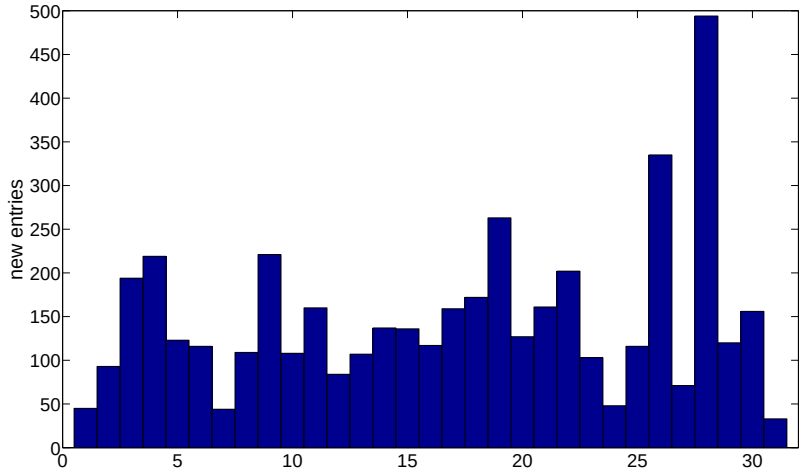


(c) Count of the times the entry was edited

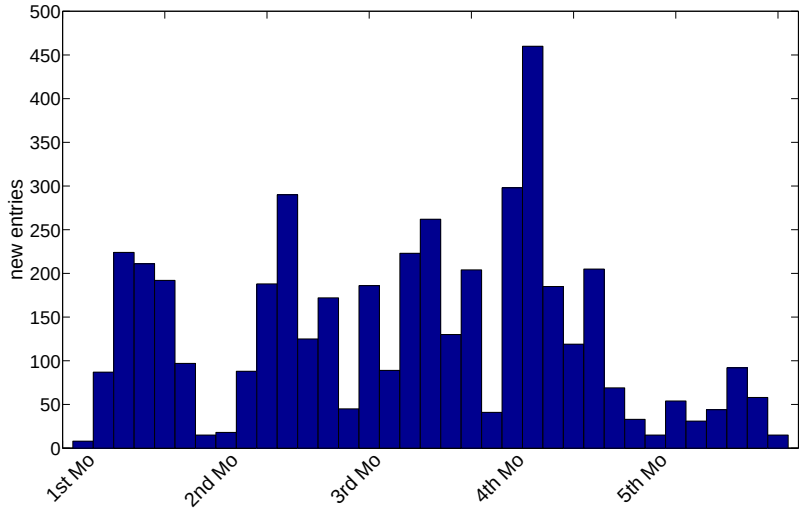
Figure 4.9: NVD count of entries to the first and last substantial edit



(a) New entries per month



(b) New entries per day in the month



(c) New entries per week day of the month

Figure 4.10: NVD new added entries

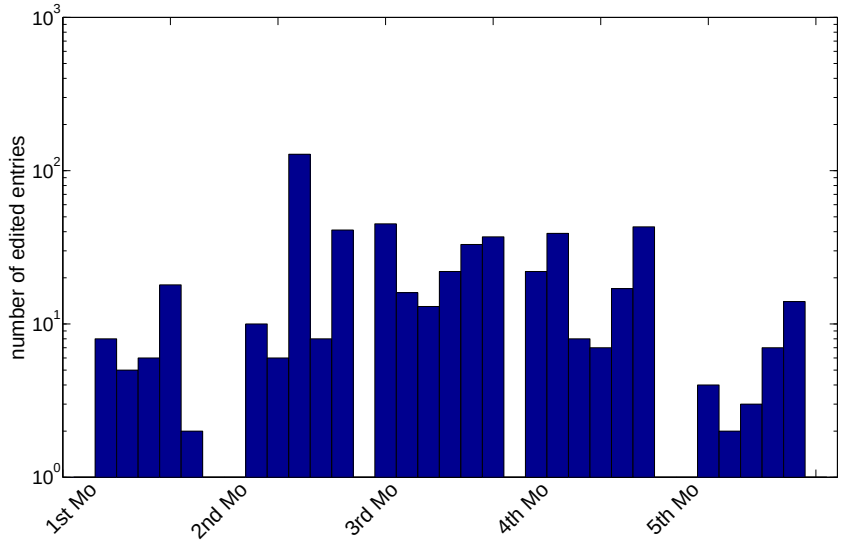


Figure 4.11: NVD edits on entries related to Microsoft products

quickly whereas others might require more work and are published a bit later. The second publishing date is around the 20th of the month which is probably when the newly discovered vulnerabilities of the month are published. Towards the end of the month the open work should be finished, and therefore more entries are published.

Figure 4.10(c) shows how new entries are published over the week days of the month. The largest bar on the 4th Tuesday in the month is also due to the adding of the 2012 year file because the 28th of February was on the 4th Tuesday in February. Also the 4th Monday in the month is not that significant because the 26th of September falls on the 4th Monday of the month where the accumulated data after a download problem was checked in but there are also other significant days recognizable.

First The 2nd Wednesday which reflects the changes of the 2nd Tuesday in the month which is the Microsoft patch day. Second the 4th and 5th Friday together represent the 4th, 5th and rarely 3rd Thursday of the month which means which also supports that towards the end of the month the open work should be finished.

Is it possible to see the Microsoft patch day?

As already suggested in the previous section about newly added entries the Microsoft patch day (second Tuesday of the month) is visible. The number of edited entries which are related to Microsoft products can be seen in figure 4.11. There is more activity after the 2nd Tuesday of the moth with continued work in the following two weeks. The 2nd Thursday is very significant since it marks the edits made on the 2nd Wednesday because it does not fit into the general pattern of finishing work towards the end of the week and month. There are in

general no changes on weekends except for the 3rd, 4th and rarely 2nd Sunday which are represented by the bars on the 3rd and 4th Monday.

How often are severities reassessed?

This question is particularly interesting because an edit in the CVSS part of the entry means that some information concerning the vulnerability changed and not some typo was corrected, more vulnerable software was found or that there are new references.

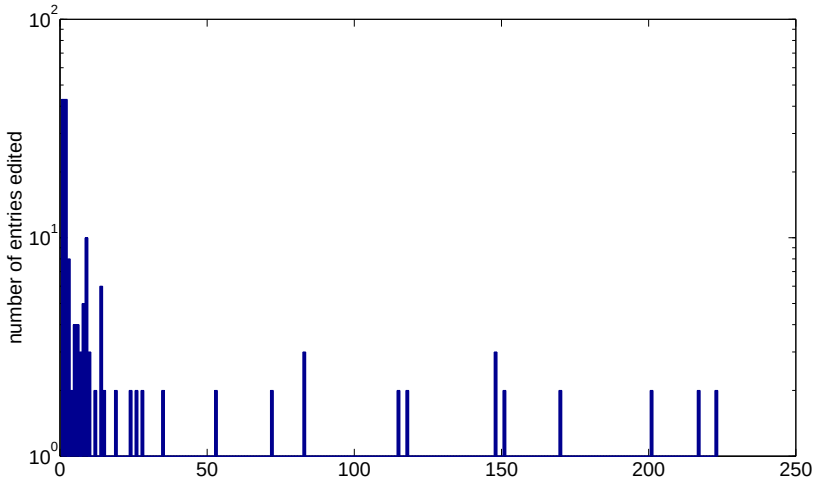
Over the observed period 3221 edits that are some related to the CVSS information were documented and 2734 of entries were edited. In the CVSS information the following three types of edits can happen.

- The whole CVSS section was previously missing and is now added. This normally happens when a new entry is releases an the CVSS information is added later. This is the by far the most common type of edit with a total of 2594 edits.
- The whole CVSS section is removed. This only happens when the entry is deprecated because all fields get deleted and is very rare and only four were observed.
- The information of one of the parameters within the CVSS is edited (e.g., access vector, confidentiality impact, availability impact, etc.). A change of any of these parameters also changes the CVSS score since it is derived by a formula based on the values of the parameters. This type of edit is uncommon with a total of 214 edits in the observed time.

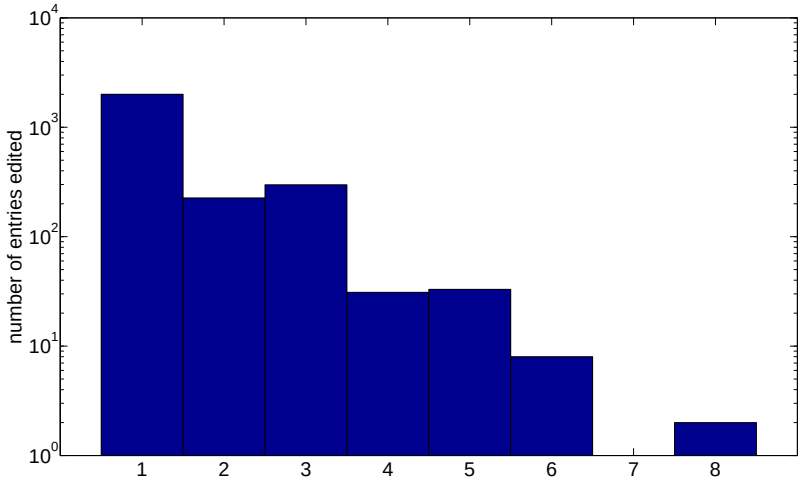
The changes within the CVSS parameters are probably the most interesting. These are normally only edited once except for two entries which were edited twice. The second edit was the day directly after the first.

The next interesting question is after how many days was this information changed. There was also information edited in entries for which we do not know the initial publication date. These entries were excluded from the analysis since we cannot calculate the precise edit duration and any approximation is prone to large errors. This is because even if the CVE identifiers was issued in a certain year, it does not mean that the NVD entry was published in that year since entries to reserved identifiers might show up years later in the NVD. Figure 4.12(a) shows how many entries were edited the first time after how many days. The figure has a logarithmic y-axis. Most of the changes happen within the first month after the entry was published. Edits much later than a month happen very seldom.

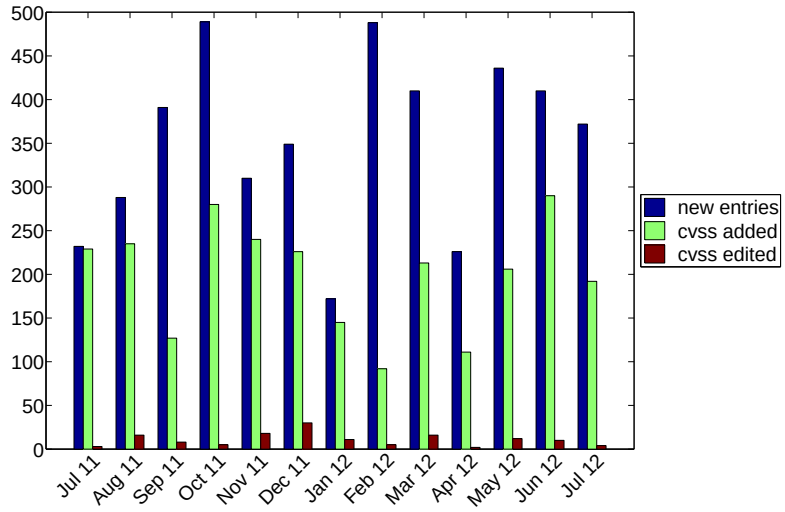
When a whole CVSS section is added later to the entry it is important to know how much later this happens. If it happens very much later it raises the question why the entry has been already published but if it is just a few days after it should not be of major concern. Figure 4.12(b) shows the delay distribution of entries, for which the CVSS section was added later. The figure has a logarithmic y-axis. For the majority of the entries the missing CVSS section is added the next day or within three days after the entry was added.



(a) Number of days until the first CVSS data edit



(b) Number of days until the CVSS data is added



(c) Comparison between new entries, delayed cvss data and edited cvss data

Figure 4.12: NVD edit statistics to CVSS related information

The remaining few are added within at most eight days. This means that the missing data is added pretty quickly and one can expect that latest a week after the entry was published in the NVD the CVSS information is present.

In the last part we wanted to know how many of the entries the CVSS section was added or edited later on. The data is summarized for each month. As reference date the date of the initial add of the entry was also used for the CVSS add and the CVSS parameter edit. Therefore, the edits are also assigned to the same month as the entry add. The result can be seen in figure 4.12(c). The number of edited entries in relation to the total added entries is relatively small with a range from less than 1% up to 8.5%. The number of entries for which the CVSS section was published delayed is much larger ranging from 18% to 98%.

Are there any edit wars?

Edit wars are characterized by editing the same information from one value to another one and back multiple times. This means that the entry is edited on many days with the same number of edits per day. On each day the same elements are edited and the performed changes in each element alternate between edits. This analysis was performed manually. Possible candidates were selected by the number of days the entry was edited. The following entries were checked:

- CVE-2003-0497 and CVE-2003-0498 were each edited on 264 days. This was very likely an error because only the last modified date field was updated until on 12th of May 2012 on each one reference was added. Since then these entries were not edited anymore.

- CVE-2007-0527 was edited on 186 days. This is also very likely to be an error since every few days several products are removed and a few days later added again. The vulnerable configuration list which holds the same information as the vulnerable software list is not edited.

- CVE-2007-6538 was edited on 160 days. This is probably an error. In this entry three products are added, removed and added over and over again. It is done in the software and the configuration list.

- CVE-2007-5131 was edited on 154 days. This is probably an error. In this entry two products are added, removed and added over and over again. It is done in the software and the configuration list.

- CVE-2007-3687 was edited on 153 days. This is probably an error. In this entry one product is added and removed over and over again. It is done in the software and the configuration list.

- CVE-2007-3063 was edited on 153 days. This is probably an error. In this entry one product is added and removed over and over again. It is done in the software and the configuration list.

- CVE-2007-4984 was edited on 150 days. This is probably an error. In this entry one product is added and removed over and over again. It is done in the software and the configuration list.

Some edits theoretically could be edit wars because there is no information that would contradict it. But the edits made to the entries are done very frequently and the dates on which they are performed often match for multiple entries. Also the normally very little elements are edited mostly in the vulnerable configuration list and the vulnerable software list which contain normally quite number of vulnerable product. It does not make much sense that one would argue so persistently over a few versions of an application in the vulnerable software list where as other versions of the same application stay in the list. It is ,therefore, hard to imagine that these would be edit wars and far more likely to be errors caused by scripts.

Does the sum of edited entries per day correspond to the number of edits of *vuln:last-modified-datetime*?

This question rose because it was observed that sometimes just the *vuln:last-modified-datetime* was edited as well as sometimes elements were edited and the last modified date was not updated.

In total 17045 unique entry edits (i.e., one edit per entry per day) were registered but only 14165 *vuln:last-modified-datetime* updates which means in 2880 cases the last modified date was not updated.

The majority of the edited elements are the *vuln:product*. Most of these edits are caused by very few entries that frequently add and remove elements to the software list. This is done over a longer period of time which suggests that there is either a scripting or an export error.

The element with the second largest number of undated edits is *vuln:scanner* with a total of 760 edits. All these elements were added to older entries which was probably an update of older entries.

The remaining few edits fall on *vuln:references* (23 add and 23 removes), *vuln:summary* (18 changes), *entry*(3 removes), one add of a *vuln:vulnerable-software-list* and one change of a *cvss:integrity-impact* which also causes a *cvss:score* update. These all either look like a fix of errors (e.g., references, summary) or are errors themselves (e.g., the three removed entries).

The short answer to the question is no, the number of unique entry edits do not correspond to the number of last modified date updates. The sources of undated edits all seem to origin in some kind of error that either causes the edit or the previously mistakes is corrected. Another explanation could be that the edited information originates from a different database. Therefore, an update of information would not trigger an update of the last modified date in the NVD.

4.2.2 OSVDB Analysis

In Section 4.1.3 some very general information about the edits of the open source vulnerability database was given. In the following the OSVDB history will be analyzed in more detail. These results are used to verify the results

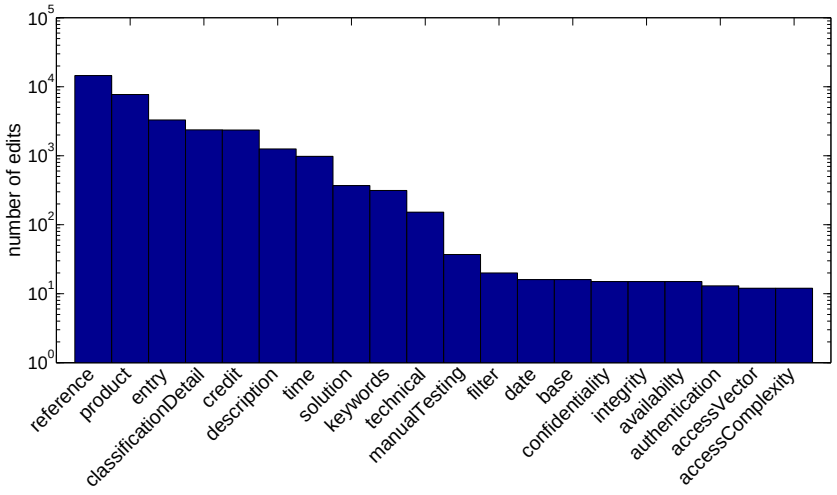


Figure 4.13: Tag edits in the OSVDB

obtained about the NVD, therefore the OSVDB will not be analyzed in the same detail which limits the questions to more general ones.

Which data (XML element) was edited?

Figure 4.13 shows the number of edits per tag sorted in descending order. The y-axis uses a logarithmic scale.

Most of the edits happen in the element *reference* which holds the reference to an external source. This is not surprising since other vulnerability databases also create an entry for the vulnerability and a reference is added. For entries referencing to CVE entries the update of the references can be scripted to update the *references* with the references associated with the CVE entry.

A lot of edits can also be found in the element *product* which holds the product information of vulnerable applications. Since many applications can be vulnerable to the same vulnerability and not all might be known when the entry is published first, therefore it is not surprising to see many edits within this element.

Many edits are found in the element *entry* which hold the information of on whole entry. This is because every time an entry is seen for the first time it is logged as such. It can also happen that an entry gets removed. During the four months of observation this happened 14 times. A remove also happened to another 131 entries but they were readded again. But this might also be caused by a download problem which results in a missing entry that is present again the following day. Interestingly enough to several entries this effect occurred more than once. This effect appears less since the download tool retries the download of every entry for which it failed.

The information within the tag *classificationDetail* contains information about different fields in the classification section on the website. These try to classify

the vulnerability in a similar way to the CVSS parameters in the NVD. In over 86 % of the edits new information is added, such as an impact description, who disclosed it or that there are solutions available. The remaining edits fall mostly to changes which update the given information.

The *credit* elements hold the information who claims the credit for the discovery of the vulnerability. A *credit* can be just a name or an internal link to the credit history of that person. In general it is not surprising to see changes within this element since at first it might be not clear who discovered the vulnerability. This element is also suspect to the *attribute-edit* effect which causes changes to appear as a remove of the old element and an add of the new element. The changes are mostly about fixing typos in names or converting the name to the internal link.

The *description* element holds the description of the vulnerability similar to the *vuln:summary* in the NVD. The OSVDB seems to import data from the CVE list where also reserved entries are published. After the detailed information is released the *description* element has to be updated which causes the edits.

The *time* element holds one specific time stamp of the *timeline* element. This timeline can have different time stamps such as the disclosure date, vendor informed date, vendor solution date or the exploit publish date. This list is updated over time. In most cases these are minor corrections in the disclosure date but also new fields are added. Therefore, changes have to be expected over time.

The *solution* element describes the possible solutions to resolve the vulnerability. This can either be that no solution is known to the OSVDB, the application should be patched or upgraded or a workaround is described. Changes to this element are also to be expected, especially if the entry is published early.

The *keywords* element holds a series of strings that summarize the vulnerability in related keywords. These are identifiers for the vulnerability such as a CVE identifier, the port used, the exploit string or a related product. The more information about a vulnerability is gained the more keywords might have to be updated.

The *technical* element holds the information of the non-mandatory field *Technical* of the website. It gives more technical details related to the vulnerability which are not present in the *description*. These are details which are in general added later.

The *manualTesting* element contains some exploit code or a description how the system or an application can be tested for the vulnerability. This field is not very common, and therefore the edits to this element are very rare, only 36 over four months.

The *filter* element contains a reference to a tool, vulnerability scanner or filter which should be able to detect the vulnerability (e.g., snort, nessus). Edits are very rarely to happen which is probably because either the information is present when the entry is published or the information has to be obtained manually.

The remaining elements *date*, *base*, *confidentiality*, *integritiy*, *availabilty*, *authentication*, *accessVector* and *accessComplexity* are all part of the *cvss* element.

These elements describe the CVSS rating of the vulnerability. This information is generally imported from the NVD and updated only in very few cases.

How many edits are there per entry?
On how many days was the entry edited?

Each entry will appear at least once when it shows up the first time.

The OSVDB identifiers are created continuously which means when the identifiers are sorted, then they are also sorted chronologically. Therefore, when plotted, the oldest entry will be on the left side whereas the newest is on the right side. Unlike the CVE identifiers the OSVDB identifiers do not give the year in which they were issued.

The expectation that newer entries experience more edits than older ones was fulfilled. This can be seen in figure 4.14(a) showing the number of edits per entry on a y logarithmic scale. The few peaking entries which experience more than 50 edits are all mainly due to edits in the products list. The majority of the entries experience very few edits. Less than 0.2 % of the entries experienced more than 20 edits in the last four months.

This effect is even more visible when the numbers of days the entry was edited on are plotted (figure 4.14(b)). Old entries are rarely edited more than three times whereas new entries are edited more often. There are also no entries which are edited significantly more often than others.

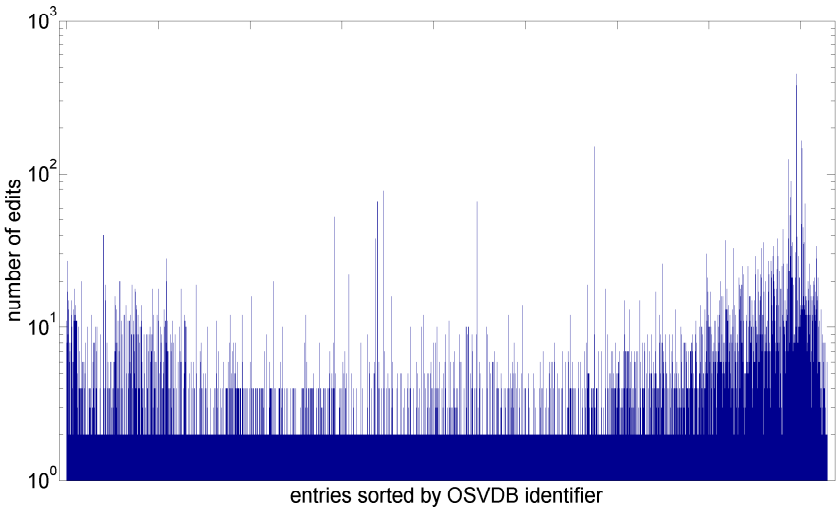
How many edits are there per day?
How many entries are edited per day?

The figure 4.15(a) shows the number of edits per day over the time the database was downloaded. There are a few times when there are no edits present. On these occasions the download failed or had to be stopped. This was sometimes necessary after a new version of the download scripts were rolled out or the infrastructure was temporarily not available (e.g., emergency power shut down test).

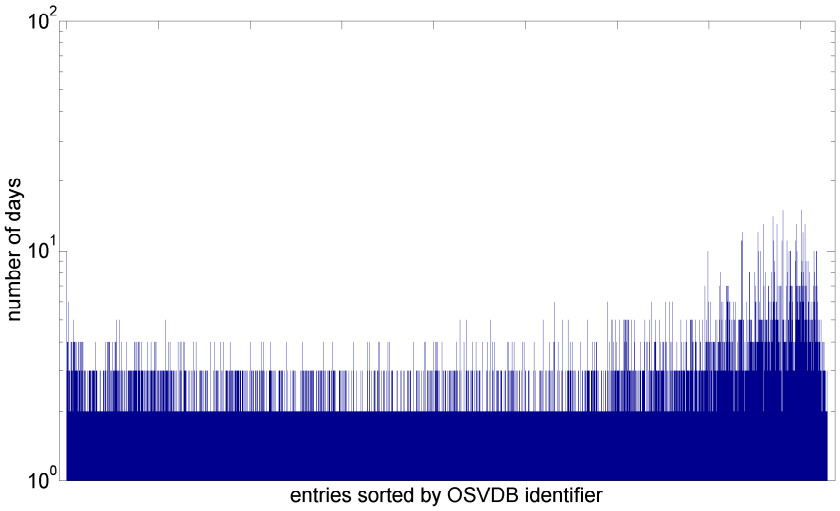
Normally the number of edits per day are in a range between 40 and 400 edits. There are very few days that stand out regarding the total number of edits. Two of the dates follow a download gap (2012-06-29 and 2012-05-10) and the number of edits is likely due to the fact that the edits of the missing days were aggregated on these dates. On the 11th of June 1'058 credits were updated causing the large number of edits. On the 24th of April in several vulnerabilities Samba was added with many vulnerable versions as well as many references causing the majority of the edits.

On the 15th of May a minimum of 36 edits is reached and the number of edits continuously grows until the end of June and declines again a bit in July.

When looking at the number of edited entries per day (figure 4.15(b) using a logarithmic y scale) it is very similar to the edits per day 4.15(a). This is because most of the edits experience very few changes which caused bars to scale in the same way.

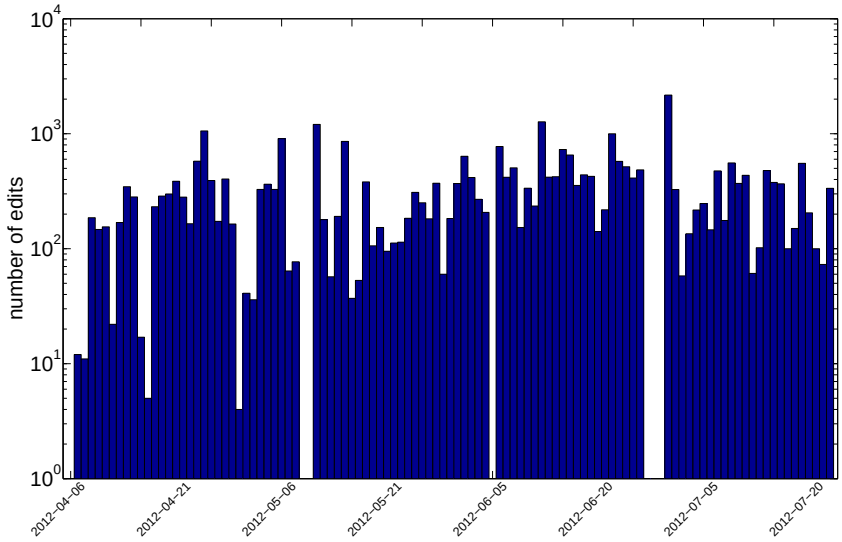


(a) Number of edits by entry

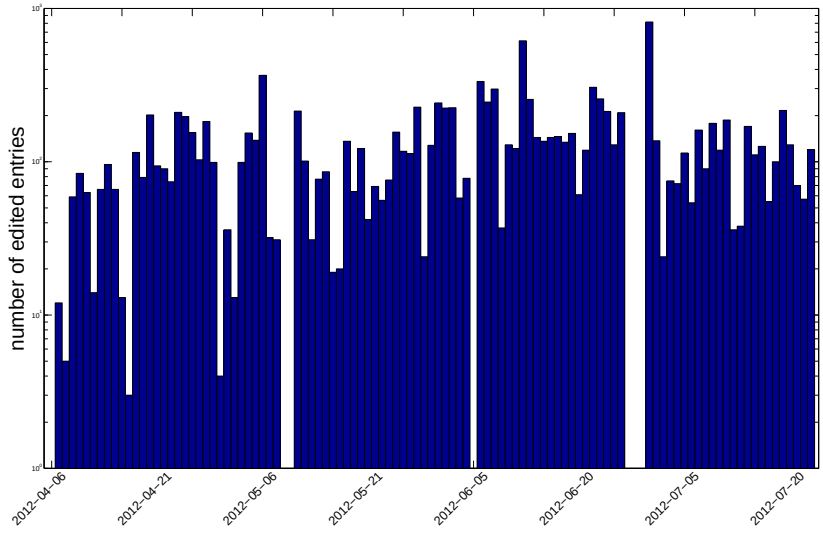


(b) Number of days the entry was edited

Figure 4.14: OSVDB edit statistics by OSVDB identifier

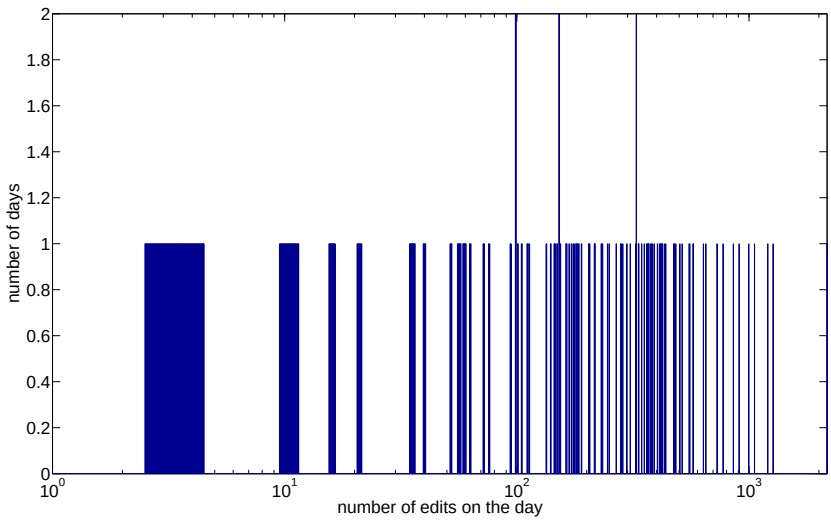


(a) Number of edits per day

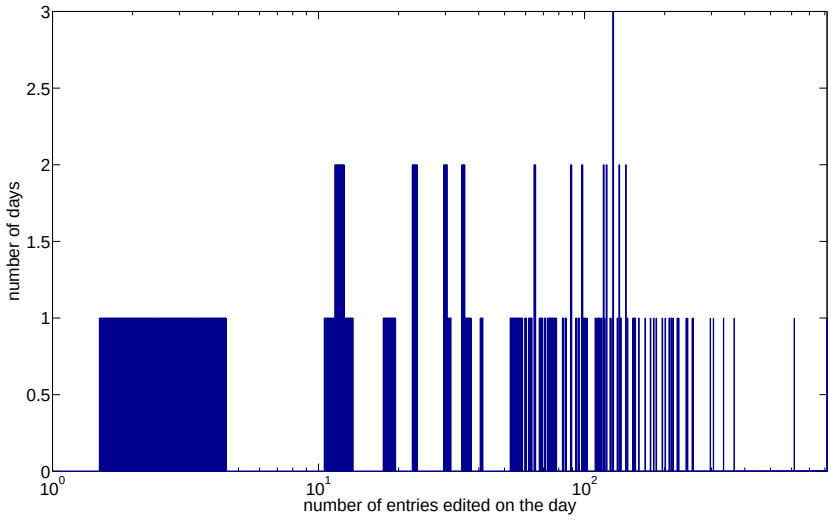


(b) Number of entries edited per day

Figure 4.15: OSVDB edit statistics per day



(a) Number of edits per day



(b) Number of entries edited

Figure 4.16: OSVDB count of the edit statistics per day

Figure 4.16(a) shows the counts of the values of figure 4.15(a) with a logarithmic x-axis. If there is a bar of height 3 at position 10 this means that there are 3 days with 10 edits. This graph supports the previous statement that the majority of the days have between 40 and 400 edits. But there is no significant grouping of edits per day.

The picture is slightly different when looking at the count of figure 4.15(b) which can be seen in figure 4.16(b) with a logarithmic x-axis. If there is a bar of height five at position seven it means that there are five days where seven entries were edited. Here a grouping between 50 and 300 edited entries per day starts to be visible which might get stronger with more data.

How many edits are there per month?
How many entries are edited per month?

The figure 4.17(a) shows the number of edits per month over the time the database was downloaded. The total number of edits ranges between 5500 and 8200 but in June jumps to nearly 14000 edits. When the data for July will be complete, that monthwill actually have more edits but the sample stops on the 24th of July. In June the number of edits is in general very high as can be seen in figure 4.15(a). It seems like in June many entries were updated with additional information because there are significantly more edits in references and credits in May and July which cause this significant difference compared to the other months.

As already mentioned when looking at the days, the number of edits per entry is generally low. Therefore, the number of edits follows generally the same curve as the number of edited entries with a certain factor. For these months the factor ranges from 2.68 in June to 3.9 in April.

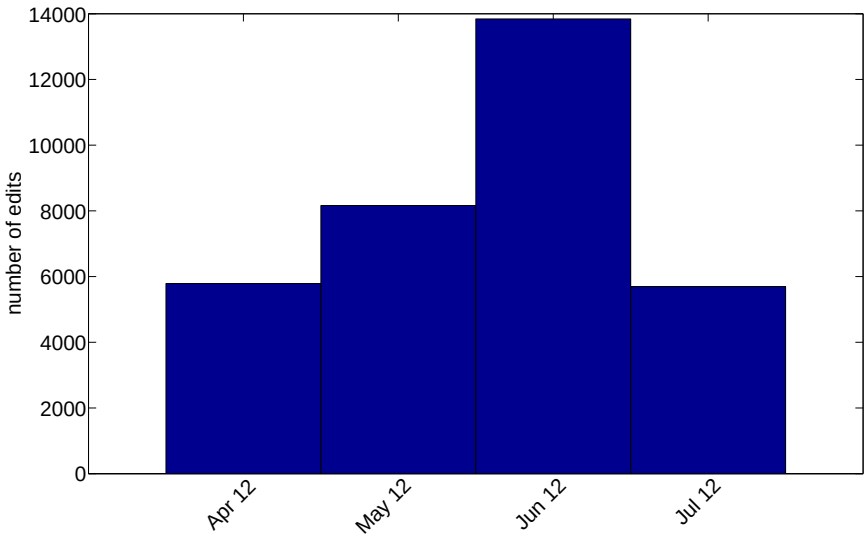
Are there certain days in the month with increased activity?
How many entries are edited per week day?

For these questions only the edited entries were considered because the number of edits does not give any relevant additional information besides that, rarely there are a few entries where many entries are edited at once.

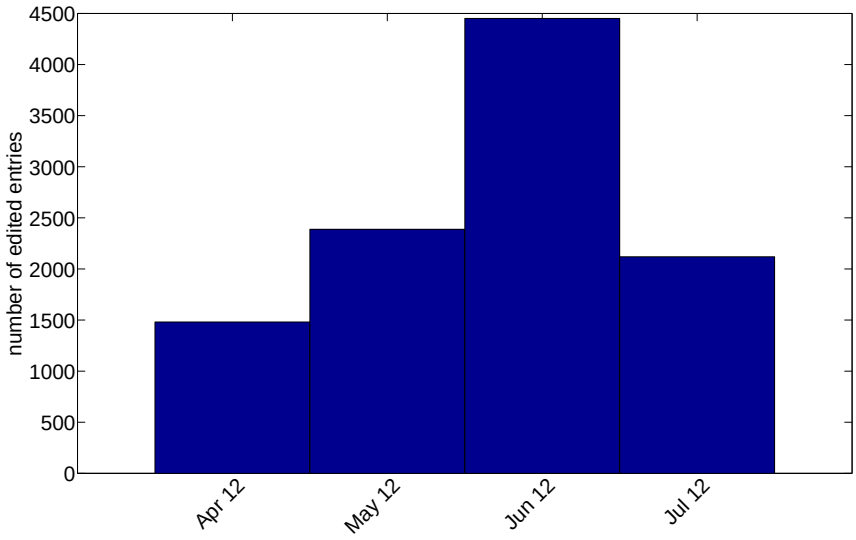
Figure 4.18(a) shows the number of edited entries per day in the month (1st to 31st). The month days from 1 to 5 and 23 to 31 are also represented less since the data was collected starting 6th of April until 22nd of July.

There are peaks on the 5th, 10th, 11th and 29th. The increased number of edits on the 10th and 29th can be explained by the missing downloads on the preceding dates which caused the edits to be accumulated on the 10th of May and the 29th of June.

It would be too early to say that on the 5th there is significantly increased activity since it only happened twice in May and June but not in July. On these dates references in many entries were updated. The edits on the 11th mainly result from the credits update on the 11th of June when the credits in many entries were updated.

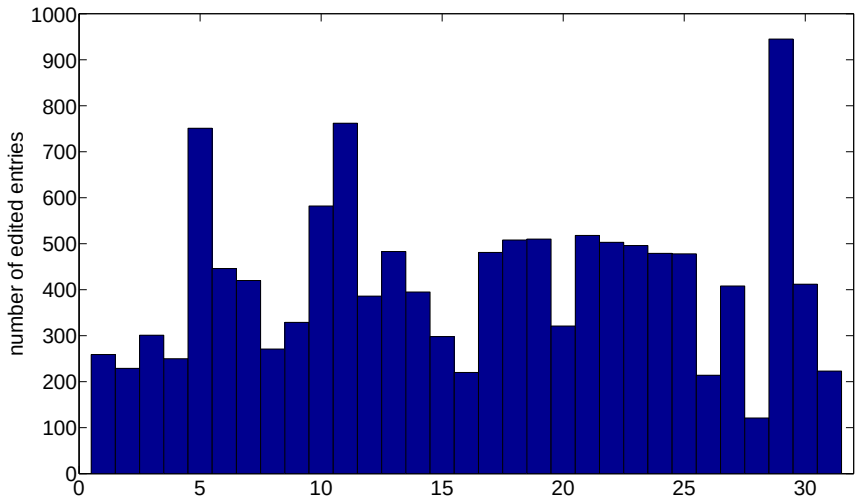


(a) Number of edits per month

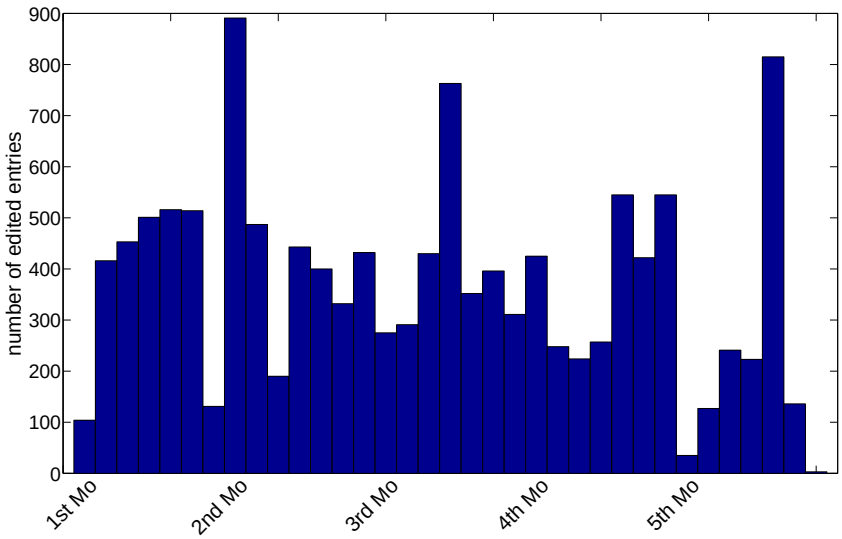


(b) Number of entries edited per month

Figure 4.17: OSVDB edit statistics per month



(a) Number of edited entries per day of the month



(b) Number of edited entries per week day of the month

Figure 4.18: OSVDB edit statistics within a month

When taking these peaks out of the picture, fewer entries are edited at the beginning of the month which increases towards the 20th and then declines a bit towards the end of the month. The increased activity around the 6th and 7th is due to larger changes conducted on the 6th and 7th of June when mainly references and keywords were added but also a larger number of new entries was added.

The second figure (4.18(b)) shows the edits within a month by week days starting with the first Monday on the left to the first Tuesday, first Wednesday and so on to the fifth Sunday on the right. The first Monday does not have to be the first day in the month since it can also start with any other week day. It is simply the first Monday encountered in each month. The 5th week consists of the week days that fall on dates beyond the 28th of the month and for that reason are less common.

No clear weekly working pattern can be seen except for the first and the fifth week when not many entries are edited on Mondays and on Sundays.

The peak on the second Monday is because of the previously mentioned credit update on the 11th of June. The 29th of June with the accumulated edits after a download error falls on the fifth Friday of the month.

On the third Thursdays of the month there is frequently more activity. During the first week of the month also more activity can be observed which even increases towards the end of the week. At the end of the fourth and also during the fifth week the number of edited entries increases. This is especially significant for the 5th week because these days are not present every month.

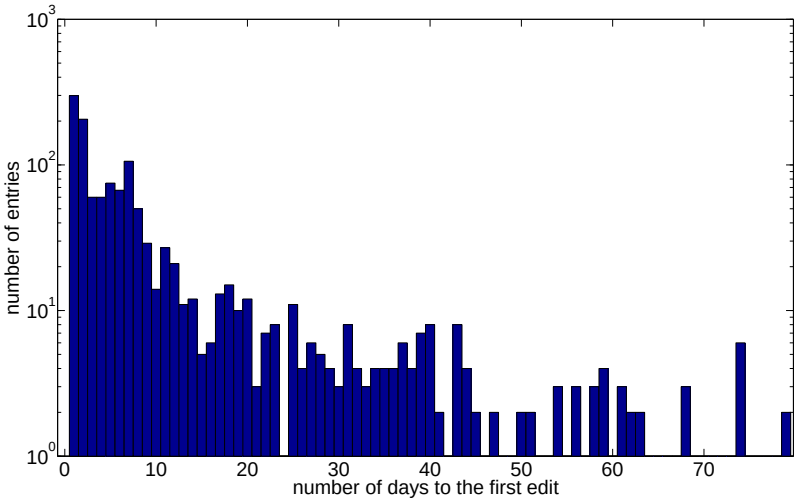
After how many days was the entry first edited?
After how many days was the entry last edited?

These are very interesting questions since the answers tell us how reliable the published information is regarding edits after a certain period of time. Entries for which the original entry add date is unknown no precise duration can be calculated and are, therefore, excluded. This means the edit durations are only calculated for entries for which the entry add was observed.

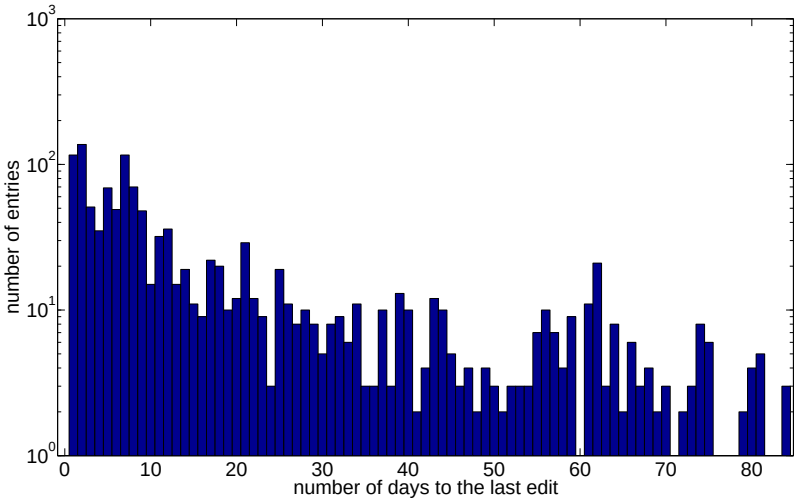
In figure 4.19(a) each bar represents the number of entries that were edited the first time after the number of days equal to its position on the x-axis. The second figure 4.19(b) shows the same counts for the last time the entry was edited. Their y-axis use a logarithmic scale. The last figure shows how often the entry was edited. The entries that do not experience a change at all are excluded of this analysis. They make up over 61.9% of the entries in data set and only the remaining 38.1% actually experience an edit in the observed time.

The figures for the first edit (figure 4.19(a)) and the last edit (figure 4.19(b)) look very much alike. This is because many entries are only edited once as can be seen in 4.19(c). Over 93.5% of the first and 81.4% of the last edits fall within the first 30 days but 93.5% of the first and 99% of the last edits are conducted within the first 60 days after the entry was added.

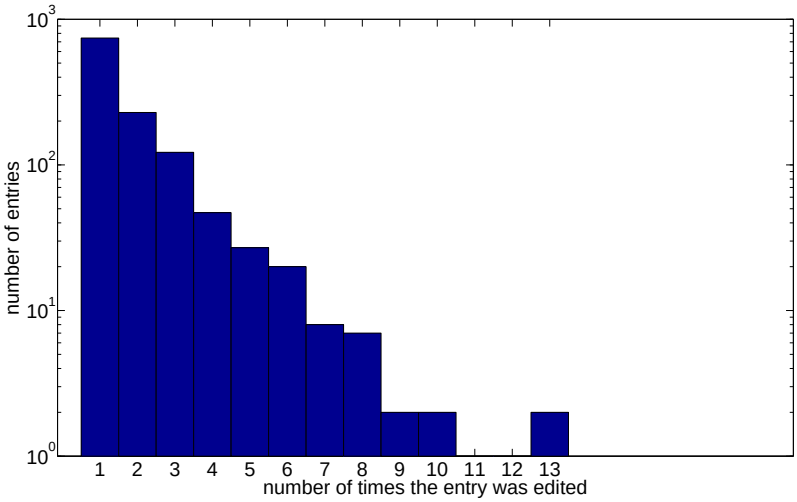
Most of these edits are not really substantial because they are mainly edits in the products list or the references. When *reference*, *product*, *credits*, *time*, *filter*, *keywords* and *manualTesting* are excluded, the results change slightly. Only



(a) Number of entries counted by the duration until the first edit



(b) Number of entries counted by the duration until the last edit



(c) Count of the times the entry was edited

Figure 4.19: OSVDB count of entries to the first and last edit

14.8% of the entries experienced an edit in substantial elements and 85.2% remained unchanged.

The majority of the entries their first substantial edit (4.20(a)) within the first 30 days after the entry was added. This group accounts for 94.2% of the entries.

For the most recent edit (4.20(b)) there is also a large group of entries edited within the first 30 days after the entry was added which accounts for over 92.7% of the edited entries.

It is important to note that most of this information is edited only once rarely twice as can be seen in figure 4.20(c).

This picture might change when there is more data collected. With a longer observation period a more distinctive distribution regarding all edits might emerge as well as there are probably more substantial edits after a longer period of time.

When are new entries added?

For this question only the newly added entries were considered. The figure 4.21(a) shows the number of newly added entries per month. The months April and July might have a few more new entries if the whole month would be present but there are clearly more new entries in May and June.

Figure 4.21(b) shows the distribution of newly added entries over the days of the month. The peaks on the 29th and 10th are again due to the accumulated edits because of the download errors on 10th of May and 29th of June. The new entries on the 12th are mainly caused by one day on 12th of June. The large number of new entries on 13th and 19th is not caused by just one date. The sums of these days build up over several months, and therefore it seems that on these dates there is increased activity for adding new entries.

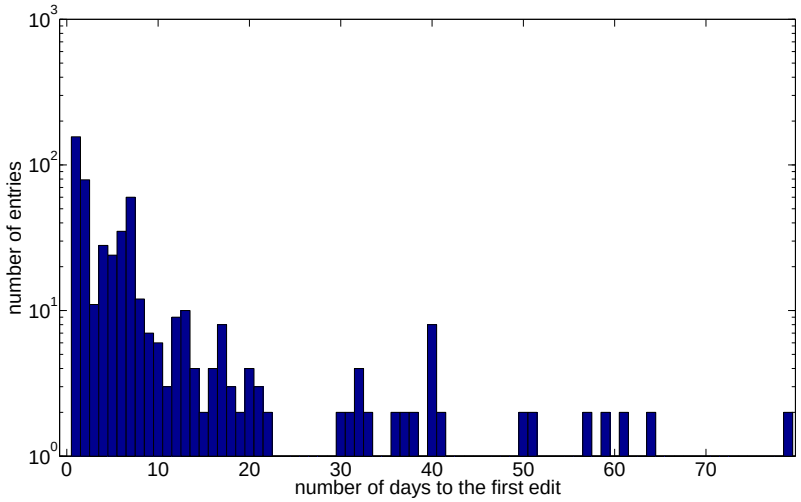
When these larger edits are ignored the following pattern emerges. At the beginning of the month entries are added which drops towards the 10th and stays low until the 16th. On the 17th it increases again fast with a peak on the 19th and then slowly decreases until the 26th. At the end of the month the number of newly added entries might actually stay or increase but with the limited data set it is hard to say.

How complete are the entries?

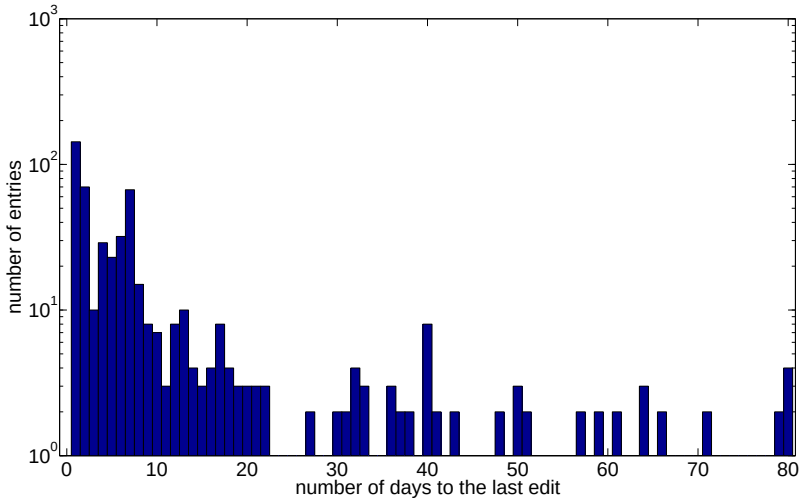
This question rose after noticing that each OSVDB entry offers a completeness percentage. The value measures how many fields of the entry are filled out. Figure 4.22 shows the distribution of the completeness values.

A large number of entries are only partially filled out. Over 53% of the entries are actually less than 50% complete whereas 35% of the entries are 80% or more complete. Only in 16% of all entries in the OSVDB all mandatory fields are filled out.

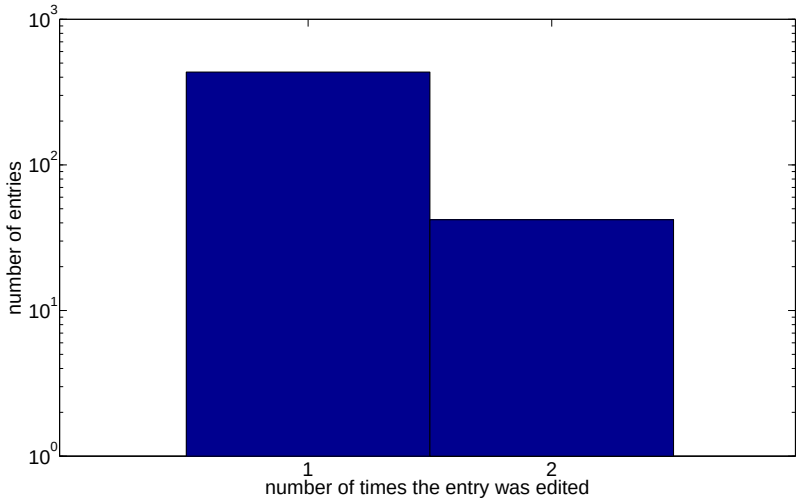
Since not all field are mandatory it is actually possible for an entry to reach 100% even if some important data is missing. For some of the fields this makes



(a) Number of entries counted by the duration until the first edit

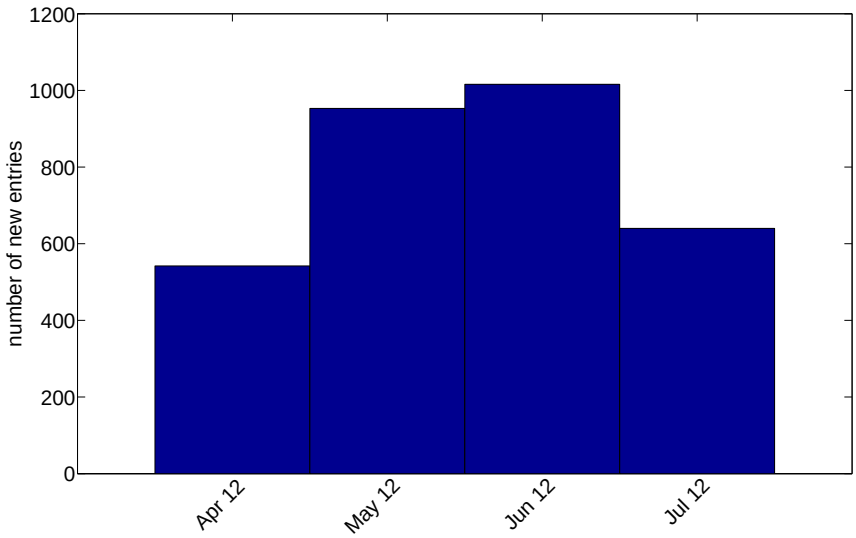


(b) Number of entries counted by the duration until the last edit

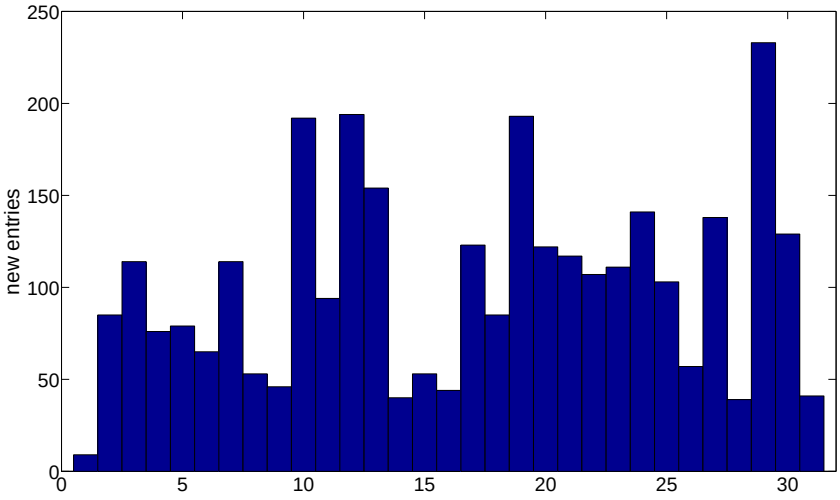


(c) Count of the times the entry was edited

Figure 4.20: OSVDB count of entries to the first and last substantial edit



(a) New entries per month



(b) New entries per day in the month

Figure 4.21: OSVDB new added entries

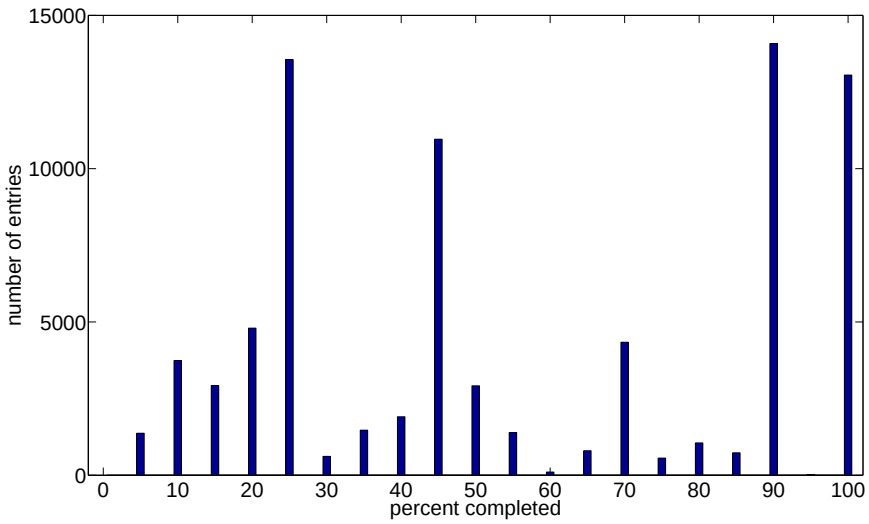


Figure 4.22: OSVDB self claimed entry completeness

sense because it is additional information such as the *Solution*, *Technical*, *Tools and Filters* or *Credit*.

But also the CVSS section which contains a threat rating of the vulnerability is also a non-mandatory field.

4.3 History Comparison

The optimal way to compare the results is to use data sets which are obtained from the sources within the same time window. Currently there are nearly four months of data available for the Open Source Vulnerability Database which proves to be not enough because the results are easily distorted by a few extreme edits. To minimize this problem all the available data in each data set is used.

This complicates the comparisons of raw numbers. The numbers can either only be compared within a specific time window existing in both data sets or have to be normalized. Another possibility is to compare the general observed patterns.

4.3.1 Comparing the Overview and Edits by Tags

Both vulnerability databases show that there is more information added over the time than removed. This effect is very extreme in the OSVDB where 83% of all edits are newly added elements whereas the NVD reaches only 45%. These values are actually an upper bound for the added elements because of the *attribute-edit* effect mentioned in 4.1.4.

To create the lower bound for the added elements it has to be assumed that all the removes and a corresponding number of adds come from this effect. For the OSVDB the lower bound for adds is at 74% and for the NVD at 16%.

In case of the lower bound of OSVDB the added elements still dominate the number of edits whereas in the NVD the changes dominate.

When looking at the edit distribution by tags (Figures 4.1 and 4.13) both databases exhibit very similar characteristics. The references and products are the elements that get edited most often and information related to the threat rating is changed rarely. The field in between is mainly filled by elements giving additional information about the vulnerability such as the common weakness enumeration or the timeline.

4.3.2 Comparing the Edits by Tags and by Entries

The general characteristic, that newer entries are more likely to experience updates, is visible for both databases (figures 4.2 and 4.14). The number of times the entries are edited are fairly equal whereas the number of performed edits differs greatly. Entries in the OSVDB are unlikely to experience more than 100 edits but for entries in the NVD this is not that unusual. This is mainly due to updates in the vulnerable software list which includes all the versions of a vulnerable application. When an application with a large number of versions is vulnerable (e.g., Google Chrome) this immediately causes a large number of edits. In such a case the OSVDB usually only lists the highest vulnerable version which significantly reduces the number of edits.

4.3.3 Comparing the Edits by Days and by Month

Regarding the daily number of edits the NVD and OSVDB are in a very similar range (figures 4.4 and 4.16). This seems to contradict the numbers given in the initial overview where the NVD shows nearly six time the average edits per day as the OSVDB. But this is due to a very few days with an extreme amount of edits like the Linux name change day which alone accounts already for 20.6% of all observed edits in the NVD.

In the daily number of edited entries the NVD and OSVDB deviate. In the NVD rarely more than 100 entries are edited per day whereas in the OSVDB the majority of the days have between 50 and 300 edited entries.

Together these observations show that in the OSVDB generally more entries are edited but with each edit less elements are changed.

Additionally the edits per day of the NVD show two distinct groups of work days. The group with less edits and edited entries corresponds to weekend days whereas the group with more edits and edited entries corresponds to work performed during the week. This is also clearly supported by the number of edited entries per week days in the month in figure 4.7(b). This indicates that the work is performed by someone with regulated work times which suggests an employee.For the OSVDB currently no such weekly working pattern can be found. In figure 4.18(b) only the week days in the first and last week have clearly more edited entries than the weekend days. This suggests that either we have currently not enough data or there is no distinct working patter. In case there really is no such weekly working pattern this might indicate that the

work is also performed off hours and/or on weekends which would be consistent with an open source project.

When looking at the total edits per month the general pattern is basically the same (figures 4.5 and 4.18). The total number of edits is also in a very similar range, except for June when the OSVDB has many more edits which is caused by adding a lot of credits and references. In the NVD the relatively large number of edits compared to the edited entries in June can be explained by updates in the vulnerable software list in several old entries.

The number of edited entries in the OSVDB is in general higher than in the NVD which has several reasons. For one, more new entries are published per month in the OSVDB than in the NVD (4.3.5). These entries can also be published when they are only partially filled out and have to be completed over time. And second, the OSVDB is currently under reconstruction and meanwhile also the entries seem to get content updates like the credits.

4.3.4 Comparing the Time until first and last Edit

In the NVD over 82% of the newly added entries are edited whereas in the OSVDB only 38% of the entries experience an edit. In both databases the majority of the edited entries experience their edit within the first 30 days. In the NVD also a larger number of edits experiences their edit between 50 and 150 days after the entry was added. In the OSVDB this is not the case (figures 4.8(a) and 4.8(b)). This is probably because we did not collect enough data for the OSVDB yet.

When only looking at the substantial changes (figures 4.9 and 4.20) both databases show that if the information is edited, it is done very shortly after the entry was added. In the NVD over 58% of the newly added entries whereas in the OSVDB only 14.8% of the new entries experience a substantial edit. This is mainly because for the NVD it is quite common that the CVSS section is added the day after the entry was published (figure 4.12(c)).

4.3.5 Comparing the new Entries

The newly added entries per month (figures 4.10(a) and 4.21(a)) show a very similar patter in the corresponding months. Also the general pattern in the days of the month (figures 4.10(b) and 4.21(b)) mostly matches since at the beginning of the month more new entries are published as well as increased publishing around the 20th as well as at the end of the month.

The total number of newly added entries are much higher in the OSVDB than in the NVD. In the currently available data set the OSVDB added from 1.72 up to 2.47 times the new entries the NVD added. This is mainly because the OSVDB has a wider range of vulnerabilities it tries to document and uses many different of data sources. Therefore, a large number of the new entries does have a reference to the NVD.

Among the sources is also the CVE list which contains reserved identifiers that are also imported into the OSVDB. Such reserved entries are incomplete, and therefore not published by NVD until all the information is disclosed. If an

entry in the OSVDB was imported from the CVE list this is clearly visible in the description field by the provided reference until it is edited. Currently over 43% of the present entries in the OSVDB are directly imported entries from the CVE list. These seem to be mainly older entries because only 9% of the newly added entries were imported from the CVE list which means that most of the new entries are created from other sources.

4.3.6 Comparing the Time until Entries are updated

For this comparison several entries of Mozilla related vulnerabilities were chosen because the Mozilla project also maintains their own security advisory database for all the vulnerabilities related to their software.

The investigated entries were all officially reported on 24th of April 2012 which allows for enough time to pass to observe the edits. The involved identifiers can be split into two groups.

- The OSVDB IDs 79872 to 79878 which correspond to CVE-2012-1126 until CVE-2012-1132 and 79880 to 79891 which correspond to CVE-2012-1133 until CVE-2012-1144 as well as OSVDB ID 79879.
- The OSVDB IDs 81513 to 81524 which correspond to CVE-2012-0467 until CVE-2012-0479, 81525 corresponding to CVE-2011-3062 and 81526 corresponding to CVE-2011-1187.

All the entries in the NVD, except CVE-2011-3062 and CVE-2011-1187, are released on the 25th of April and not edited since. This means that these entries where created on the day when Mozilla released the information. The remaining two entries with identifiers issued in 2011 are already existing entries in which on the 26th of April (published on the 27th of April) the existing references are changed from Bugzilla to the security advisories. All the updates are performed and published within three days after Mozilla has published the advisories.

For the OSVDB this looks very different. The entries of the first group appear for the first time before the data collection started as imports of reserved identifiers from the CVE list. According to the information available in the OSVDB entries these vulnerabilities were disclosed on the 23rd of February 2012 and a solution was available on 24th of February. This is basically correct if the focus is on the component *FreeType* for which an update was released on that date rather than on the Mozilla product *Firefox Mobile*. On the 27th of April the updated summaries of the CVE list are imported to these entries which state *Firefox Mobile* as possible vulnerable product but the products list does not get updated. Later only references and credits are added.

The entries of the second group are all newly added on the 28th of April, except for the entries with the IDs 81525 and 81526 which were added on the 30th of April. The description of these entries clearly show that they were imported from the CVE list. On the 2nd to 5th of May the descriptions are rewritten and the reference to the CVE import is removed. All the entries state that the disclosure and vendor solution date is on the 24th of April which

corresponds to the information in the Mozilla security advisories. Later only references and credits are added.

This allows us to create to following timeline:

1. On the 24th of April Mozilla releases the updates and the vulnerability information.
2. On the 25th of April the CVE and NVD entries are updated. In the early morning of the 26th of April the NVD is exported and published on the website.
3. On the 26th of April the already existing entries in the OSVDB are updated with the newly available information in the CVE list.
4. On the 27th of April the new entries are added to the OSVDB created from information available in the CVE list.
5. Between the 2nd and 5th of May the newly added entries in the OSVDB are manually updated.
6. References and credits are added to the entries in the OSVDB during May and June.

Similar timelines can be found for many different vulnerabilities. If for the corresponding OSVDB entry exists also a NVD entry then the NVD entry is mostly updated within three days after the information is officially disclosed. For the OSVDB it takes between 7 and 21 days until the information is updated.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The overall edit pattern in the NVD and the OSVDB are very similar. Newer edits experience more edits than older ones, the number of edits by day of month follows a similar pattern in the NVD as in the OSVDB, and similarly for the monthly publishing cycle for new entries. Therefore, it is safe to assume that the observed edits are maintenance effects.

Most of the edits in both databases are performed in the references and vulnerable products. Also other elements such as the last modified date, credits, timeline, solution, manualTesting, keywords and scanners and filters experience quite a few edits. The impact of edits within all these elements is not that serious because they refer to internal or external resources, make information finding easier or give additional information about how to test or resolve the vulnerability.

When the observation of edited elements is limited to more substantial elements like the summary, the description, the CWE and the CVSS information then nearly all the edits occur within the first 30 days after the entry was added. This shows that 30 days after the entry was added the information of these elements can be regarded as stable.

Many of the later edits in substantial elements in the NVD are caused by CVSS elements. Most of the edits in these elements also happen very closely to the publish date of the entry but it sometimes happens that new information about the vulnerability is discovered which requires that the CVSS threat rating has to be adjusted.

Therefore, we conclude that it is possible to use National Vulnerability Database as ground truth for not very fresh entries.

In the OSVDB there are several fields that are not mandatory. These fields are *manual testing notes*, *keywords*, *technical*, *solution*, *tools & filters* and *CVSSv2*. In the entries where a *CVSS* rating is given it is actually imported from the NVD. But in case the rating is changed in the NVD, it does not get updated in the OSVDB. Even if these fields are not filled the entry can reach a completeness rating of 100%.

It is possible to publish incomplete entries in the OSVDB which should be

completed over time. This causes that the majority of the entries are currently incomplete. This theoretically allows that new entries can get published faster and the community can help to complete the missing information. In reality it takes longer until information is updated in the OSVDB than in the NVD. These problems are likely caused by is lacking resources and support of its users.

The upside of the OSVDB is that since it seems to use more data sources, amongst which also the CVE list and the NVD are, it is possible to find information to vulnerabilities which are not present or not yet published in the NVD.

The Open Source Vulnerability Database is, therefore, a good source to find additional information to NVD entries or vulnerabilities which are not (yet) present in the NVD. In case the information was not only imported from the CVE list then an the OSVDB entry can be used as second opinion to the related NVD entry.

5.1.1 Answering the Assignment Questions

Most of the questions of the assignment are answered in the results chapter (4.2.1) and are, therefore, not discussed again. These questions are:

- What changed?
- How did it change?
- When did it change?
- Are there any periodic changes?
- Are there times of large changes?
- When entries are newly added to the NVD?
- Are there times of large (bulk) changes?

Does the *published-datetime* field reflect the actual date when the entry was first time visible?

The date given in the element *vuln:published-datetime* in the NVD reflects the date on which the entry was added. In the history the entry is the first time visible the day after because the database is exported and published the next day in the early morning.

Are there any edit wars?
Are there things that much-changed entries have in common?

Entries that are changed very often do not exhibit signs of an edit war as can be seen in 4.2.1. The entries that were investigated for edit wares share that they are changed very frequently. These changes have in common that a set of elements is edited over and over again. Mostly these are elements in the vulnerable products list and/or the vulnerable configuration list which are

added, removed and added again. This is not done for all the entries on the same day and also not every day but happens every two to five days.

How does the churn develop over time?

This question is answered indirectly in the NVD results 4.2.1. There are results per day and per month. No clear pattern was is visible but is was possible to locate a backlog that was processed in late 2011.

Are there periodic phenomena?

Yes as can be seen in the NVD results 4.2.1. There is a weekly working pattern, monthly edit pattern, monthly new entry pattern but also the Microsoft patch day is visible.

Are there entries that appear first in one database then in the other? How can these entries be characterized? Is one consistently ahead of the other?

For this only entries that are shared can be compared which means only entries with a CVE identifier. The NVD is based on the CVE list and the OSVDB imports the CVE list. The major difference is that the NVD only releases vulnerability information to CVE entries which are disclosed, therefore reserved CVE identifiers do not show up in the NVD. The OSVDB imports all the entries of the CVE list.

In general the NVD releases information faster than the OSVDB. The NVD publishes newly available, previously unreserved vulnerability information faster than the OSVDB. Regarding reserved CVE identifiers the NVD waits until the information is officially disclosed. Since the reserved CVE identifiers are also imported into the OSVDB it is possible that the entry gets updated, with information from other sources than CVE or NVD, before the information is officially disclosed. This can be seen in the Mozilla example in 4.3.6.

Are the changes consistent?

It seems that after the entry was manually updated in the OVSDB the data is not imported from the NVD anymore. If the CVSS information is changed in the NVD after it was imported into the OSVDB then the CVSS information in the OSVDB does not get updated. In this case the changes are not consistent. A more throughout analysis could be not performed due to time constraints.

5.2 Future Work

In this section possible future work based on this thesis is proposed.

In the performed analysis it was rarely necessary to compare the content edits of the edit type *change* which are stored in the parameters *data_old* and

data_new. Analyzing the content might give better results for example by separating the fixing of a typo from a relevant edit in certain elements, or allow deeper analysis such as the distribution of the edited CVSS ratings.

With more data available for the OSVDB a more systematic inter-database comparison could be performed.

During the XML comparison, whenever an edit in the text node of an uniquely identifiable XML element occurs, the history entry only states the tag name (in the parameter *tag*) in which the text was changed from *data_old* to *data_new*. This is not enough because multiple tags with the same tag name can exist in the same entry but they are uniquely identifiable by their tag name and attributes (e.g., element *time* in the OSVDB). This can be resolved by either writing the tag name and attributes to the *tag* parameter or by giving the whole old and new XML element in *data_old* and *data_new*.

Tools using multiple scripts sometimes share variables such as work or SVN repository directories. Currently these are defined in each script separately at the beginning or handed over as arguments. This has a high chance for errors which might cause that the tool is not running or checking in information in the wrong directory. This can be improved by a shared properties or configuration file.

The way and amount of data stored of the OSVDB can be optimized. More data fields that might be of interest could be added (e.g., self claimed completeness). Currently the XML element tag names do not use a namespace like in the NVD which might cause them to clash with other variables. This is currently the case with the CVSSv2 *date*, stating the CVSS creation date, which clashes with the edit history entry *date*, stating the edit date, in the default evaluation script. With the increasing number of entries it might be necessary to split the XML file into smaller files to reduce the required system resources and processing time.

Appendix A

"How To" Descriptions

This chapter holds the "how to" descriptions for the developed scripts in this thesis. There are three different types of executables used which are shell scripts, Perl scripts and a java archive (JAR). The usage of the different typed will be described in the following subsections. The other sections of this chapter combine the executables into functional groups that performs a larger task (e.g history creation). The subsections focus on a specific task such as creating the history of the NVD and will specify the necessary details how to use them.

A.0.1 Shell Scripts

The shell scripts are primarily used to perform system operation (e.g create and remove directories, checkout svn directories, etc.). Therefore, they often contain many variables at the beginning of the script defining SVN sources, working directories, grep patterns etc. These should be checked out and if necessary adjusted before the scripts are run, especially when multiple scripts have to work together to perform a task, because often the scripts share variables that have to match.

A.0.2 Perl Scripts

The Perl scripts are used to perform string based operations such as extracting the useful information of one OSVDB website entry. These scripts normally do not perform system operations and assume that the working environment (e.g., working and output directories) are properly set up by the shell script that called them. The required information (input files, output directory, etc.) are then given as argument to the Perl script. Still it is recommended to short check the variables of the Perl scripts before first execution. It is possible to directly call a Perl script given that the environment is prepared and the necessary arguments are given to the script.

A.0.3 Java Archive

The only jar file is used for the comparison of the XMLfiles. It is built under similar assumptions as for the Perl scripts regarding the run environment, which

has to be prepared by other means, such as a shell script. It is also possible to run the jar file directly given that the environment is prepared and the necessary arguments are given to the executable.

A.1 Data Collection

This section documents the scripts involved in the data collection. Since the data collection of the NVD was not part of this thesis it will not be covered here.

A.1.1 OSVDB Data Collection

This subsection describes the OSVDB data collection. This task is achieved in two steps. First the information is fetched in HTML form. In a second step the useful information is extracted omitting the not needed HTML code. This can be done either in two separate steps (*legacy full HTML download* and *legacy XML processing*) or directly in one step (*download and direct XML processing*). It is recommended to use the *download and direct XML procession* version. The separate steps are also documented here for completeness and because they still might be useful but they are regarded legacy.

Legacy Full HTML Download

The task of these scripts is to download and archive the complete HTML data of all the OSVDB entries. They are stored in different files based on the year of the first encountered CVE id. When the download is completed the data is committed to the repository. All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/osvdb-html/*. Their specific tasks are the following:

- *runOsvdbUpdate.sh* checks out the log directory and runs *updateOsvdbCveData.sh* forwarding the output to a log file which is committed to the repository after completion.

- *updateOsvdbCveData.sh* checks out the data and script directory, prepares the work directory, copies the scripts to the work directory and runs *getOSVDB.pl*. When the download is completed the data files are copied to the SVN working directories and committed to the repository.

- *getOVSDB.pl* fetches one OSVDB entry after another which are saved in file matching the year of the first CVE ID found in the entry. Entries without a CVE ID are stored into a separate file. The script will stop running when the number of consecutive empty entries reaches the value specified in *$maxEmptyRow*.

Default usage

1. Check the SVN repository for the directory where the data will be stored and that it contains a log directory.
2. Check that *runOsvdbUpdate.sh* and *updateOsvdbCveData.sh* are locally available.
3. Check the variables in *updateOsvdbCveData.sh* and *runOsvdbUpdate.sh* to match the SVN repsitory paths, local working directories and script paths.
4. Check the variables in *getOSVDB.pl*. Pay special attention to *$BaseCounter* that defines the first id checked and *$maxEmptyRow* that determines after how many consecutive empty OSVDB entries the script will stop.
5. Check that that the changes of *getOSVDB.pl* are committed.
6. Run *runOsvdbUpdate.sh*. It is recommended to forward the output to a file and run the script in the background since it can take from several hours.

It is possible to run the scripts on their own but in general it is recommended to use *runOsvdbUpdate.sh* since it will also commit the log to the SVN repository. No Arguments are passed along to any of these scripts.

Legacy XML Processing

The task of these scripts is to extract the useful information of previously downloaded full HTML entries and store it in one XML file which is committed to the repository. The historic order of the full HTML revisions will be preserved. This process is implemented in a server-worker architecture to distribute the significant workload.

The server creates a list of tasks that it has to complete. Each task is stored in a task file that contains the following information: 1. the revision number associated with the task, 2. the revision number of the preceding task that has to be finished first, 3. the list of files that have to be processed to complete this task. 4. the time the task was created. Furthermore the task file is also used as the sentinel file to the corresponding revision to signal that the revision has been processed and committed to the repository. Therefore, the task file is committed to the repository in the sentinel directory.

Based on these tasks jobs are created which are a simple mapping of one file and one revision. These jobs are then sent to the workers. Such a job file contains the following information: 1. the URL to the repository directory where the file can be found that has to be processed, 2. the file name of the file that has to be processed, 3. the revision number, 4. the server identifier for ssh to send the result, 5. the directory the result is expected to be sent to, 6. the file name of the result, 7. the worker the job was sent to, 8. the time the job was sent to the worker, 9. the time the result was present on the server. The

jobs then are processed by the workers and the result is sent back to the server that checks which tasks are completed and commits the processed data to the SVN repository. The requirements and restrictions are the following:

- An existing directory on the SVN repository with full HTML entry data of the OSVDB as data source is needed.
- An existing directory on the SVN repository with a sub-directory for sentinel files, to store the output and sentinel files of already processed revisions.
- It starts processing with the first revision in the HTML source for which no sentinel file is present. Caution is advised when working on a SVN repository directory that already contains data. If a revision of the data source is unwanted it has to be excluded of processing in the script (e.g., *$F_REVISION_BROKEN* in *legacyOsvdbPreprocessInitTasks.sh*) or by a sentinel file.
- It will continue until the last revision present at the moment of initialization.
- The server cannot be a worker.
- The *legacyOsvdbPreprocessUpdateTasks.sh* hast to be run frequently (i.e., cronjob every 30 min) to ensure continuous processing.
- If a worker fails to deliver a result the server might still send out new jobs to be processed. But because it cannot complete the corresponding task, it will not commit revisions beyond the failed task. This can be resolved by manually moving the job from *working* to *todo* after deleting some of the information (please consult another job file in *todo*) or wait for all the jobs currently being processed and then re-run *legacyOsvdbPreprocessInit-Tasks.sh*. Doing this will preserve previously committed revisions due to the sentinel files but also already processed jobs for which the task was not completed yet in the previous run, as long as the *receive* directory is not cleaned up.

All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/osvdb-xml/* with the prefix *legacyOsvdbPreprocess*. Their specific tasks are the following:

- *legacyOsvdbPreprocessInitTasks.sh* is run on the server and initializes it. The script creates a list of worker machines to send jobs to, the task files, the jobs files of the jobs that have to be processed by the workers.
- *legacyOsvdbPreprocessUpdateTasks.sh* is run on the server and checks for received results and completed tasks. If a task is completed (the previous revision is committed and all the jobs of the task are processed) combines the results and commits it to the SVN repository. It also sends new jobs to the workers that are working on less than *$MAX_JOB* jobs.

- *legacyOsvdbPreprocessHtml.sh* is stared on the worker by the server over SSH. It exports the given file and revision and runs *legacyOsvdbPreprocessHtml.pl*. Upon completion the result is sent back to the server to the given *receive* directory.
- *legacyOsvdbPreprocessHtml.pl* is run on the worker. It processes the given file and outputs the result to the given output path. If in this script any changes are made that change the output in any way, the same changes have to be implemented in *osvdbPreprocessHtml.pl* to ensure consistency.

Default usage

1. Check the SVN repository for the directory where the XML files will be stored and make sure that it contains a sentinel directory (i.e., *done-revisions*).
2. Check that *legacyOsvdbPreprocessInitTasks.sh* and *legacyOsvdbPreprocessUpdateTasks.sh* are locally available to the server.
3. check that *legacyOsvdbPreprocessHtml.sh* and *legacyOsvdbPreprocessHtml.pl* are locally available to the workers.
4. Check the variables in *legacyOsvdbPreprocessInitTasks.sh* and *legacyOsvdbPreprocessHtml.sh* to match the SVN repository, the work directories to suit the server as well as the limiting variables (i.e., *$MAX_JOBS*, *$F_REVISION_BROKEN*, etc.).
5. check the variables in *legacyOsvdbPreprocessHtml.sh* to suit the workers.
6. Run *legacyOsvdbPreprocessInitTasks.sh* on the server.
7. Run *legacyOsvdbPreprocessUpdateTasks.sh* on the server periodically. The time has to be chosen carefully since a smaller time increases the performance but it can lead to concurrent running instances of the script because the svn commit can take some time. It is recommended to forward the output to a file or /dev/null and run the script in the background since it can take from several hours to several days.

It is not possible to run *legacyOsvdbPreprocessUpdateTasks.sh* prior to *legacyOsvdbPreprocessInitTasks.sh* since no tasks and jobs are present. It is possible to run *legacyOsvdbPreprocessHtml.sh* and *legacyOsvdbPreprocessHtml.pl* manually. The required arguments are explained in the respective scripts.

Download and direct XML Processing

These scripts combine the tasks into one, and therefore download and extract the useful information of the OSVDB. This is done in parallel to increase the performance. The results are then combined and committed as a combined XML file to the SVN repository. This greatly reduces the time required but because up to 2**$MAX_JOBS* of extraction tasks are running simultaneously, it might consume quite some system resources. It is not possible to process

legacy full HTML downloads. For this please refer to *Legacy XML Processing*. All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/osvdb-xml/* without the prefix *legacy*. Their specific tasks are the following:

- *runOsvdbGetAndProcess.sh* checks out the log directory and runs *osvdbGetAndProcess.sh* forwarding the output to a log file. Upon completion the log file is commited to the repository.
- *osvdbGetAndProcess.sh* checks out the data directory and prepares the work directories. It then starts the number of download tasks (*osvdbGet.pl*) specified by *$MAX_JOBS* to download a specified amount of entries and sleeps for *$SLEEP* seconds. After each wakeup it checks if the downloads are done and if so, starts for each done download a preprocessing task (*osvdbPreprocessHtml.pl*) and a new download. This is done until the all the entries are downloaded and processed. The results are then combined and committed to the repository.
- *osvdbFrame.pl* creates the head and tail for the combined XML file. This is a convenience file to simplify the combination process.
- *osvdbGet.pl* downloads the complete HTML data for a range of OSVDB entries to a temporary location.
- *osvdbPreprocessHtml.pl* extracts the useful information from the HTML entries downloaded by *osvdbGet.pl*. After the processing is completed, the sentinel file is deleted to signal that the processing is done. If in this script any changes are made that change the output in any way, the same changes have to be implemented in *legacyOsvdbPreprocessHtml.pl* to ensure consistency.

Default usage

1. Check the SVN repository for the directory where the resulting XML file will be stored and make sure that it contains a log directory.
2. Check that *runOsvdbGetAndProcess.sh* and *osvdbGetAndProcess.sh* are locally available.
3. Check the variables in *unOsvdbGetAndProcess.sh* and *osvdbGetAndProcess.sh* to match the SVN repository, the work directories and for correct runtime variables.
4. Check the variables in *osvdbFrame.pl* to match the ones in *osvdbGetAndProcess.sh*.
5. Check that the changes of *osvdbFrame.pl*, *osvdbGet.pl* and *osvdbPreprocessHtml.pl* are commited.
6. Run *runOsvdbGetAndProcess.sh*. It is recommended to forward the output to a file and run the script in the background since it can take from several hours.

It is possible to run *osvdbGetAndProcess.sh* directly but in general it is recommended to use *runOsvdbGetAndProcess.sh* since it will also commit the log to the SVN repository. Directly running *osvdbGet.pl* or *osvdbPreprocessHtml.pl* is possible if the required arguments are provided. The arguments are described in the respective scripts.

A.2 History Creation

This section documents the scripts involved in the history creation. The scripts produce a history of the given data source by comparing the content of a revision with its previous revision. The resulting history documents each difference as a *change*, *add* or *remove*. The resulting format is the same for all data sources since every history creation script bundle uses the same executable (*runCompareVdbXml.jar*) to create the history.

A.2.1 NVD History Creation

The task of these scripts is to create the edit history of a specific NVD XML file which is defined by its year. It first creates the revision list of the file given by the year. Then it compares each revision with the previous revision to create the complete edit history. The result is committed to the repository. Caution, do not mix history files created by these scripts with a history created by *NVD incremental history creation* (A.2.2) because this method does not create the sentinel files required by the incremental version. Since the source data is split into several files defined by its year the resulting history file will also only contain the history of the processed source file. Since the separate files are much smaller, the processing requires less resources. Additionally, processes can be started on different machines (two or more processes on the same machine will not work since they will use the same working directory) to process the files in parallel. In case it is necessary to create one large history, one can combine the files and sort by date or create an according *history evaluation* operation (see Section A.3). All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/history-generate/*, most of them contain *nvd* in their name. Their specific tasks are the following:

- *runNvdCreateHistory.sh* checks out the log directory and runs *nvdCreateHistory.sh* with the given *year* as argument. Upon completion the log is committed to the repository.

- *nvdCreateHistory.sh* checks out the script directory, the input data directory and history directory. It then sets up the work directory where the scripts are copied to and creates the revision list of the input file defined by the given year. For each revision it runs *nvdXmlCompareWrapperEmpty.sh* or *svn diff* with *nvdXmlCompareWrapper.sh* which writes the differences to the history file. When all revisions are processed the history file is committed to the repository.

- *nvdXmlCompareWrapper.sh* is a wrapper script for *svn diff* to correctly run *runCompareVdbXml.jar* with the necessary arguments. It also specifies the memory available to the java runtime environment.

- *nvdXmlCompareWrapperEmpty.sh* is a similar wrapper script that is only called when the first revision is processed. This is necessary because for the first revision exists no previous revision, and therefore uses *nvdcve-2.0-empty.xml* instead. It also specifies the memory available to the java runtime environment.

- *runCompareVdbXml.jar* detects the changes on the XML element level between the two given input files and writes the result to the specified output file. The *vuln:references* are an exception to the XML element level comparison because the element can only be uniquely identified with their sub-elements. Therefore, the references are compared as a whole.

Default usage

1. Check the SVN repository for the directory where the history file(s) will be stored and make sure that it contains a log directory.

2. Check that *runNvdCreateHistory.sh* and *nvdCreateHistory.sh* are locally available.

3. Check the variables in *runNvdCreateHistory.sh* and *nvdCreateHistory.sh* to match the SVN repository, the work directories and for correct runtime variables.

4. Check the variables in *nvdXmlCompareWrapper.sh* and *nvdXmlCompare-WrapperEmpty.sh* to match the used data file type (*nvd*) as well as that the called jar file will have enough memory (at least 20 times the size of the largest file that will be compared). Allocating too little memory will cause the comparison to fail while too much could cause swapping or failure due to not enough available memory.

5. Check that the changes of *nvdXmlCompareWrapper.sh*, *nvdXmlCompare-WrapperEmpty.sh* and *runCompareVdbXml.jar* are committed.

6. Run *runNvdCreateHistory.sh <year>* where *year* is the year in the file name that will be processed. It is recommended to forward the output to a file and run the script in the background since it can take from several hours to several days.

It is possible to run *nvdCreateHistory.sh* directly but in general it is recommended to use *runNvdCreateHistory.sh* since it will also commit the log to the repository. To only get the difference between two files one can either run *runCompareVdbXml.jar* directly or use *nvdXmlCompareWrapper.sh* with the required arguments. The necessary arguments are described in *nvdXmlCompareWrapper.sh* or in the java source code of the jar file.

A.2.2 NVD incremental History Creation

The task of these scripts is to update the already present edit history of a specific NVD XML. At first it creates the revision list of the file and then creates a job for each revision that was not previously processed (i.e., for which no sentinel file exists). The result is appended to the existing history file and committed to the repository. Caution, do not mix history files created by these scripts with the history created by *NVD history creation* (A.2.1). Because the non-incremental version does not create the required sentinel files. Running the incremental version without the correct sentinel files present causes the history to contain duplicates. This is because the incremental scripts process every revision for which no sentinel file exists and appends the results to the existing history file. All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/history-generate/*, most of them contain *nvd* and *incremental* in their name. Their specific tasks are the following:

- *runNvdCreateHistoryIncrementalAll.sh* starts the *runNvdCreateHistoryIncremental.sh* on different machines with different years as argument. This is used to all the NVD history files as simple as possible.
- *runNvdCreateHistoryIncremental.sh* checks out the log directory and runs *nvdCreateHistoryIncremental.sh*. Upon completion the log is committed to the repository.
- *nvdCreateHistoryIncremental.sh* checks out the script directory, the input data directory and the history directory. It then sets up the work directory where the scripts are copied to and creates a job file for each not previously processed revision. For each job it runs *osvdbXmlCompareWrapperEmpty.sh* or *osvdbXmlCompareWrapper.sh* which writes the differences to the a temporary history file. After the revision is processed the additional history is appended to the existing history. The updated history and the respective sentinel file are committed to the repository. This is repeated until all jobs are processed.
- The descriptions of *osvdbXmlCompareWrapperEmpty.sh*, *osvdbXmlCompareWrapper.sh* and *runCompareVdbXml.jar* can be found in section A.2.1.

Default usage

1. Check the SVN repository for the directory where the history file will be stored and make sure that it contains a sub directory for the sentinel files and another sub directory for the log files.
2. Check that *runNvdCreateHistoryIncrementalAll.sh* will start all the necessary tasks on the assigned machine. Do not start more than one task on one machine.
3. Check that *runNvdCreateHistoryIncremental.sh* and *nvdCreateHistoryIncremental.sh* are locally available.

4. Check the variables in *runNvdCreateHistoryIncremental.sh* and *nvdCreateHistoryIncremental.sh* to match the SVN repository, the work directories and for correct runtime variables.

5. Check the variables of *nvdXmlCompareWrapper.sh* and *nvdXmlCompareWrapperEmpty.sh* to match the used data file type (*nvd*) as well as that the called jar file will have enough memory (at least 20 times the size of the largest file that will be processed). Allocating too little memory will cause the comparison to fail while too much could cause swapping or failure due to not enough available memory.

6. Check that the changes of *nvdXmlCompareWrapper.sh*, *nvdXmlCompareWrapperEmpty.sh* and *runCompareVdbXml.jar* are committed.

7. Run *runNvdCreateHistoryIncrementalAll.sh*.

It is possible to run *runNvdCreateHistoryIncremental* or *nvdCreateHistoryIncremental.sh* directly with the year as argument to just update one file. In case this is done it is recommended to use *runNvdCreateHistoryIncremental.sh* since it will also commit the log to the repository. To only get the difference between two files one can either run *runCompareVdbXml.jar* directly or use *nvdXmlCompareWrapper.sh* with the required arguments. The necessary arguments are describes in *nvdXmlCompareWrapper.sh* or in the source code of the jar file.

A.2.3 OSVDB History Creation

The task of these scripts is to create the edit history of the combined OSVDB XML file. At first, the revision list of the file is created and then each revision is compared with the previous revision to create the edit history. The result is committed to the repository. Caution, do not mix history files created by these scripts with a history created by *OSVDB incremental history creation* (A.2.4) because the non-incremental version does not create the sentinel files required by the incremental version. Running the incremental version without the correct sentinel files present causes the history to contain duplicates. This is because the incremental scripts process every revision for which no sentinel file exists and appends the results to the existing history file. All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/history-generate/*, most of them contain *osvdb* in their name. Their specific tasks are the following:

- *runOsvdbCreateHistory.sh* checks out the log directory and runs *osvdbCreateHistory.sh*. Upon completion the log is committed to the repository.

- *osvdbCreateHistory.sh* checks out the script directory, the input data directory and the history directory. It then sets up the work directory where the scripts are copied to and creates revisions list of the input file. For each revision it runs *osvdbXmlCompareWrapperEmpty.sh* or *osvdbXmlCompareWrapper.sh* which writes the differences to the history file. When all revisions are processed the history file is committed to the repository.

- *osvdbXmlCompareWrapper.sh* is a wrapper script for *svn diff* to correctly run *runCompareVdbXml.jar* with the necessary arguments. It also specifies the memory available to the java runtime environment.

- *osvdbXmlCompareWrapperEmpty.sh* is a similar wrapper script that is only called when the first revision is processed. This is necessary because for the first revision exists no previous revision, and therefore uses *osvdb-preprocessed-empty.xml* instead. It also specifies the memory available to the java runtime environment.

- *runCompareVdbXml.jar* detects the changes on the XML element level between two given input files and writes the result to the specified output file.

Default usage

1. Check the SVN repository for the directory where the history file will be stored and make sure that it contains a log directory.

2. Check that *runOsvdbCreateHistory.sh* and *osvdbCreateHistory.sh* are locally available.

3. Check the variables in *runOsvdbCreateHistory.sh* and *osvdbCreateHistory.sh* to match the SVN repository, the work directories and for correct runtime variables.

4. Check the variables of *osvdbXmlCompareWrapper.sh* and *osvdbXmlCompareWrapperEmpty.sh* to match the used data file type (*osvdb*) as well as that the called jar file will have enough memory (at least 20 times the size of the file that will be processed). Allocating too little memory will cause the comparison to fail while too much could cause swapping or failure due to not enough available memory.

5. Check that the changes of *osvdbXmlCompareWrapper.sh*, *osvdbXmlCompareWrapperEmpty.sh* and *runCompareVdbXml.jar* are committed.

6. Run *runOsvdbCreateHistory.sh*. It is recommended to forward the output to a file and run the script in the background since it can take from several hours to several days.

It is possible to run *osvdbCreateHistory.sh* directly but in general it is recommended to use *runOsvdbCreateHistory.sh* since it will also commit the log to the repository. To get the difference between two files one can either run *runCompareVdbXml.jar* directly or use *osvdbXmlCompareWrapper.sh* with the required arguments. The necessary arguments are describes in *osvdbXmlCompareWrapper.sh* or in the java source code of the jar file.

A.2.4 OSVDB incremental History Creation

The task of these scripts is to update the already present edit history of the combined OSVDB XML file. At first it creates the revision list of the file and then creates a job for each revision that was not previously processed (i.e., for which no sentinel file exists). The result is appended to the existing history file and committed to the repository. Caution, do not mix history files created by these scripts with the history created by *OSVDB history creation* (A.2.3). Because the non-incremental version does not create the required sentinel files. Running the incremental version without the correct sentinel files present causes the history to contain duplicates. This is because the incremental scripts process every revision for which no sentinel file is existing and appends the results to the existing history file. All the necessary scripts can be found on the SVN repository in the directory *svn//scripts/history-generate/*, most of them contain *osvdb* and *incremental* in their name. Their specific tasks are the following:

- *runOsvdbCreateHistoryIncremental.sh* checks out the log directory and runs *osvdbCreateHistoryIncremental.sh*. Upon completion the log is committed to the repository.
- *osvdbCreateHistoryIncremental.sh* checks out the script directory, the input data directory and the history directory. It then sets up the work directory where the scripts are copied to and creates a job file for each not previously processed revision. For each job it runs *osvdbXmlCompareWrapperEmpty.sh* or *osvdbXmlCompareWrapper.sh* which writes the differences to the a temporary history file. After the revision is processed the additional history is appended to the existing history. The updated history and the respective sentinel file are committed to the repository. This is repeated until all jobs are processed.
- The descriptions of *osvdbXmlCompareWrapperEmpty.sh*, *osvdbXmlCompareWrapper.sh* and *runCompareVdbXml.jar* can be found in section A.2.3.

Default usage

1. Check the SVN repository for the directory where the history file will be stored and make sure that it contains a sub directory for the sentinel files and another sub directory for the log files.
2. Check that *runOsvdbCreateHistoryIncremental.sh* and *osvdbCreateHistoryIncremental.sh* are locally available.
3. Check the variables in *runOsvdbCreateHistoryIncremental.sh* and *osvdbCreateHistoryIncremental.sh* to match the SVN repository, the work directories and for correct runtime variables.
4. Check the variables of *osvdbXmlCompareWrapper.sh* and *osvdbXmlCompareWrapperEmpty.sh* to match the used data file type (*osvdb*) as well as that the called jar file will have enough memory (at least 20 times the size of the file that will be processed). Allocating too little memory will cause

the comparison to fail while too much could cause swapping or failure due to not enough memory.

5. Check that the changes of *osvdbXmlCompareWrapper.sh*, *osvdbXmlCompareWrapperEmpty.sh* and *runCompareVdbXml.jar* are committed.

6. Run *runOsvdbCreateHistoryIncremental.sh*. It is recommended to forward the output to a file and run the script in the background since it can take from several hours to several days.

It is possible to run *osvdbCreateHistoryIncremental.sh* directly but in general it is recommended to use *runOsvdbCreateHistoryIncremental.sh* since it will also commit the log to the repository. To get the difference between two files one can either run *runCompareVdbXml.jar* directly or use *osvdbXmlCompareWrapper.sh* with the required arguments. The necessary arguments are describes in *osvdbXmlCompareWrapper.sh* or in the source code of the jar file.

A.3 History Evaluation

The history is evaluated by scripts which perform different tasks depending on their arguments. To perform many different evaluations in sequence the *operations* tool was developed. First, the *operations* tool is described, followed by the individual evaluation scripts that perform the evaluations.

A.3.1 The Operations Tool

This tool was initially developed to perform many executions of the *default evaluation script*. It greatly simplifies the update of the evaluations when a newer version of the history is available. Later, the *operations* tool was improved to perform some preprocessing and adapted to also execute other evaluation scripts. All the necessary files can be found on the SVN repository in the directory *svn//scripts/history-evaluate/*.

- The *nvd-evaluate.sh* and *osvdb-evaluate.sh* scripts are the *operation script* files. The *operation script* reads the *operations configuration* file, performs the preprocessing and executes the operations defined in the *operations* file. Each database has its own script because some parameters are different. It is much more convenient to define these parameters once per source and later just run the script than giving them every time as arguments. First, the script reads the parameters given in the configurations file. Then it creates the work and output directories if necessary and if the work directory was already present it will be cleaned. If specified in the *operations configuration* file, bad new lines in the history are fixed. These can happen when new lines within the content are not escaped. It checks if all the lines start either with *add*, *remove* or *change*; and if it is not the case the line will be added to the previous line. Last, the parameters of one operation are read, the selection performed, the filter performed and the evaluation script started. This is repeated until all operations are executed.

- The *nvd.config* and *osvdb.config* are the *operations configuration* files that contain parameters which are needed for all the operations such as the history source directory, the work directory or the output directory. Each parameter is expected in a specific line. The detailed definition of the parameters is in the *read me* file. It is possible to create other configuration files than the given ones if it is needed.

- The *nvd.ops* and *osvdb.ops* are the *operations* files which contain the parameters for each operation executed by the *operation script*. The operation was originally specified for the *default evaluation script*. The order of the parameters, therefore, might look a bit chaotic when used for other evaluation scripts. The detailed definition of the parameters can be found in the *read me* file. For each operation all the parameters must be correctly specified (currently thirteen) otherwise the result is wrong or the script will not run. It is possible to create other operation files if it is necessary. This can be handy if only a few operations have be performed and not all of them.

- The *read me* file holds the definitions of the parameters of the configuration and operations file. It also lists the arguments an evaluation script will be handed

Default Usage

1. Check that the parameters in the operations script point to the correct configuration and operations file.

2. Check that the parameters in the configurations file are correct.

3. Check that the operations in the operations file are correct.

Create a new operation

1. Choose the appropriate evaluation script and choose the parameters that will be given to the evaluation script.

2. Choose the file name of the output file.

3. Choose if only a specific set of history entries should be evaluated (select parameters).

4. Choose if a specific set of history entries should be excluded (filter parameters).

5. Fill the operation according to the specification in the *read me* file. Every parameter has to have a value. In case a parameter is not used it is recommended to use "-" as default value.

A.3.2 Evaluation Scripts

These scripts perform the actual evaluation of the history. Some of the scripts are designed to be used in junction with the operations tool. If that is the case it is noted in the comment at the beginning of the script. The other scripts cannot be run by the operations tool because they either are just a helper script (e.g., *script-getID.pl*) or they perform a very specific check which does not have to be run every time. In such a case it is much easier to write a stand alone script because otherwise the requirements of he operations tool have to be fulfilled which demands quite some extra code. All the scripts described in this section can be found on the SVN repository in the directory *svn//scripts/history-evaluate/*.

Creating a new evaluation script

All the history evaluation scripts can be executed directly but if the operations tool has to be able to run them, they have to fulfill the following requirements.

- The script must handle a specific set of arguments. The up to date version can be found in the *read me* file. Currently the script will be handed the following arguments

  1. The directory where the history files can be found.
  2. A text file where each line contains the file name of one history files that has to be processed.
  3. The output file path which is build by concatenating the output directory of the configurations file with output file name of the operation (operation parameter number two).
  4. The operation parameter number three.
  5. The operation parameter number four.
  6. The operation parameter number five.
  7. The operation parameter number ten.
  8. The operation parameter number eleven.

- The script has to be in the directory specified in the configuration file.

The default evaluation script (*script-eval-count.pl*)

This script is the main script used to analyze the history. Initially, it was only designed to count the amount of changes for each date but over time it was extended to perform many different counts. The format the result is returned, can be greatly influenced by the arguments. With the different arguments and the *select* and *filter* options of the operations tool it is possible to create a multitude of different counts to answer questions like “How many new entries were added per month?” or “On how many days were the entries modified?”. Since there are many possible combinations giving an example for each would be too much, therefore each argument is explained in detail:

- The 1st argument (argument 0 in Perl) is the directory of the history files.
- The 2nd argument (argument 1 in Perl) is a text file where each line contains the file name of one history file.
- The 3rd argument (argument 2 in Perl) is the path to the output file.
- The 4th argument (argument 3 in Perl) is the primary counter. It can have the values *id* or *date*. This determines according to which history element the output will be created. If *id* is chosen the counts are written to the output file per identifier (i.e., NVD or OSVDB identifier). This can be used to answer questions like "How many times was the entry edited?". If *date* is chosen the counts are written to the output file per date. This can be used to answer questions like "How many edits happened during each day?".
- The 5th argument (argument 4 in Perl) is the secondary counter. It has the value not chosen in the primary counter. The possible values are *date* or *id*.
- The 6th argument (argument 5 in Perl) is a date type. The value of this argument is only used if the value of the primary counter is *date*. Depending on this value the results can be aggregated over to specific time windows.
  - The value *date* leaves the date in the complete format (e.g., 2011-12-21).
  - The value *year* aggregates the results per year by reducing the date to the year (e.g., 2011).
  - The value *month* aggregates the results per month by removing the day from the date (e.g., 2011-12).
  - The value *day* aggregates the results per day in the month by reducing the date to the day (e.g., 21). Therefore, it can have values from 1 to 31.
  - *weekday* aggregates the results per week day. The weekday is determined as returned as number from 1 to 7 where 1 is Monday an 7 is Sunday.
  - *weekdayMonth* aggregates the results per week day in the month by determining the week number and the week day (e.g., week-2-day-3). The example means that the day is the second Wednesday in the month. This can be used to detect certain weekly work patterns within a month (e.g., first Monday in the month or last Friday).
  - *total* aggregates the results over the whole time by setting the date of all counts to *total*. With this counts over the whole observation period can be created.

- The 7th argument (argument 6 in Perl) is a value that specifies how many of the counts with the largest number of edits are removed. This is sometimes necessary because otherwise a few days with an extreme amount of edits (e.g., NVD Linux kernel renaming) distort the results. The counts of these days are removed before the dates are aggregated.

- The 8th argument (argument 7 in Perl) specifies the output format. If the value is *human* the output is more comprehensible for a human reader an it also adds more information per count. For any other value than *human* the output will be written in one line. It is recommended to use the value *nonHuman* in the operations file to keep it better readable. The *human* output is thought to be read by a human and not for further processing since the results are written in multiple lines. The *nonHuman* output it thought to be used in further processing such as sorting or MATLAB.

The resulting output greatly depends on the arguments, and therefore can vary. But the general format is the following:

1. the value of primary counter which is either the identifier of the entry or the date,

2. the number of edits for this *primary counter* (counter in human readable format),

3. the number of adds for this *primary counter*,

4. the number of changes for*primary counter* and

5. the number of removes for this *primary counter*.

If the parameter *human* is given to argument eight additionally all the involved *secondary counter* and edited XML tags are given.

Default Usage

1. Choose whether *id* or *date* is the appropriate primary counter and assign the other to the secondary counter.

2. In case *date* is the primary counter choose the appropriate time window. The possible values are defined in the *read me* file.

3. Choose how many of the largest edit counts have to be removed. Three means that the entries of the three largest edit counts are removed. Use zero in case none should be removed.

4. Choose the output format *human* or *nonHuman*.

5. Choose an output file name.

Dump script (*script-dump-all-results.sh*)

This script simply dumps the content of the given history files to a temporary file. The temporary file is sorted and the result is written to the output file. This script can be used to create one large history file of partial history files like the NVD history. It can also be used to create partial history files of entries that are of special interest by using the *select* and *filter* options of the operations tool. Such a partial history can then be analyzed manually or by a script later. The entries are sorted by the following paramters: 1. the date of the edit, 2. the identifier of the entry which was edited, 3. edit type, 4. the tag of the edited XML element.

Default Usage

1. Choose an output file name.
2. In junction with the operations tool set the not required arguments to a default value.
3. Write the parameters to the operation or run the script.

It is possible to run this script without the operations tool but in general it is recommended to use it since it can perform preprocessing. The necessary arguments are described in the script file.

Dump script with result sorted by ID (*script-dump-all-results-sortby-ID.sh*)

This script is a minor variation of the dump script *script-dump-all-results.sh*. The resulting history dump is primarily sorted by the identifier and then by the date. This can be helpful when the history of a set of previously selected identifiers has to be analyzed because the history of each identifier is together in one segment.

Duration after the entry is added until edits occur (*script-editDuration.sh*)

This script is very similar to *script-nvd-cvss.sh*. It calculates the days after the entry was added until the first and the last edit occurs. Multiple history files are combine into one. Finally, edit durations are calculated by *script-editDuration.pl*.

Default Usage

1. Check that the paths to the Perl helper scripts are correct.
2. Check that the patterns defined in the script satisfy the task.
3. Choose a file name to which the *script-editDuration.pl* helper script will write the results to. In the operation this is the third parameter.

4. Choose the *select* parameters for the operation.
5. Choose the *filter* paramters for the operation.
6. In junction with the operations tool set the not required arguments to a default value.
7. Write the parameters to the operation or run *script-editDuration.sh* with the necessary arguments.

The script is designed to be used in with the operations tool but it can be run on its own, as long as the required arguments are given correctly as specified in the script.

Helper script to calculate the edit durations (*script-editDuration.pl*)

This script is a helper script for *script-editDuration.sh*. It pareses the given history and calculates the number of days between the date the entry was added and the first as well as the last edit. The result is written to the specified output file.

NVD CVSS count and dump (*script-nvd-cvss.sh*)

This script dumps all the history entries related to CVSS edits and calculates the days between the changes by giving the dump to *script-nvd-cvss.pl*. First, a temporary dump is created of all the history entries related to CVSS edits. In a second step an identifier list is created by using *script-getID.pl*. Finally, the durations between the edits are calculated by *script-nvd-cvss.pl*

Default Usage

1. Check that the paths to the Perl helper scripts are correct.
2. Check that the patterns defined in the script satisfy the task.
3. Choose a file name for the dump. In the operation this is the third parameter.
4. Choose a file name to which the *script-nvd-cvss.pl* helper script will write the results to. In the operation this is the fourth parameter.
5. In junction with the operations tool check that no select or filtering regarding CVSS data is specified since this is done by the script.
6. In junction with the operations tool set the not required arguments to a default value.
7. Write the parameters to the operation or run *script-nvd-cvss.sh* with the necessary arguments.

The script is designed to be used in with the operations tool but it can be run on its own, as long as the required arguments are given correctly as specified in the script.

Helper script to calculate the CVSS edit time (*script-nvd-cvss.pl*)

This script is a helper script for *script-nvd-cvss.sh*. It parses the given CVSS history dump and calculates the number of days between the date the entry was added and the CVSS information was added, the number of days until the CVSS information edited the first time and the number of days until the CVSS information was edited last.

Helper script to get the identifiers (*script-getID.pl*)

This script can be used to get a unique list of all the identifiers contained in the given history file. This unique list is written in lexicographical order to the output file. It is possible to run the script on its own as long as the required arguments are given as described in the script.

Stand alone script (*script-nvd-check-lastmodifieddate.sh*)

This script can be used to check if the sum of edited entries over all the days matches the sum of *vuln:last-modified-datetime*. It creates several result files for manual investigation. In a first step these sums are calculated. To do this a result file of a previously performed history analysis file is evaluated. In a second step all the history entries without a corresponding edit in the *vuln:last-modified-datetime* are dumped for manual investigation. In the last step several count evaluations are performed on the dump file.

Default usage

1. Create the required history result as described below and check that the file path is correct in the script.
2. Create the required history dump only including edits in the *vuln:last-modified-datetime* element as described below and check that the file path is correct in the script.
3. Create the required complete history dump as described below and check that the file path is correct in the script.
4. Check that the file path variable to the default evaluation script is correct.
5. Check if the set output directory variable in the scrip is correct.
6. Check if the set output files name variables in the script are correct.
7. Run *script-nvd-check-lastmodifieddate.sh*.

To create the required history result, which excludes added entries, the *operations tool* has to run an *operation* with the following parameters: 1. script-eval-count.pl, 2. output file name, 3. date, 4. id, 5. date, 6. noSelect, 7. - 8. filter, 9. ^add.*tag='entry' 10. 0, 11. human.

To create the required history dump, which only including of the *vuln:last-modified-datetime* element, the *operations tool* has to run an *operation* with the following parameters: 1. script-dump-all-results.sh, 2. output file name, 3. - 4. - 5. - 6. select 7. ^change.*tag='vuln:last-modified-datetime', 8. noFilter, 9. -, 10. -, 11. -.

To create the required complete history dump, the *operations tool* has to run an *operation* with the following parameters: 1. script-dump-all-results.sh, 2. output file name, 3. - 4. - 5. - 6. noSelect 7. -, 8. noFilter, 9. -, 10. -, 11. -.

Stand alone script (*script-osvdb-addRemoveAdd.sh*)

This script can be used to check which OSVDB entries are added removed and added again. To create the identifier list the script uses *script-getID.pl* as a helper script. The result written to the output fill states for every removed entry how many times it was added and how many times it was removed.

Default usage

1. Check if the variables in the script are correct.
2. Run *script-osvdb-addRemoveAdd.sh*.

Appendix B

Assignment Description and Declaration of Originality

Declaration of Originality

The declaration of originality contains a blue stripe on the left side which was caused by the scanner. Even after repetitive scans the problem remained.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Master Thesis

Edit History of the NVD and Similar Vulnerability Databases

Mathias Karlsson

Advisor: Dr. Stephan Neuhaus, stephan.neuhaus@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

16 February 2012 – 15 August 2012
SVN Revision 9351 of 2012-02-17 17:24:52 +0100

1 Introduction

This master thesis is an empirical work providing the tools for the analysis of the structure and processes behind the National Vulnerability Database (NVD)[1] and similar vulnerability databases such as the Open-Source Vulnerability Database (OSVDB)[2]. Ideally, the thesis will also yield first insights into these structures and processes.

The Common Vulnerabilities and Exposures database (CVE)[3] is the largest clearing house for software-related vulnerabilities, and the NVD is a publicly available view of that database. This database is often used as ground truth for empirical work on software vulnerabilities, but since it is a human effort, it is clear that the data must necessarily be noisy. However, since the process of NVD entry creation and maintenance is not public, assessing just how noisy the data actually is and therefore how much one can trust information in the NVD, is unclear. This master thesis will shed some light on this.

The main vehicle to do that is by looking at the database's edit history. Naively, one would assume that an entry, once made, will remain unchanged; however, some entries do in fact change. Sometimes it is the product enumeration (CPE[4]) that is changing, but sometimes, the severity assessment (CVSS[5]) changes too, and even for very old entries! Sometimes it seems as if edits are made, then undone, and then redone again. If there were evidence of edit wars in the NVD, this would severely undermine its use as a research tool, since it cannot be relied upon for ground truth.

Of course, it would be easiest if one could simply ask MITRE or NIST for information on how NVD entries get created, but these organisations are quite tight-lipped about their processes. In fact, some entries in the NVD change only their "last changed" date, suggesting that there are hidden fields in the NVD that people at MITRE or NIST can see, but that are hidden from the public. Therefore, we will have to treat the NVD itself as an object of study and see what we can learn.

[1] http://www.nist.gov
[2] http://osvdb.org
[3] http://cve.mitre.org
[4] http://nvd.nist.gov/cpe.cfm
[5] http://nvd.nist.gov/cvss.cfm

2 Assignment

2.1 Objectives

This thesis will provide tools to analyse the edit history of the NVD and at least one other large vulnerability database (such as the OSVDB). These tools will enable the analysis of changes in these databases in time, in the large and in the small:

- Analysis of the evolution of an individual entry: what changed, how did it change, and when? Are there periodic changes? Are there times of large changes? When entries are newly added to the NVD, does the `published-datetime` field reflect the actual date when the entry was first visible?
- Finding of entries with particularly many edits: are there edit wars (and how would they be characterised)? Are there things that much-changed entries have in common?
- Analysis if changes in the large: how does the churn develop over time? Are there periodic phenomena? Are there times of large (bulk) changes?
- When comparing the NVD with the OSVDB, are there entries that appear first in one database and then in the other? How can these entries be characterised?

At the moment, there already exist daily downloads of the NVD in an SVN repository for about seven months, and very simple scripts to enable a very crude analysis of these downloads. These scripts may be used as a starting point.

2.2 Tasks

There are some common tasks that are orthogonal to the other, more technical tasks outlined below:

Validation. All programs and tools that are written a part of this thesis must be run against the NVD or OSVDB and must perform their assigned tasks. Some of these tools are later to be run unassisted as cron jobs, so they need sufficient error detection and handling so as not to overwrite important information or to yield incorrect results.

Evaluation. Both the NVD and the OSVDB are large databases, and looking at the edit history of these databases will produce more data still. All programs or scripts should therefore perform their tasks reasonably fast, and algorithms should be selected that have good running times.

Documentation. All programs and scripts will be used long after the thesis is finished, and perhaps even once all people immediately connected with it have left ETH. Therefore, they need to be documented so that people unconnected with their development can use them.

2.2.1 Familiarisation With the NVD History as Stored in the SVN

As mentioned above, there is already a large dataset of daily changes, in an SVN, going back to July 2011. This dataset needs to be processed so as to enable the analyses outlined in Section 2.1. Therefore, familiarisation is important:

- Can changes be extracted reliably by line-oriented tools such as diff?
- At the moment, NVD feeds are preprocessed by a crude sorting program before being committed to the SVN. Is that sorting sufficient to remove superficial but not substantial changes in the NVD XML feed stemming from NIST's XML export?

2.2.2 Extraction of Changes Between Successive Days

We need mthods to extract changes between two successive versions of the NVD and the OSVDB, and these changes must be displayable in the same format. Such changes must allow the identification of the entity that is changed, what has changed, and from what to what.

2.2.3 Characterisation of Changes

We als need methods to assess changes, both qualitatively as quantitatively, and both for the NVD and the OSVDB. For example: are there edit wars? How often are severities reassessed? Are there trends? Are there periodic activities (workdays versus weekends, year-ends, holidays)? Are there entries that get changed more often than average?

2.2.4 Storage of a Versioned OSVDB

In order to look at changes between successive versions of the OSVDB, and to compare the changes in the OSVDB with those in the NVD, we need historical data for the OSVDB. The OSVDB is distributed in mySQL dumps, which is not really good for computing differences, so a way has to be found to transform the mySQL dumps into something more suitable for analysis (perhaps via the roundabout way of importing the dump into a mySQL database first).

2.2.5 Characterisation of Inter-Database Differences

We need tools to compare the differences between the databases, for example, to ask questions like: Is one consistently ahead of the other? Are there categories where one is ahead, and other categories where the other is ahead with a change? (E.g., it could be that open-source vulnerabilities are reported faster in OSVDB, but closed-source may be faster in NVD.) Are changes consistent?

2.2.6 Writeup of the thesis

If the results are good enough, i.e., if there emerge interesting patterns and a compelling interpretation, an academic paper is also planned.

3 Milestones

- Provide and maintain a project plan which identifies the milestones.
- Two intermediate presentations: Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report. The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps (Wednesdays, 1000). The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

References

[1] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, (New York, NY, USA), pp. 529–540, ACM, 2007.

[2] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *Database and Expert Systems Applications* (A. Hameurlain, S. Liddle, K.-D. Schewe, and X. Zhou, eds.), vol. 6860 of *Lecture Notes in Computer Science*, pp. 217–231, Springer Berlin / Heidelberg, 2011.

[3] S. Neuhaus and T. Zimmermann, "The beauty and the beast: vulnerabilities in red hat's packages," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, (Berkeley, CA, USA), pp. 30–30, USENIX Association, 2009.

[4] S. Neuhaus and T. Zimmermann, "Security trend analysis with cve topic models," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, (Washington, DC, USA), pp. 111–120, IEEE Computer Society, 2010.

[5] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes," in *Proceedings of the Third international conference on Engineering secure software and systems*, ESSoS'11, (Berlin, Heidelberg), pp. 195–208, Springer-Verlag, 2011.

[6] K. Zhang, R. Statman, and D. Shasha, "On the editing distance between unordered labeled trees," *Inf. Process. Lett.*, vol. 42, pp. 133–139, May 1992.

[7] K. Zhang, "Computing the editing distance between unordered labeled trees is np-complete," in *Proceedings of the third international conference on Young computer scientists*, ICYCS'93, (Beijing, China, China), pp. 641–644, Tsinghua University Press, 1993.