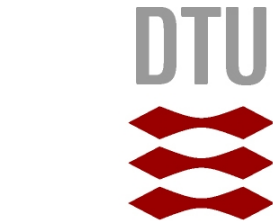


Danmarks Tekniske Universitet

UNIVERSIDAD DE OVIEDO

Gestión de la reputación de un sistema de software de código abierto basado en la fiabilidad de sus dependencias

Cristina GARCÍA GARCÍA

13-Mayo-2016

Universidad Técnica de Dinamarca
Departamento de Matemáticas Aplicadas e Informática
Christian Damsgaard Jensen, edificio 322,
2800 Kongens Lyngby, Dinamarca
Teléfono +45 4525 3351
Christian.Jensen@imm.dtu.dk
www.compute.dtu.dk

Escuela Politécnica de Ingeniería de Gijón
Departamento de Informática
Ángel Neira Álvarez, edificio 8,
33203 Gijón, Spain
Teléfono +34 985 18 24 81
neira@uniovi.es
www.di.uniovi.es

Índice general

1 Motivación

El uso de componentes externos está muy extendido en la industria del software. Algunos de ellos han sido desarrollados por proyectos de código abierto, *Open Source Software* (OSS), debido a grandes ventajas como los bajos costes. Los programadores pueden participar en los proyectos que más les interesen, lo que suele resultar en componentes de mayor calidad.

El principio fundamental de los proyectos de código abierto, la libertad de los desarrolladores, está sin embargo asociado con el riesgo. Aunque la calidad de este tipo de proyectos suele ser generalmente buena, sucesos como la vulnerabilidad *Heartbleed* en Open SSL enfatizan la importancia de asesorar y gestionar los componentes desarrollados externamente. Estos componentes pueden, a su vez, estar basados en otros proyectos, lo que resulta en un grado de incertidumbre sobre cuáles son los elementos necesarios. Las vulnerabilidades que afecten a cualquiera de estas dependencias pueden afectar la fiabilidad del proyecto y por tanto, para evitar brechas de seguridad, la calidad de estos componentes debe ser considerada.

El objetivo de este Trabajo Fin de Grado es investigar maneras de asesorar sobre la calidad de un proyecto OSS, considerando aspectos relacionados con la seguridad del mismo, para así gestionar la reputación de un sistema OSS. Para ello, se examinarán los componentes y frameworks desarrollados externamente por otros proyectos OSS. Este trabajo consiste en:

- Analizar diferentes perspectivas para asesorar sobre fiabilidad.
- Investigar formas de identificar las dependencias entre los componentes de un sistema OSS.
- Determinar métricas de seguridad que indiquen la fiabilidad de dichas contribuciones.
- Desarrollar y evaluar un prototipo de prueba de concepto.

2 Introducción

El término **software** se refiere a la colección de información e instrucciones de un ordenador. Una *librería de software* engloba los datos y el código de programación que pueden ser usados en diferentes programas de software. Su código está organizado de tal forma que pueda ser reutilizado por diversos programas. Cuando un programa invoca una librería, gana su funcionalidad sin que sea necesario implementarla.

Las librerías son un ejemplo muy conocido de reutilización de código, aunque no el único. Otros ejemplos son *frameworks*, *plugins* o componentes. Esta reutilización de código resulta en **dependencia**. Ryan Berg, jefe de seguridad en Sonatype, afirma que el 80 %-90 % del código de una aplicación pertenece a otras librerías o componentes. La aplicación no es sólo el código escrito por el programador, sino todas las dependencias que son necesarias, lo cual tiene un gran impacto en la seguridad del proyecto. Históricamente el foco se ha puesto en lo que el desarrollador ha escrito. Sin embargo, ahora esto es sólo una pequeña parte del resultado final. Si el proyecto tiene una dependencia que a su vez está basado en otro componente que tiene una vulnerabilidad, puede que esta afecte también al proyecto. Por tanto es fundamental considerar las dependencias y sus problemas al abordar cuestiones de seguridad.

Una vulnerabilidad "catastrófica", debido a su impacto, fue expuesta en 2014: Heartbleed [1]. OpenSSL es una librería de criptografía muy conocida y utilizada en otros proyectos OSS como Apache, uno de los servidores web más usados del mundo, o por compañías como Facebook o Google. Todos ellos se vieron afectados por este error que resultó en vulnerabilidades muy serias. Información confidencial como contraseñas de usuarios pudo haber sido obtenida de estas web "seguras" debido a Heartbleed. Acontecimientos como este muestran el gran impacto que una vulnerabilidad en una dependencia puede tener en los proyectos que dependen de ella. Por tanto, en este trabajo, la fiabilidad de los proyectos OSS se abordará en base a sus contribuciones.

3 Estado actual

3.1. Métricas

A pesar del gran desarrollo de la industria del software en estas últimas décadas, no existe ningún estándar para asesorar sobre su calidad. Una **métrica** proporciona información sobre alguna propiedad del sistema y por ello existen numerosas métricas dependiendo de qué propiedad del software se quiere cuantificar. Complejidad Ciclomática, número de líneas de código (SLOC), número de errores por línea de código, cobertura de código o tiempo medio entre fallos (MTFB) son algunos ejemplos de métricas que han sido utilizadas. Sin embargo, no resultan adecuadas para este proyecto. Dado el gran número de dependencias que un proyecto puede tener, se ha considerado como requisito deseable que las métricas sean fáciles de obtener para todos los proyectos y que un análisis fiable y objetivo pueda hacerse basándose en ellas.

Estudios como el realizado por Neuhaus y otros [2] muestran como vulnerabilidades pasadas pueden ser utilizadas para predecir qué componentes van a tener nuevos errores. Sin embargo, los autores centraron su estudio en el proyecto de Mozilla, y por tanto no es aplicable a otros. La información necesaria para este asesoramiento debe poder ser consultada en una base de datos general con información sobre cualquier tipo de proyecto. Para ello se puede usar ***Common Vulnerabilities and Exposures*** (CVE).

3.2. CVE

CVE [3] es un sistema para asignar nombres comunes a vulnerabilidades conocidas. Surge de la necesidad de crear un estándar para facilitar el intercambio de información sobre los errores encontrados.

Los identificadores o números de CVE son únicos y comunes e indican el año en el que la vulnerabilidad ha ocurrido así como cuatro dígitos, expandibles, que hacen referencia a la vulnerabilidad.

$$\text{CVE} + \text{AAAA} + \text{NNNN}$$

Sin embargo, saber el número de vulnerabilidades no es suficiente para un análisis fiable, ya que no todas ellas tienen el mismo impacto. Por este motivo surge CVSS: permite priorizar vulnerabilidades asignando diferentes puntuaciones.

3.3. CVSS

CVSS, ***Common Vulnerability Scoring System***, calcula tres tipos diferentes de métricas para reflejar la gravedad de una vulnerabilidad.

- **Métricas Base**: representan aquellas características que permanecen invariables. Está formada a su vez por métricas de *explotación* e *impacto*.
- **Métricas Temporales**: reflejan características que evolucionan con el tiempo.
- **Métricas de Entorno**: permiten adaptar las puntuaciones a la organización del usuario.

Las puntuaciones varían de 0 a 10, siendo 10 la más severa. El impacto de una vulnerabilidad se considera bajo si la puntuación se encuentra entre 0-3.9, medio entre 4-6.9, alto si varía de 7-8.9 y crítico de 9 a 10.

Tanto las vulnerabilidades pasadas (CVE) como su gravedad (CVSS) proporcionan información objetiva sobre la fiabilidad de un proyecto. Al ser estándares de extenso uso permiten una comparación fiable entre diferentes proyectos. Por ello, dichas métricas han sido seleccionadas para este trabajo.

4 Análisis

Los proyectos de Software de Código Abierto se desarrollan mediante colaboración pública y el código está disponible para su modificación. Este modelo de desarrollo único puede implicar un mayor riesgo al no tener control sobre los desarrolladores (su experiencia, talento...) al contrario de lo que ocurre en una compañía, lo que puede influir en la seguridad de un componente. Por otro lado, la calidad del software en sí tiene también gran importancia para determinar su fiabilidad: un software mal diseñado tiene muchos más bugs y vulnerabilidades en el código. Los diferentes enfoques están íntimamente relacionados: en general, buenos programadores construyen software de buena calidad, y viceversa.

Se puede abordar el asesoramiento de un componente de software externo de diferentes maneras:

- **Estudiando el historial de los desarrolladores de ese componente.** La experiencia y reputación de los programadores que trabajan en un proyecto puede resultar muy indicativa sobre su calidad final. Además, la organización del equipo así como la toma de decisiones pueden arrojar información adicional que ayude a discernir la calidad de los proyectos: si se trata de un equipo cerrado, con programadores con mucha reputación, se puede entrever que el componente final será de buena calidad. Sin embargo, la "reputación" de estos programadores debería estar basada en previos proyectos que hayan realizado, dependiendo de la calidad de los mismos. Por tanto, primero es necesaria una fórmula que distinga los proyectos fiables de aquellos que no lo son.

- **Examinando el código en busca de indicadores de su calidad.** Otra alternativa es analizar el código en busca de errores y vulnerabilidades que indiquen la calidad de un proyecto. Algunas herramientas, como *FindBugs*, cumplen con esta finalidad analizando el código en busca de errores. Sin embargo, no pueden encontrar todas las vulnerabilidades que puedan existir en el proyecto. Normalmente se utilizan para encontrar errores como inyección SQL o XSS (Cross-Site Scripting), por lo que pueden no ser muy fiables para predecir futuros comportamientos de los componentes.

- **Analizando el historial de los componentes individuales.** Este enfoque consiste en evaluar el riesgo de un proyecto OSS basándose en su comportamiento pasado. La evolución del número y gravedad de las vulnerabilidades puede indicar si un proyecto no se ha diseñado correctamente o si apenas está siendo revisado. Si el número de vulnerabilidades es muy bajo hay dos teorías plausibles: que el

componente es muy bueno y de ahí las pocas vulnerabilidades encontradas o que el proyecto no está siendo revisado, lo que sería muy preocupante ya que puede haber muchos errores que no hayan sido detectados. Para discernir cuál de las dos causas es más posible, el número de usuarios y colaboradores del proyecto puede resultar útil. En proyectos muy populares o en los cuales hay muchos programadores trabajando se puede asumir que la causa no es una falta de revisión. **Black Duck Open Hub** [1] proporciona información como el número de usuarios y de colaboradores, que puede servir para estimar la opción más probable.

[1] **Black Duck Open Hub:** https://www.openhub.net/

5 Diseño

A la hora de analizar las métricas empleadas, es importante tener en cuenta que:

- **Es normal que haya vulnerabilidades.** Puede ser más problemático que no haya vulnerabilidades notificadas, ya que la razón podría ser una falta de esfuerzo en la búsqueda y reparación de estos errores.
- **"Más vulnerabilidades" no siempre significa "menos seguro".** Un aumento del número de vulnerabilidades puede ser debido a un mayor esfuerzo de búsqueda de estos errores, por lo que no se puede asumir que la seguridad esté disminuyendo.

Estos principios van a tenerse en cuenta a la hora de analizar los resultados obtenidos para el número de CVE reportados por proyecto, así como los CVSS asociados.

5.1. CVE

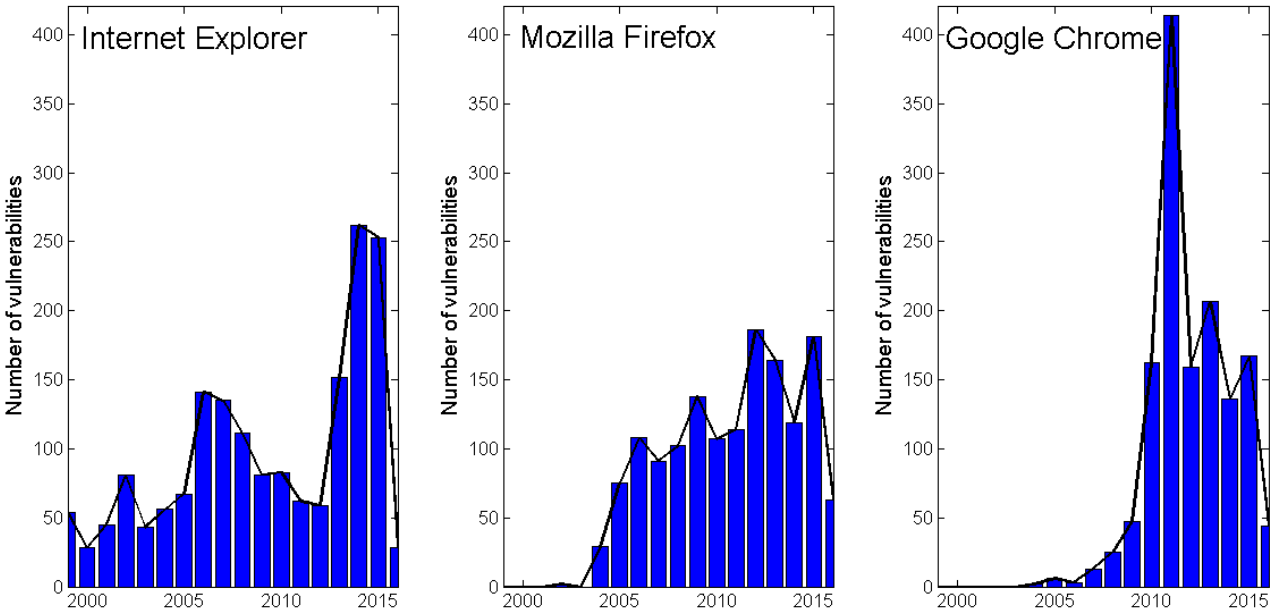


Figura 5.1: Evolución del número de vulnerabilidades para a) Internet Explorer b) Firefox Mozilla c) Google Chrome.

Comparar únicamente el número de CVE asociados al proyecto no sería suficiente. Como indica uno de los principios anteriormente mencionados: es normal que haya vulnerabilidades. Una librería que fuese muy popular, podría ser muy segura aunque hubiese tenido muchos CVE notificados ya que más personas han analizado el código. Sin embargo, la tendencia del número de vulnerabilidades si puede aportar información más fiable. La figura 5.1 ejemplifica esta situación ya que muestra la comparación entre tres navegadores web muy conocidos: Internet Explorer, Mozilla Firefox y Google Chrome.

Se puede observar que el número de vulnerabilidades encontradas tanto para Mozilla como para Chrome se incrementa en los primeros años. Esto se debe al aumento de la popularidad de los navegadores y no a una disminución de su seguridad, como se ha mencionado anteriormente.

Aunque en este caso Chrome es el que recopila un mayor número de vulnerabilidades en un año (2011), parece ser el más seguro ya que, cada año desde entonces, el número de vulnerabilidades ha ido decreciendo, lo que no ocurre en los otros dos casos. La evolución del número de CVEs para Firefox es más estable, mientras que para Internet Explorer, está creciendo.

5.2. CVSS

Aunque la evolución del número de vulnerabilidades encontradas resulta útil, la gravedad de las mismas debería ser también comparada. Siguiendo con el ejemplo anterior, se ha analizado los CVSS para los tres navegadores web.

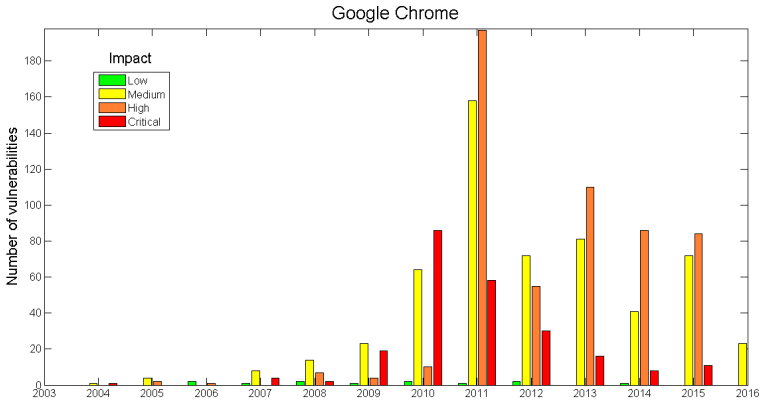


Figura 5.2: Número de vulnerabilidades por año de Google Chrome clasificados por su puntuación de CVSS.

La figura 5.2 muestra como no sólo el número de vulnerabilidades de Chrome está dismiyendo, sino también la gravedad de las mismas. Esto es especialmente llamativo para aquellas vulnerabilidades consideradas críticas (en rojo). Sin embargo, para el caso de

Firefox (figura 5.3), esta evolución no es tan clara. Además, aunque haya una tendencia a la baja de las vulnerabilidades críticas en los últimos años, la proporción entre estas y el número total es mucho mayor que en el caso anterior.

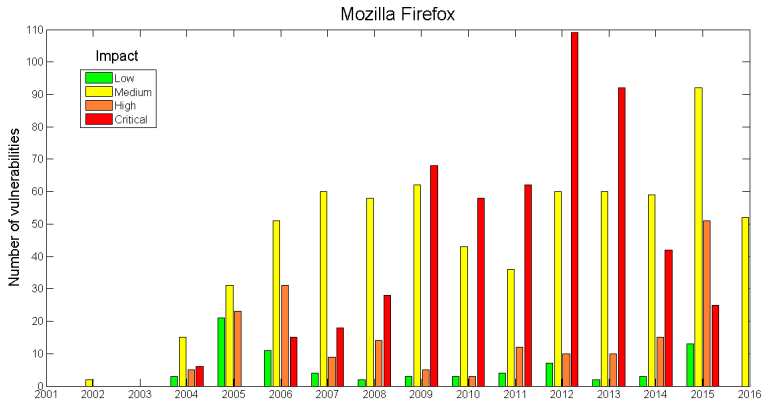


Figura 5.3: Número de vulnerabilidades por año de Mozilla Firefox clasificados por su puntuación de CVSS.

Aunque el caso de Internet Explorer es el más ilustrativo, figura 5.4, ya que en este caso la severidad de las vulnerabilidades ha aumentado con el tiempo. Este comportamiento es el que se debería evitar para una de las dependencias del proyecto.

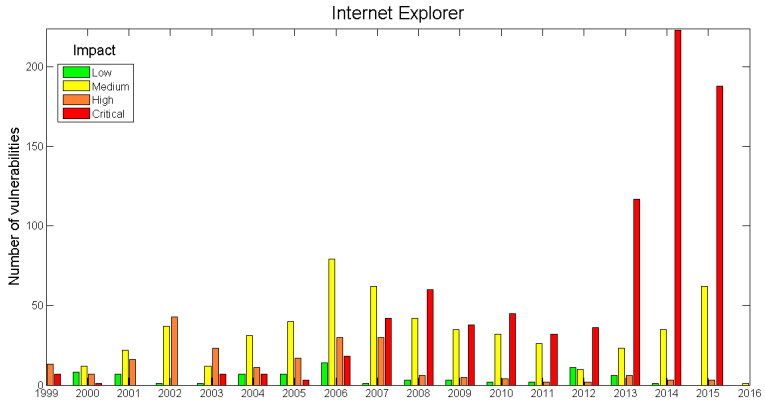


Figura 5.4: Number of vulnerabilities per year of Internet Explorer classified by their CVSS score.

5.3. Puntuación

A la hora de asignar una puntuación a los proyectos, se ha tenido en cuenta el análisis anterior. Por tanto, la puntuación se basará en la evolución durante los últimos años. Además, aquellas vulnerabilidades consideradas críticas y de alto riesgo recibirán un mayor peso en el resultado final, ya que el objetivo es destacar los problemas que se encuentren.

5.3.1. DependencyScore

Previamente a calcular la puntuación final, las dependencias se analizan individualmente, así como el proyecto: el resultado se denominará *dependency score* y refleja el **riesgo** de un componente en particular. A su vez, esta puntuación se conforma de otras dos: una asociada a la evolución de los CVE (*VulnerabilityScore*) y otra asociada a los CVSS (*SeverityScore*). El resultado final se calcula como:

$$DependencyScore = 0{,}2 \times VulnerabilityScore + 0{,}8 \times SeverityScore$$

- **VulnerabilityScore**: tiene en cuenta dos parámetros: el número de CVE notificados cada año, $NCVE$, y su tendencia, $TCVE$. Para asignar los diferentes valores, diferentes proyectos OSS se han analizado. En el caso de que $NCVE$ indique un número muy bajo de vulnerabilidades (entre 0 y 5 al año), el número de usuarios también se ha tenido en cuenta. Como se ha mencionado en 4, este valor indica cual es la causa más probable de este bajo número de vulnerabilidades: un buen código o una falta de revisión.
- **SeverityScore**: para esta puntuación, se ha tenido en cuenta la evolución de las vulnerabilidades según su severidad: críticas, de alto, medio o bajo riesgo, así como la proporción de vulnerabilidades críticas o de alto riesgo en comparación con el total. Estos valores van acompañados de diferentes pesos para destacar aquellos componentes con mayor riesgo.

5.3.2. FinalScore

Se han considerado diferentes enfoques para obtener una puntuación final a partir de los datos obtenidos para todos los componentes. Se ha decidido que, para encontrar aquellos elementos vulnerables que pueden suponer un riesgo para el proyecto, lo más lógico es que la puntuación final sea la mínima de todos los elementos.

6 Implementación y evaluación

Se ha escrito un pequeño programa en Perl que, al indicarle el nombre de un componente de software, obtiene sus dependencias, los CVE y CVSS asociados a ellas y la puntuación final. La herramienta *cve-search*, cuyo código fuente se puede consultar en https://github.com/cve-search/cve-search, se ha utilizado para recabar información sobre CVE y CVSS.

Black Duck Open Hub ha sido consultada en caso de duda para considerar a un proyecto fiable o no en función del número de usuarios y de contribudores. La tabla 6.1 muestra los resultados que se obtienen al analizar diferentes proyectos basándose en su número de vulnerabilidades y su gravedad, sin tener en cuenta las dependencias de cada uno. En algunos casos como Iceweasel, la puntuación varía enormemente. De ser considerado seguro al tener en cuenta únicamente el bajo número de CVE, a ser considerado de riesgo al constatar el bajo número de usuarios, ya que este dato parece indicar que no hay mucha actividad relacionada con el proyecto y por tanto se debería desconfiar de las pocas vulnerabilidades reportadas.

Proyecto OSS	**DependencyScore (sin usuarios)**	**DependencyScore (considerando usuarios)**
KeePass	10	0
tar	4.54	10
OpenSSL	3.32	3.32
Firebug	6.92	10
glibc	3.08	10
Apache	1.12	1.12
MySQL	4.48	4.48
Iceweasel	10	0
Firefox	0	0
Chrome	4.20	4.20
Explorer	0	0

Cuadro 6.1: Puntuaciones de *Dependency scores* obtenidas para diferentes proyectos OSS.

La tabla 6.2 muestra la puntuación final de diversos proyectos, esta vez teniendo en

cuenta sus dependencias. En algunos casos, como OpenSSL, su puntuación final se debe al propio proyecto, ya que sus dependencias se consideran seguras. Sin embargo, otros como MySQL muestran que existe algún problema con alguna de las dependencias del proyecto, pues su puntuación es menor a la que obtiene el proyecto por sí mismo.

Proyecto	**Final**	**ND**	**Media**	**ND (sin punt=0)**	**Media (sin punt=0)**	**Puntuación Mínima Dependencias**	**Puntuación Proyecto**
tar	10	8	10	0	10	10	10
glibc	10	4	10	0	10	10	10
KeePass	0	79	9.87	1	0	0	0
OpenSSL	3.32	17	10	0	10	10	3.32
MySQL	0	100	9.31	8	1.41	0	4.48
wireshark	5.32	126	10	0	10	10	5.32
Apache	0	131	9.4	9	1.25	0	1.12
Firefox	0	136	9.93	1	0	0	0
Chrome	0	151	9.89	2	1.66	0	4.20

Cuadro 6.2: Puntuaciones obtenidas para diferentes proyectos.

7 Conclusiones

El objetivo de este proyecto era investigar maneras de asesorar la fiabilidad de un proyecto open source teniendo en cuenta los componentes que utiliza: sus dependencias. Para ese asesoramiento, se han analizado diferentes métricas y enfoques y finalmente el número de vulnerabilidades, CVE, y su gravedad, CVSS, han sido seleccionadas.

La herramienta diseñada provee una puntuación *FinalScore*, que varía entre 0 y 10, basada en las puntuaciones *DependencyScore* obtenidas para cada dependencia y para el proyecto en sí.

Los diferentes pesos asignados han sido ajustados tras analizar y comparar diferentes proyectos OSS. Este trabajo ha demostrado la influencia que las dependencias tienen sobre la fiabilidad de un componente que las utiliza. Es posible detectar dependencias problemáticas antes de que aparezca una vulnerabilidad grave que pueda afectar a otros proyectos.

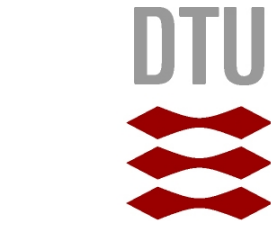
Es muy importante destacar la gran importancia que tiene entender cuáles pueden ser las partes vulnerables de un proyecto de software para evitar brechas de seguridad. Si las dependencias de un proyecto no se consideran, puede que aparezcan agujeros de seguridad que no sean detectados.

Bibliografía

[1] The heartbleed bug. http://heartbleed.com/.

[2] Neuhaus S and Zimmermann T et al. Predicting vulnerable software components. *Computer and communications security*, 2007.

[3] Debian Policy Manual. Declaring relationships between packages. https://www.debian.org/doc/debian-policy/ch-relationships.html.



Danmarks Tekniske Universitet

UNIVERSIDAD DE OVIEDO

Reputation management of an Open Source Software system based on the trustworthiness of its contributions

Cristina GARCÍA GARCÍA

13-May-2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Christian Damsgaard Jensen, building 322,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
Christian.Jensen@imm.dtu.dk
www.compute.dtu.dk

Polytechnic School of Engineering of Gijón
Information Techonlogy Department
Ángel Neira Álvarez, building 8,
33203 Gijón, Spain
Phone +34 985 18 24 81
neira@uniovi.es
www.di.uniovi.es

Contents

1 Abstract

Externally developed components and frameworks are used at large in the software industry. Some of these are developed by Open Source Software (OSS) projects, due to numerous advantages such as costs savings. Developers can contribute to the projects they like most and this usually translates into higher quality components.

The most important principle of Open Source, freedom, however, is associated with risk. While the quality of OSS projects is predominantly high, events such as the Heartbleed vulnerability in Open SSL emphasises the importance of quality assurance for externally developed components. These components and frameworks may themselves be based on the efforts of other software development projects, so there is a degree of uncertainty about the components required. Vulnerabilities in any of the dependencies may affect the trustworthiness of the component, hence the importance of assessing the quality of these components to avoid security flaws.

The purpose of this thesis is to investigate means to determine the quality, in particular with respect to security, of OSS projects. It deals with reputation management of an OSS system with the aim of providing advice on its quality. This is done by examining the externally developed components and frameworks developed by other OSS projects. The thesis consists of:

1. Study different approaches for trust management.
2. Investigate ways to identify the dependencies among components in the OSS system.
3. Define security metrics to measure the trustworthiness of the contributions.
4. Development and evaluation of a Proof-of-Concept prototype

1.1 Acknowledgments

This thesis would not have been possible without my supervisor, Dr. Christian Damsgaard Jensen (Technical University of Denmark). I am extremely grateful for all his always valuable remarks and suggestions.

I would also like to express my gratitude to my home university advisor, professor Ángel Neira Álvarez (Polytechnic School of Engineering of Gijón, Spain), for all his support during this year abroad.

2 Introduction

2.1 Background

Software is a generic term for the collections of computer data and instructions. A *software library* is a collection of data and programming code that may be used in several software programs. Its code is organized for the purpose of being reused by multiple programs. When a program invokes a library, it gains the functionality of that library without having to implement it itself. There are two types of libraries:

- **Static library:** the code of the library is resolved at compile-time and it is integrated directly into the code of the program. Originally, there were only static libraries. The main advantage is that the application can know whether all the libraries needed are present and that they are the correct version. This avoids dependency problems, known as **dependency hell**.
- **Shared library:** the code is referenced by programs at run-time and the reference is only made to the code that it uses in the shared library. The library code is only integrated into the program if any of the library functions are called. **Dynamic-link library**, or DLL, is Microsoft's implementation of this concept. Shared libraries reduce the amount of code included in the executable and allows the library files to be shared among many applications.

The operating system also provides several libraries that are used in most applications: *system libraries*. One well-known example is the C library that supports indispensable functions for a programmer such as basic input/output or file operations. The most famous system library in Linux is the GNU C Library (*glibc*).

Libraries are a well known example of code reuse, but not the only one. *Frameworks* are generally used by developers to reuse large pieces of software. *Components* can also be reused to save time and resources, which has lead to *Component-Based Development* (CBD). In CBD the system is structured as a collection of components. Another example of software reuse are high-level programming languages, such as C or Java [1]. They provide a level of abstraction that helps developers to be more efficient than writing the code with assembly languages. Another type of software are *plugins* and extensions, that extends the functionality of another piece of software.

This code reuse results in **dependency**. Ryan Berg, Chief Security Officer at Sonatype, states that between 80%-90% of the code of an application belongs to libraries

and other components. The application is not only the code written by the developer, but all the dependencies that are needed. This has a huge impact when talking about security. Traditionally, the focus was on what the developer has written. But nowadays, that is just a small fraction of the overall application. If your project has a dependency, which in turn depends on another component that has a vulnerability, it may affect also your project. It is therefore essential to understand this dependencies when addressing security issues.

2.2 Dependencies

In software engineering, the term *"dependency"* refers to packages that a program needs to work, without regard to which type of library it is, static of shared. It might happen that the project depends also on a specific version of the software. In those cases, the package will usually also work with any other version more recent than the one specified.

Almost all software projects involve working with dependencies. Often, these will themselves depend on other libraries or frameworks. This is known as **transitive dependency** and the recursive pattern of transitive dependencies will result in a tree of dependencies, as shown in figure 2.1 for the software library **OpenSSL**. The black lines represent the *depends* relations and the blue lines, the *pre-depends*. The different types of dependencies are explained in section 5.3.1.2.

This reuse of code is of greater utility in order to save time and expense, and also contributes to build higher quality projects, as developers do not have to elaborate everything themselves. For example, if encryption of the messages is required for an application, instead of doing it herself, the developer will use an existing package that will do the encryption. However, if not properly managed, lots of problems may arise.

2.2.1 Dependency hell

"Dependency hell" may appear because of incompatible version requirements for some packages' dependencies. It can take many forms and occur for many reasons and some of the most common types of dependency hell are:

- **Many dependencies.** If a project depends on many libraries, they require not only large amounts of disk space, but also a way to locate all the dependencies - this can be avoid by having a repository.
- **Long chains of dependencies.** This problem arises when a package depends on another one, and this one, in turn, depends on another package or library, and so forth. If the dependencies have to be resolved manually, conflicts between versions or circular dependencies may appear. For this reason, package managers are used.

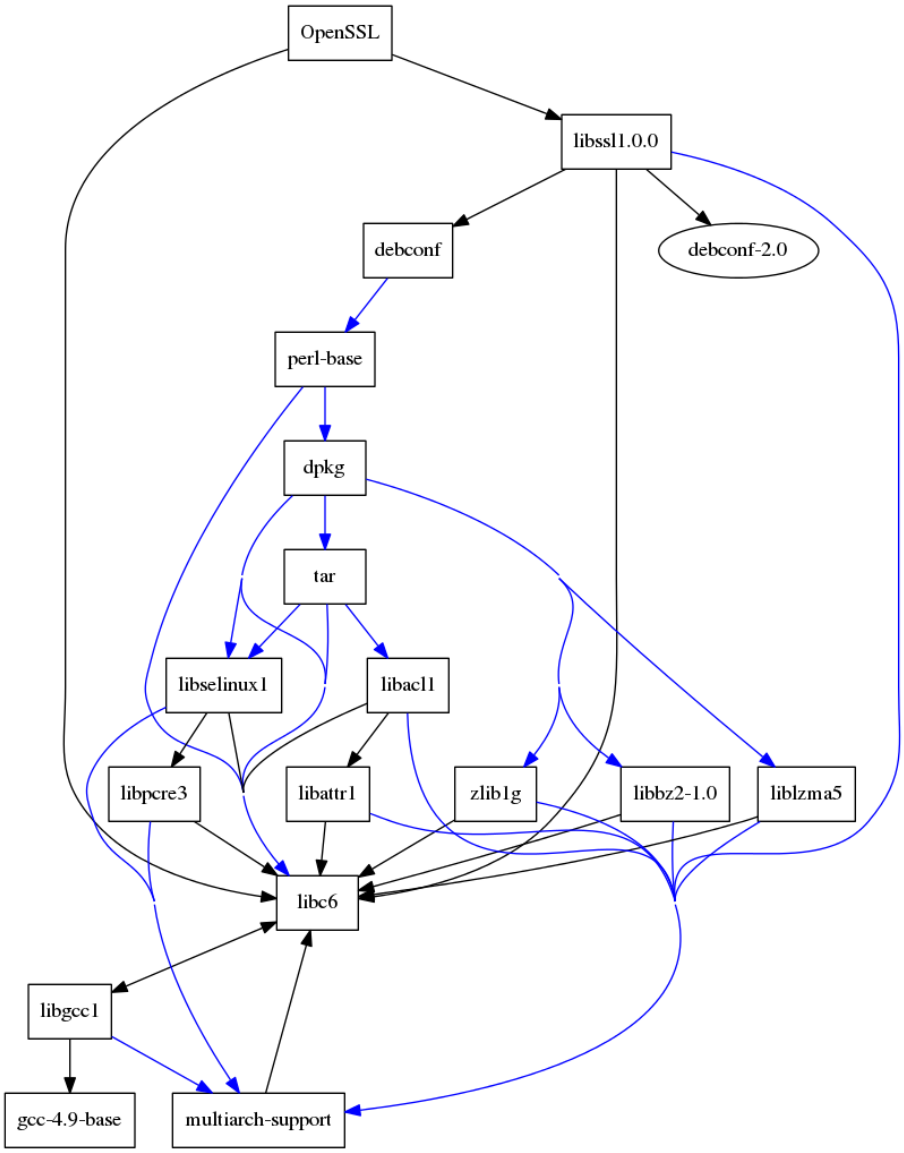


Figure 2.1: Dependency tree of OpenSSL.

- **Conflicting dependencies.** This take places when two different versions of the same package are required, but cannot be simultaneously installed.
- **Circular dependencies.** Occurs when a package x depends on another package y, which in turn depends on package z that depends on a previous version of the original package x. Therefore, the user must install all packages simultaneously.

Several solutions have been provided to deal with "Dependency hell":

- **Smart package management.** Many Linux distributions have repository-based

package management systems to automatically resolve dependencies searching in software repositories. This eliminate dependency hell for the packages that are in those repositories.

- **Version numbering.** A standardised numbering system is used: packages use a specific number, or **major version**, for each version and a subnumber, **minor version**, for each revision: 1.2 (version 1, revision 2). The version only changes when programs using the actual one will not be compatible anymore, while if still can work with it, the revision is used.
- **Private DLLS.** It was Microsoft's answer to DLL (Dynamic-Link Library) hell. This arises when applications needed different versions of shared libraries, so when one installed a different version that the existing one, another application would stop working. Microsoft uses version-specific information or an empty *.local* file to enforce the version of the DLL that is used by the application. This allows shadowing of library versions by specific programs.
- **Portable applications.** Application that is self-contained. It does not require any package to be installed: it is coded to have all necessary components included or to keep them in its own directory.

Dependencies are part of the code of an application and therefore it is highly important to understand what are they and which are the problems that may arise in order to assess the trustworthiness of a project.

2.3 Trustworthiness

The concept of *trust* is not a standard in the software industry. Depending on the research, different attributes are considered to evaluate it, such as reliability, safety, maintainability or usability. [2], [3], [4] all consider different scopes of trust and propose different frameworks.

Many of the studies related to trustworthiness have focused on security as the most relevant attribute, due to the enormous impact a vulnerability can have, putting very sensitive data at risk. Nowadays, every sector relies on software, from public health to banking, defense or telecommunications. Many governments and organizations are aware of this problem and the **trustworthiness** of a system is now a main focus on research.

As mentioned before, software projects often incorporates other components and frameworks for their own use. Due to the enormous impact of OSS some of these components may be open source. While generally this improves the quality and lower the cost of the projects, it can also lead to many problems if not properly managed. Security risks may result from this lack of scrutiny of the software dependencies.

In 2014, a "catastrophic" vulnerability, due to the impact it had, was disclosed: the Heartbleed bug [5]. OpenSSL is a very known cyptographic library, used for instance in the Apache OSS project, one of the web servers most used worldwide, or by companies like Facebook or Google. All of them were affected by this bug, which resulted in severe vulnerabilities. Therefore, sensitive data like users' password could have been obtained from this "secure" websites due to Heartbleed. This shows the huge effect that a vulnerability in one library can have for all the other projects that rely on it. Therefore, in this project, the trustworthiness of an OSS project will be addressed based on its contributions.

There are different approaches to asses the quality of an external software:

- By evaluation of the track record of the developers of that component
- Through examination of the provenance of individual components
- Analysing the code looking for indicators of the quality of the software

This approaches are intertwined: usually good developers produce high quality software, and vice versa.

A more thorough analysis of different metrics to quantify this quality of the code is explained in chapter 3.

2.4 Purpose

The goal of this work is to investigate ways to asses the trustworthiness of an open source project based on its dependencies. The purpose is to take into consideration the track record of the constituting components to quantify how trustworthy they are considered to be.

The majority of the research done has focused on the security of the system itself. This project aims to assess the trustworthiness from another approach.

Prior to the assessment, a research into different quality metrics is done to find the ones that are more relevant for evaluating the trustworthiness of a dependency. After, different approaches for trust management are analysed to better understand all the aspects that influence the security of a project.

2.5 Proposed solution

Several metrics have been analysed to determine the ones that are most relevant for the assessment. The purpose is to infer the behaviour of the code based on its track record, in order to be aware of untrustworthy software that may impact the security of

our system.

There are three metrics that have been selected: the number of dependencies of the project, the vulnerabilities related to the contributions and how severe they are. However, there may be special cases in which these metrics do not provide enough information to infer whether the component is trustworthy or not. In such situations, the number of users of the project is used to estimate which option is more likely.

2.5.1 Number of dependencies

"A chain is only as strong as its weakest link".

One main principle when talking about software is that no system can be ever proved to be absolutely secure. Many vulnerabilities may remain unknown and others may be introduced in future upgrades. These flaws may sometimes be used against the projects that depends on that software. Therefore, it is fair to state that the trustworthiness of a project will decrease with the number of dependencies.

2.5.2 CVE

One way to evaluate the pedigree of a software system is to look at the number of past vulnerabilities that the project has had.

The CVE (Common Vulnerabilities and Exposures) is a standard for known security issues. Each vulnerability receives a CVE Identifier. Therefore, if a large number of CVEs are assigned to a dependency, it could mean that the project is less trustworthy compared to another with fewer past vulnerabilities.

It may happen that one project has no CVEs reported or that the number is very low, which could mean that the software is very well design and there are truly no vulnerabilities or that they remain undiscovered because no one has revised thoroughly the source code. To estimate which scenario is more likely, the **number of users** will be evaluated.

2.5.3 CVSS

However, as not all vulnerabilities have the same impact, the CVSS (Common Vulnerability Scoring System) standard is commonly used to assign severity scores to vulnerabilities, based on several metrics. The scores range from 0 to 10, where 10 is the most severe.

A more detailed explanation about these standards, their importance and how they are going to be used is given in chapter 3.

2.6 Evaluation

Metrics aforementioned have been used in a Proof-of-Concept prototype in order to assess the trustworthiness of a given project. It obtains the dependencies of an open source project and evaluates the trustworthiness of each of them. Finally, an overall rating is proposed.

2.7 Project plan

This thesis deals with the reputation management of an OSS system with the aim of advice of its quality, by examining the externally developed components and frameworks developed by other OSS projects.

It has been structured as following:

1. Study different approaches for trust management.
2. Investigate ways to identify the dependencies among components in the OSS system.
3. Define security metrics to measure the trustworthiness of the contributions.
4. Development and evaluation of a Proof-of-Concept prototype.

Chapter 3 reviews the literature and what metrics have been proposed to evaluate the quality and trustworthiness of a component. Chapter 4 addresses the different approaches for trust management. Chapter 5 analyses the metrics selected for evaluating the trustworthiness of a dependency and how they impact the final score. Finally, chapter 6 evaluates and compares the tool for different OSS projects.

3 State of the art

3.1 Quality of software

Although software engineering has experience high growth during the last decades, there is still no standardized metrics to assess the quality of it. Metrics are of critical importance, as Lord Kelvin stated: *"If you can't measure it you can't improve it"*.

A **software metric** provides information about some properties of a system. There are numerous software metrics depending on which property of a software system is analyzed. Some examples regarding complexity, reliability, availability, security or maintainability are:

- Cyclomatic Complexity
- Source Line of Code
- Bugs per line of code
- Code coverage
- Mean Time Between Failures

3.1.1 Cyclomatic Complexity

A lot of research has been done in measuring the complexity of a software. One well-known metric was developed by McCabe [6] and it is known as *Cyclomatic Complexity*. It aims to indicate the complexity of a software based on the number of linearly independents paths through the source code. For instance, if there was only one single *IF* statement in the code, the complexity would be 2, as there would be two paths through the program's source code: one if the statement is *True* and another one if it is *False*. Some of the advantages of this metric are:

- Easy to compute and apply.
- It gives the minimum number of test cases required, which can be useful for the testing process.
- There are tools available for many different programming languages.

However, a program will be considered complex regarding only the number of decisions to be made, without taking into considerations its size or the program's data. Furthermore, all the conditional structures are weighted the same, irrespective of their complexity. Therefore, the value obtained may be misleading because there are a lot of simple comparisons and decision structures.

More information about other complexity metrics can be found in [7].

3.1.2 Source Line of Code

Source Line of Code (SLOC) is generally used to estimate the programming productivity or maintainability of the software. It can be an indicator of the amount of effort that will be required to develop a program.

The main advantage is that is very easy to compute. However, it also has some drawbacks:

- It can be ineffective when comparing programs written in different languages, as some require many more lines of codes to perform the same task.
- It is not a good approach to measure the productivity of a developer. Skilled developers may be able to write complex functions with few lines whereas an inexperienced one would require more lengthy functions, but simpler, to perform the same task. Furthermore, it may also have the adverse effect of increasing the complexity of the code, as developers will be incentive to expand it.

3.1.3 Bugs per line code

A software bug is a mistake in a program that causes the system to fail or to behave in unintended ways. The number of bugs per line code may indicate the quality of the software.

But when considering this option, another question appears: how many bugs are too many? According to [8], the average is about 15-50 errors per KLOC (1000 lines of code), depending on the project size. The author states that it is possible to achieve zero defects, but at a high price, which may not be suitable for commercial software. For instance, this was achieved by NASA, but most projects cannot afford the cost of this testing.

3.1.4 Code coverage

The purpose is to measure what percentage of code has been tested by a *test suite*. If the program has been tested in more depth, the likelihood of containing bugs is lower. To measure it, one or several **coverage criteria** are used:

- **Basic coverage criteria:** to determine if each function, statement, branch or condition has been tested.

- **Modified condition/decision coverage:** it requires that each entry and exit point are invoked and that each decision takes every possible outcome.
- **Multiple condition coverage:** it is necessary that all combinations of conditions in each decision are tested.
- **Parameter value coverage:** if parameters are taken in a method, all the common values for those parameters should be tested.

This measurement can help developers to look for those parts of the code that are not usually accessed and confirm that the most important conditions have been tested.

3.1.5 Mean Time Between Failures and Reliability

Software *reliability* is the probability of the component working properly and it is measured in terms of **Mean Time Between Failures** (MTBF). A *failure* befalls when a system does not succeed in meeting the desired objectives.

MTBF is the average time between failures of a repairable system. Once the MTBF is known, it is possible to calculate the probability of a system to be working in normal conditions. It is commonly used to measure hardware reliability but it may also apply to software, if the failures considered are 'bugs'. Thus, the time in which the software is working properly, without any new bug found, can be estimated. Furthermore, the time required to fix the problems can be measured by the MTTR (Mean Time To Repair).

The reliability of a system is strongly associated with its trustworthiness, since it increases when the number of bugs found declines. However, it has been argued that the MTBF is not a good way to measure software mainly because there is no natural degradation as in hardware [9]. Software fails because of design problems and not because the system wears out.

3.2 Trustworthiness

In this project, *software trustworthiness* is the property to be measured. A system is said to be trustworthy if it performs as intended for a specific purpose without unwanted side-effects or exploitable vulnerabilities [3]. To be able to improve the trustworthiness of a system, first a way to measure it is needed.

However, measurement of software quality has been proved to be difficult to achieve. Different techniques and analysis have been proposed to address different trustworthy challenges. Based on them, many attributes have been considered to influence software trustworthiness, as functionality, security, reliability, usability, maintainability... Section 3.1 briefly explained some of these measurements and approaches for evaluating some

of these properties, but the lack of a standard makes very difficult to address software trustworthiness. As each organization has its own objectives, and metrics generally depends on them, this standardization task is difficult to accomplish.

The U.S. *National Institute of Standards and Technology* (NIST) proposes a framework to provide some quantification of software trustworthiness based on attributes (safety, security, reliability..) and claims [3]. Yang et al. [2] aim to assess trustworthiness based on different attributes. They have conducted a previous research to determine which attributes were considered in different works and it can be seen that, in many of these studies, the main attribute that was considered was **security**.

Most software have vulnerabilities and the major causes are: **complexity** and the lack of **motivation** to create more secure software. The latter is primarily due to economic motivations, but also because of what is called the "market for lemons": a metaphor [10] for a market with asymmetric information. In this scenario, a car dealer has good ("plums") and troublesome ("lemons") cars on sale. The former ones are worth \$3,000 and the others, \$1,000 . The vendors know which is which, but buyers do not. Therefore, buyers will not pay more than \$1,000 since they do not have as much information about the quality of the product as to know if they are buying a plum or a lemon. However, there are not good cars sold at that price, which results on only lemons to be offered. The same applies in the software market. Buyers have no reason to trust the vendors' claims which leads to less investment in security. This effect could be minimized with a good security metric, as it would provide buyers with the information they need to distinguish the plums from the lemons.

Numerous efforts to assess security have been done by governments and organizations, as the *Trusted Computer System Evaluation Criteria* [11], *Common Criteria for Information Techonology Security Evaluation* [12] or the *Systems Security Engineering Capability Maturity Model* [13]. Even if the scope has been narrowed from software quality to only consider security, it has been proved to be a difficult task and there is still no standard about this topic. For instance, there have been studies about how the software complexity can indicate the number of vulnerabilities in a project, but this complexity is also difficult to quantify.

There have been studies about how past vulnerabilities can be used to predict undiscovered ones. Neuhaus et al. [14] have developed a tool to map past vulnerabilities to components. The authors conducted a survey of the Mozilla vulnerability history and the tool was able to predict which components were vulnerable based on past incidents. They looked at correlations between the vulnerabilities and the function calls or imports, in other words, other software that is needed in order to perform a service. This approach is similar to the aim of this work. However, the main drawback is that the study has focused only in the Mozilla project. The incidents were collected from the database that is maintained for the project. The approach for this project, however, is

much broader: it is intended for all Open Source Projects. Thus, the metrics should be collected from a general database that holds information about all projects.

This can be achieved by the use of **Common Vulnerabilities and Exposures** (CVE).

3.3 CVE

CVE stands for "Common Vulnerabilities and Exposures" and is a system to provide common names for publicly known cyber security vulnerabilities [15].

Prior to the creation of this system security tools maintained their own databases with their own names for the vulnerabilities. Therefore, given two different databases it was difficult to determine whether the codes referred to the same vulnerability.

CVE was created in 1999 to deal with this problems by providing standardized identifiers. The MITRE Corporation maintains the CVE identifiers and is sponsored by US-CERT (United States Computer Emergency Readiness Team). The CVE is also used as the basis for new services. The NVD (U.S. National Vulnerability Database) provides enhanced information such as severity scores and impact ratings based on the CVE list.

Each CVE contains the standard identifier number along with status indicator, a description of the vulnerability and references.

3.3.1 How are the CVE assigned?

The CVE-ID reservation allows researches to include CVE-IDs in the initial public announcement of a vulnerability. This way, it is easier to track the vulnerabilities over time and it is ensured that a CVE-ID number is instantly available to all users. The process is as following:

1. There is a request for one CVE-ID number.
2. MITRE reserves and provides the CVE-ID to the requester.
3. The requester shares the CVE-ID with all the parties involved.
4. The requester makes the CVE-ID public and notifies MITRE
5. MITRE updates the CVE Web Site to provide the details about the CVE-ID.

In case the issue was never made public the CVE-ID will be deleted.

The CNAs or *CVE Numbering Authorities* are major software vendor, like Apple, Cisco or IBM. They function as intermediaries between a researcher and the affected vendor, without directly involving MITRE. For this purpose, MITRE provides a CNA with a pool of CVE-ID to be distributed to researches and vendors.

The CNAs are responsible for announcing the new CVE-ID, which allows MITRE to update the information in the Web site.

3.3.2 CVE-ID Syntax

CVE Identifiers (also referred to as "CVE names", "CVE numbers", "CVE-IDs" and "CVEs") are unique, common identifiers for publicly known cyber security vulnerabilities.

The original CVE-ID syntax includes:

CVE prefix + Year + 4 Arbitrary Digits

However, this syntax only supports a maximum of 9,999 unique identifiers per year. With the increase of vulnerability reports, this was not sufficient and a new syntax was implemented in 2014. The fixed four digits can expand with arbitrary digits only when needed in a calendar year. Therefore no changes are needed to previously assigned identifiers. For instance, CVE-YYYY-NNNN and if needed, CVE-YYYY-NNNNN and so on.

However, knowing the number of vulnerabilities is not enough, as they are not the same. Some may allow an attacker to execute code on the server, others may expose very sensitive data whereas in other cases, the vulnerability may only have very limited effect on the system.

CVSS is used to prioritize vulnerabilities by scoring them using several metrics.

3.4 What is CVSS?

The Common Vulnerability Scoring System (CVSS) is a standard for assessing the severity of software vulnerabilities. It was the outcome of a research by the *National Infrastructure Advisory Council* (NIAC) in 2003 to design an universal standard to prioritize vulnerability and calculating its severity. It has been entrusted to the **Forum of Incident Response and Security Teams** (FIRST) since 2005. The current version of CVSS, version 3, was released in June 2015.

The CVSS base score has been adopted as a measurement of the severity of vulnerabilities by many organizations, including the *National Vulnerability Database* (NVD), the *Open Source Vulnerability Database* (OSVDB), the *CERT Coordination Center* or companies like Cisco.

The severity scores are calculated based on three types of metrics: base, temporal and environmental, each consisting of a set of metrics (figure 3.1).

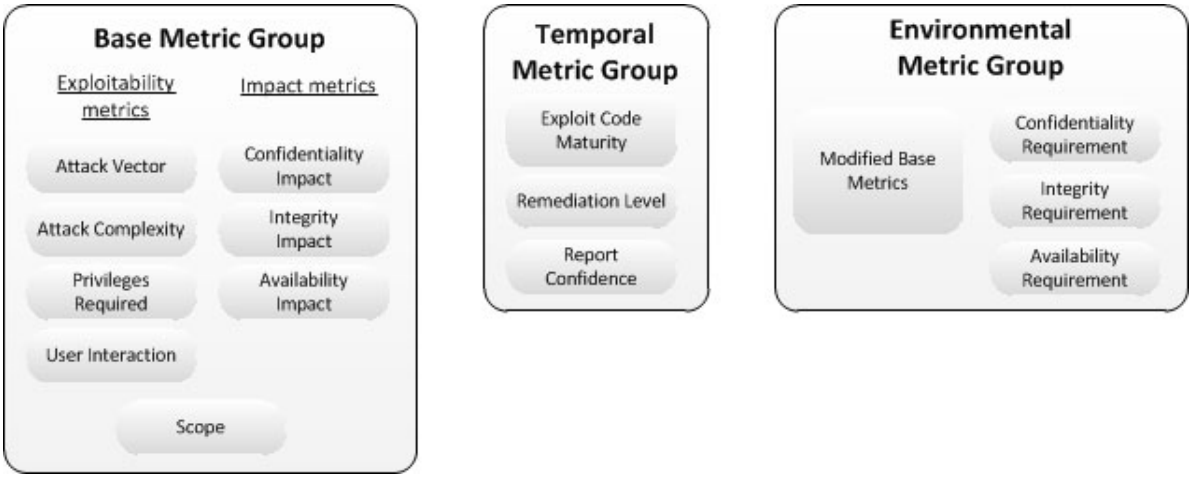


Figure 3.1: Metric Groups (image from [16])

- **Base Metrics** are intended to represent the characteristics of a vulnerability that remain unchanged. It is composed of **Exploitability metrics**, representing the vulnerable component, and **Impact metrics**, related to the consequences of the exploit.
- **Temporal Metrics** reflect the characteristics that may evolve over time but not across environments.
- **Environmental Metrics** provide a context for the vulnerability.

All the different metrics used to calculate the CVSS scores can be consulted in the appendices: A.1.

3.4.1 Base Metrics

The Base Score is itself composed of two: *Exploitability* and *Impact* sub scores. One important property considered in the third version of CVSS is the **Scope** of the vulnerability.

The **Exploitability Metrics** refer to the attributes of the *vulnerable component*. They reflect the properties of the vulnerability that lead to a successful attack, such as:

- How the vulnerability exploitation is possible: for instance, if he attacker needs to have physical access to the network or it is possible to exploit it remotely (AV)
- If there are special conditions which requires preparation in advance for the attack (AC)
- The level of privileges that an attacker must have before the attack (PR)
- If the attack requires a user to participate in order to compromise the vulnerable component (UI)

The **Scope** reflects if the exploited vulnerability can affect resources beyond its privileges. The **Base Score** is then greater when a scope change has occured.

The **Impact Metrics** reflect the attributes of the impacted component. Depending on whether there has been a changed of scope, these metrics show the **Confidentiality**, **Integrity** and **Availability** impact either to the vulnerable component or the impacted one.

3.4.2 Temporal Metrics

The **Temporal Metrics** reflects the current state of the:

- **Exploit Code Maturity:** the vulnerability score increases with the development of the exploit code
- **Remedation Level:** it reflects whether there is a solution available and whter it is temporary or an official fix
- **Report Confidence:** it measures the degree of knowledge about the vulnerability.

3.4.3 Environmental Metrics

The **Environmental Metrics** are designed to customize the CVSS score depending on the importance of the affected component to the user's organization.

The **Modified Base Impact Metrics** modify the score by assigning different weights to Confidentiality, Integrity and Availability impact metrics. Depending on the organization, three requirements can be set to High, Medium or Low:

- **Confidentiality Requirement** (CR)
- **Integrity Requirement** (IR)
- **Availability Requirement** (AR)

These three requirements allow to modify the Base Metrics to adjust them to the user's environment.

3.4.4 How is CVSS calculated?

Scores range from 0 to 10, being 10 the most severe. A vulnerability is considered to have low impact if its base score ranges from 0 to 3.9, medium between 4 and 6.9, high from 7 to 8.9 and critical if greater than that.

The metrics introduced in the introduction are assigned different scores depending on the kind of vulnerability and its impact. It is important to highlight that the most used score and the severity ranges are based only in the **Base Metrics**. However, all the three scores can be consulted in the web page of the National Vulnerability Database [17].

The summary of all the different metrics used to calculate the CVSS scores can be consulted in the appendices: A.1. The formulas to calculate the 3 different scores, Base Score, Impact and Exploitability Subscores, are also explained.

3.5 Dependency graph

Another approach for vulnerabilities prediction is to use dependency graphs. Two examples of this approach can be seen at [18] and [19].

Neuhaus et al. [18] present a vulnerability prediction model and conduct an experiment on Firefox. The dependency graphs that they consider are based on the relationship among software elements, such as classes, functions or variables. A static code analyzer is used to gather this information and the vulnerability data is obtained from the *Vulnerability Database for Firefox*.

Nguyen and Tran [19] propose a framework and metrics to identify the most critical elements at early development stages. They based their evaluation on the dependencies and the flow of information among the components. The authors argue that these indicators can reduce the maintenance cost as they allow developers to make early decisions about the component. Finally the propose a framework for security evaluation but without an empirical evaluation.

Both studies demonstrate the relation between the track record of the components and the trustworthiness of the project itself. They also highlight the importance of considering all the elements that conform the software. In the former, past vulnerabilities have been used as a metric to successfully predict which components were vulnerable on a particular case. In the latter, the analysis has been focused on the behaviour of the internal components. The indicators obtained from the dependencies resulted in early detection of problems and allowed them to identify the most problematic components.

3.6 Summary

"Trustworthiness" is a broad concept and there are numerous parameters that can provide information about it. However, some of them may be misleading: for instance, the SLOC. In order to be a good indicator, other metrics such as the skill of the programmer or the language used should be considered along with the SLOC. But then, the metric is no longer easy to compute, which was its main advantage.

Other properties may be difficult to measure, as the complexity of the code, whereas some metrics are not publicly available for all the projects, like the number of bugs per line. Other metrics, such as code coverage, may be very relevant for the developers of the project but not so important for external assessment.

To be able to assess about any OSS project, the metrics should be easy to compute, as dozens of dependencies may be analyzed for a single project. The information should also be publicly available for all OSS projects. For this reasons, the number of vulnerabilities reported (CVE) and their severity (CVSS) have been selected for the assessment. They are both wide used standards, which makes them especially convenient. They allow to make fair comparisons between very different projects while providing good indicators about the quality of the software.

4 Analysis

4.1 Trust management

Open Source Software (OSS) refers to software that is developed through public collaboration and that its source code is available for use or modification. This unique development model has some implications that should be discussed.

4.1.1 Approaches for trust management

For the evaluation of the trustworthiness of an OSS project, several approaches can be taken. For instance, the community-based model differs from other models employed for companies to develop their software. In the former case, all people may be able to contribute to the source code of the project, which implies more risk than if only people in the company can develop it. This impacts the trustworthiness of the project and should be taken into consideration. Yet an analysis of the quality, in terms of trustworthiness, of the source code of the project as well as all an investigation of the components that it depends on has a great significance. However, all these approaches are intertwined: developers' skills are judge by the quality of software that they produced, and if the pedigree of a project is good, it is mainly because good developers were involved in the development. All these approaches lead to a great number of different parameters that may be adopted for the assessment.

4.2 Team

When assessing about an open source project, the experience and reputation of the developers working on it is significant. Also, the organization of the team can influence the trustworthiness of the project: it is not the same that the development is open to everyone to contribute, than if only few well-known people can work on it.

At an earlier time, an open source project was **community managed** by definition. This concept of community-based development has proven to be useful as people can work on what they are better at and more interested in, and this generally results in high quality components [20]. However this 'self-organized' model also has its risks. The European Project RISCOSS grapples with risk management in OSS adoption, to provide

tools and methods for integrating community-based OSS development in companies.[21]

Under this principle of open development, for each project a different strategy can be used in order to:

- Explain to contributors how they should work in the project: what is expected from them as well as which protections are at their disposal.
- Describe the quality control processes.

The different strategies lead to different *governance models*.

The governance models state whether a project is open to participation. But it is also important to define the degree of openness for contribution, hence *contribution models* are used.

4.2.1 Contribution models

Eric Raymond, in [22], used the metaphor of **"The Cathedral and the Bazaar"** to contrast the open and closed source development approach.

- **Cathedral:** more similar to the traditional model. The source code is published with each software release, and a group of developers will work on debugging between these releases.
- **Bazaar:** Linux's model. It aims to maximize the number of people debugging the code by developing the code in view of the public.

Releases are made much more often in the Baazar view than in the Cathedral's model, mainly because the code is only available to few developers in the second case. It usually takes more than six months to release a new version, whereas in the former model, bugs are found very quickly because of the large amount of co-developers looking through the code in every release.

One of the main advantages of the Bazaar model is precisely these early releases. The mean time between finding a bug and fixing it is much lower than in the Cathedral model, where it may take several months before the new version with the fixes is available. However, the fact that the code is available for public scrutiny does not imply that this revision is done. For popular projects, the community is usually very active in reviewing the code. But for others, maybe not so many people is paying attention to it. This leads to the problem that this work tries to deal with: reviewing the source code of the project without considering the security issues of other components that are also used, may lead to vulnerability risks. Nevertheless, this open source approach allows the user is to review the code if she is concern about the security issues that may exist.

4.2.2 Governance models

A governance model describes how decisions are taken in the project and the rules for users to participate in it.

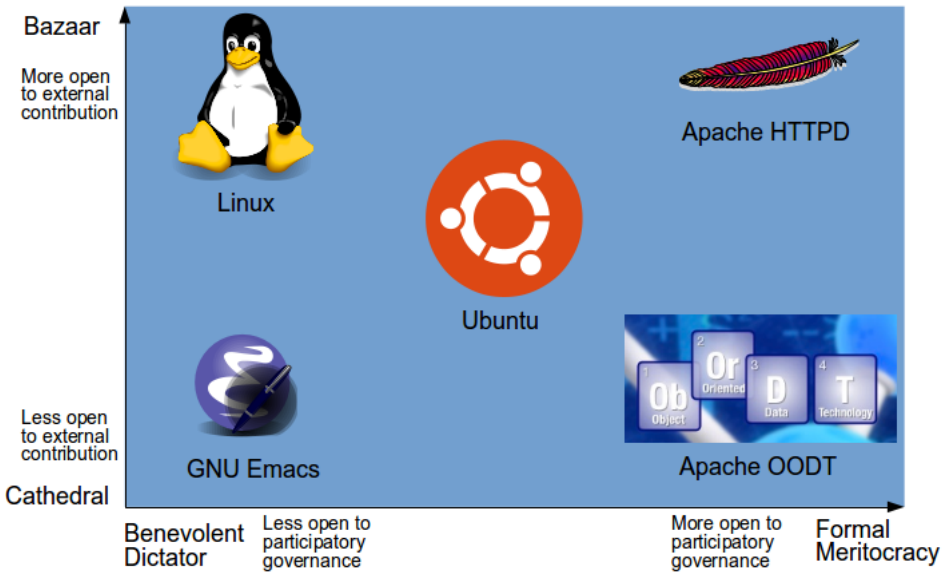


Figure 4.1: Contribution and Governance Models (image from [23]).

Governance models range from benevolent dictatorship to meritocracy, depending on the openness to participatory governance. The membership structure also depends on how the decision-making process is. Some examples of real Open Source Projects and which models they follow can be seen in figure 4.1. The two extremes that can be found are:

- **Benevolent dictatorship.** The project founders keep the control of the project and they are responsible of the final decisions.
- **Meritocracy.** The community is responsible for all decisions. However, contributors who have benefited more to the project have greater decision-making power. This governance model, in which people gain responsibility thought contribution, is called the *meritocratic model* and helps new individuals to engage with the project.

When evaluation the trustworthiness of a project, the governance models are closely related to it. For instance, it can be inferred beforehand that a project is less risky if it maintains tighter control over decision making process. Having more control of the people who are involved in decision-making also means that the risk is reduced, since it is assumed that these members' track records endorse them to undertake important decisions. However, as the community grows, they may need to take decisions in areas in which they are less experts, so this should also be taken into consideration.

4.2.3 Metrics

The evaluation of the trustworthiness of the dependencies could be calculated based in these models. Therefore, if its contribution model was like the "Bazaar" type, the trustworthiness of the project should be higher than if it was similar to the "Cathedral" style. In the former model, the code is available so many more developers can scrutinize it looking for vulnerabilities. Furthermore, releases are made very ofter, which is also positive from a trustworthiness point of view. Even if there is one vulnerability in the component, the probability that it is found and patched in a short period of time is higher than in the latter case, where new releases may take months to be available.

The governance model may have less impact on trustworthiness than the contribution one. It would seem that tighter control over the decisions is a bit more reliable, but in a meritocratic model, only people that have proved to benefit the project are in charge of the decision-making process, which also guarantees the quality.

But the trustworthiness could also be based on the skill of the programmers that developed the dependency. This "skill" could be assigned based on the quality of their previous projects and could be useful to predict how the current component would behave.

4.3 Code analysis

Analyzing the source code looking for bugs and vulnerabilities has been a major concern since the beginning of software. Chapter 3 reviewed some metrics that have been used for the evaluation, but many more are available depending on which aspects the developers want to measure.

Many tools have been developed to find security flaws. They are called *Source Code Analysis Tools* or *Static Analysis Tools*. They review the source code to detect security vulnerabilities. There is also another type: the dynamic tools, which require the code to be running. Some Open Source tools are [24]:

- **VisualCodeGrepper**: this tool scans and describes the issues found for different languages: C++, C#, VB, PHP and Java. The disadvantage: the list of vulnerabilities used for the scan cannot be modified.
- **YASCA**: it is an aggregated tool from many other popular static analysis tools and it analyzes, mainly, Java and C/C++. The main drawback is that it was design to deal mainly with SQL injections and XSS (Cross-Site Scripting), so other severe issues may not be found.
- **OWASP LAPSE+**: it is an Eclipse plugin that detects vulnerability in Java applications.

- **FindBugs**: it can be also installed as a plugin for Eclipse and it can find bugs as SQL injection or XSS, among others.
- **Flawfinder**: a tool to analyze security issues in C, sorted by risk level. The disadvantage: the number of false positives.
- **PMD**: developed for Java, it scan the source code looking for code problems, which do not need to be directly related to security issues.
- **RATS**: Rough Auditing Tool for Security. It can scan C, C++, Perl, PHP and Python languages and it uses text-based pattern matching to look for security issues. It is a very quick and efficient tool and it can be used for numerous languages.

These tools could be used in the project to find its vulnerabilities and give a score based on them. However, it is not enough to consider only that component, because vulnerabilities in any of the dependencies may affect the trustworthiness of that software project. A better approach would be to use one of these tools on all the dependencies and obtain their trustworthiness based on the number of bugs found. The results could be combined with the number of lines to find out the *number of bugs per line code* and compare them to normal values, as explained in section 3.1.3, to estimate the quality of the different dependencies.

4.4 Track record of the dependencies

The other approach that can be taken to evaluate the risk of an OSS project is based on the track record of the components of the software, to predict which components are vulnerable. In contrast with the previous section, there has not been so much research in this area.

The quality of the dependencies could be based on the problems that it has faced in the past. The evolution of the past vulnerabilities could give an insight about the trustworthiness of the software. For instance, if the design of the project was good, the number of security flaws founded will decrease for the last years. The trend suggests that the component is improving over time and thus, only few vulnerabilities remain undiscovered. However, if the project leads to more serious problems each year, one cannot be sure about how many bugs could be still existing in the code.

Conversely, the lack of errors could be due to a poor revision of the code. The principle of OSS is to make the code publicly available so that "many eyes" can look for bugs, but this does not assure that the revision is actually done. So there should be a way to draw a distinction between those projects that do not have vulnerabilities because they are very good and those which are not being revised.

But, is there any way to estimate if the project is being reviewed? One may think of looking at the *number of downloads* of the project. It is likely that those which are very popular among users have drawn the attention of the community, and this number of downloads could be closely related to it. However, this number is not available for all the projects and it could also be misleading: for instance, a user may download one software but never really use it. Or it could obtain the software because someone has given it to her via USB or CD and hence this would not be reflected in the metric. It is also a problem for companies, as they can only know how many people have a license for the software but not how many people are actually using it.

Black Duck Open Hub (formerly *Ohloh*) [1] could be another source of information. The purpose of Open HUB is to compare OSS projects based on their popularity and the activity related to them among the development community. Therefore, some information provided by this site may be useful to estimate the activity of the community for OSS projects. For instance, figure 4.2 shows the information retrieved from OpenHub for *Firefox*.

Figure 4.2: Firefox information from OpenHub.

The number of *users on Open Hub* or the number of *current contributors* could be used as metrics of the endorsement of the project. However, in some cases it is possible that no vulnerabilities have been reported and that the project has not many users, but this does not mean that the component is risky. It is also possible that the software project is not even in the Open Hub database. One example of this could be small libraries that big projects include: it may be possible that there is no vulnerability to be found, so this should be taken into consideration. For instance, *gsfonts* is a dependency of Firefox with no vulnerabilities reported and with zero users on Open Hub (figure 4.3). However, this does not mean that the library is untrustworthy: it may be, in fact, that there are no bugs in the code. A correlation between all the metrics could make it clearer whether a project should be considered trustworthy.

[1] **Black Duck Open Hub:** https://www.openhub.net/

Figure 4.3: gsfonts information from OpenHub.

4.5 Summary

Different approaches to assess about the security of an OSS project have been addressed. Both the contributors and the software components of a project impact in the trustworthiness of it.

It is undeniable that the team that develops a component has a great impact on its trustworthiness. Furthermore, the way in which the software has been organized could also influence the final result. However, these aspects are difficult to measure and need to be based on the quality of the software. A way to decide which projects are trustworthy and which ones are not has to be defined to be able to "score" the developers.

Analysing the source code looking for vulnerabilities could also serve as an indicator. However, the main drawback is that the tools used cannot find all the vulnerabilities in the project. Some of them could find bugs like SQL injections or XSS, but many others may remain undiscovered. It is a good practice to avoid common and well-known errors, but it is not so useful to predict how the software is going to behave in the future. Predictions about future vulnerabilities cannot be made based on the number of bugs found by these tools.

Finally, how to predict the trustworthiness of the project based on past information has been addressed. The main advantage of this approach compared to analysing the source code is that all the vulnerabilities are considered, whereas in the former case, some type of bugs may not be found. Also, the information that is considered for the analysis is available for any Open Source Project, which is highly important for a fair comparison. The past vulnerabilities can be found by using the CVE, as explained in 3.3.

Each vulnerability receives a CVE (Common Vulnerabilities and Exposures) Identifier in order to provide common names for known security issues. As not all vulnerabilities are the same, CVSS (Common Vulnerability Scoring System) standard is used to assign severity scores to vulnerabilities to assess their impact. The scores range from 0

to 10, being 10 the most severe.

If a large number of CVEs are assigned to a dependency, it could mean that the project is less trustworthy compared to another one with fewer past vulnerabilities. However, having a large number of them does not always imply that the project is less secure because more bugs are found [25]. It can also indicate, for instance, that the library or packet is very popular, and therefore a lot of people look through the code. This data can be correlated with the one obtained from the Open Hub website to decide which hypothesis is more probable in general.

5 Design

A tool has been developed to assess the trustworthiness of a given project. The aim of this work is to provide a score, which will range from 0 to 10, being 0 very untrustworthy and 10 the maximum score, based on information of its contributions. As security metrics are going to be used, it is important to understand how they work and what do they reflect.

5.1 Metrics

"Is trustworthiness of software measurable? The determination of trustworthiness of software is difficult. There may be different quantifiable representations of trustworthiness".

In this way the paper *Toward a Preliminary Framework for Assessing the Trustworthiness of Software* [3] starts. A "metric" is a system of related measures (compared against a standard) enabling quantification of some characteristic. When talking about security, the purpose is to quantify the degree of safety.

There is no standardized way to measure the security of software, even if many attempts have been done, as explained in chapter 3. For a metric to be considered good, it is **necessary** that they satisfy a **specific business requirement**. This leads to the different quantifiable representations abovementioned: different metrics are to be considered depending on the desired outcome, as the requirements for specifying them are usually drafted from the business needs.

5.1.1 Security metrics

Good metrics should be quantitative, objective, inexpensive, obtainable and repeatable, among other characteristics, and this applies also to security metrics [26].

When talking about security, it is scarcely possible not to mention vulnerabilities. A security **vulnerability** is a "weakness in a product that could allow an attacker to compromise the integrity, availability, or confidentiality of that product" [27]. A **bug** is a mistake that a developer can make when developing the software and that causes the system to fail, but that is not necessary a vulnerability. A fault is considered to be a vulnerability when it allows an attacker to abuse the system. However, vulnerabilities

are only dangerous when they have one or more **exploits**. Exploits are pieces of software, data or commands that take advantage of a vulnerability to change the normal behaviour of the system.

When analysing vulnerability data, some principles should be bear in mind [25]:

- **Having vulnerabilities is normal.**
  Therefore, it may be more problematic to not have vulnerabilities reported rather than the other way, as it could mean that there are no efforts being made in finding and fixing these bugs.
- **"More vulnerabilities" does not always mean "less secure"**
  An increase of the number of vulnerabilities may simple be due to an increase of the community for discovering them or that the recording practices have improved. Therefore it cannot be assumed that the security is declining.
- **Design-level flaws are not usually tracked**
  Most vulnerabilities reported are related to coding mistakes, whereas design vulnerabilities are common but not so tracked.
- **Security is negatively defined**
  The security of a system is defined according to what an attacker should **not** be able to do regarding Confidentiality, Integrity and Availability.

5.2 Metrics for the assessment

All this principles will be considered when analysing the metrics. The ones that influence this assessment are:

- The number of dependencies.
- The number of vulnerabilities reported (CVE). This metric could also be correlated with the number of users, based on the information provided by *OpenHub*, in those cases were the number of CVEs is very low.
- The severity of the mentioned vulnerabilities (CVSS)

Therefore, other metrics and approaches that have been discussed previously in chapter 3 are out of the scope of the project. However, the solution provided here could be expanded in future works by taking into consideration other relevant metrics.

5.3 Number of dependencies

The first parameter to take into consideration is the number of dependencies that the project has. First of all, a way to obtain the dependencies of the project is needed. For

the developed tool, Ubuntu and Debian distributions have been considered. In both of them the *apt* package manager can be used for this purpose.

5.3.1 Apt

The **A**dvanced **P**ackage **T**ool (APT) is a free software that handles the installation of packages. The user just needs to indicate the name of the software to install and *apt* will automatically install it and all its dependencies, which helps to avoid problems as dependency hell and eases the installation process for users.

Each package has meta-data declaring the file's dependencies. This meta-data is different depending on the package type. For *deb*, there are seven different control fields: Depends, Pre-Depends, Recommends, Suggests, Enhances, Breaks and Conflicts, while for *rpm* there are four: Provides, Requires, Conflicts and Obsoletes.

5.3.1.1 Apt-rdepends

Apt-rdepends is a tool that recursively check dependencies of a package until the entire dependency tree is spread out. It searches through the APT cache to find what packages a given one is dependent on, plus what packages these ones are also dependent on. It can be installed very easily by running the command:

sudo apt-get install apt-rdepends

5.3.1.2 Types of dependencies

Packages can have several relationships to others. In the case of Ubuntu and Debian, they both use *deb* packages while other distributions as Fedora or Red Hat works with *rpm* files.

The possible values for the dependency fields are, for *deb* packages: [28]

- **Depends:** this is an absolute dependency: the package needs it in order to be configured.
- **Pre-Depends:** this field is similar to *Depends*, but forces the installation of the dependency even before starting the installation of the desired software.
- **Recommends:** these packages have strong dependency with the one given, but not absolute: the package can still work without them but it would be unusual.
- **Suggests:** packages with dependency field as "Suggest" can be more useful and enhance the performance of the package, but they are not required for the proper functioning of it.
- **Enhances:** similar to "Suggest", but in this case the field is used to indicate packages that can improve the functionality of the given project.

- **Conflicts:** the packages cannot be installed in the system simultaneously.
- **Breaks:** the package cannot be unpacked unless the broken one is deconfigured first. The difference with *Conflicts* is that, in this case, both packages can be unpacked at the same time, but not configured.

These control fields, except for *Enhances* and *Breaks*, appear in the depending package's control file. *Enhances* is present in the recommending package's control file, and *Conflicts* in the version of depended-on package which causes the named package to break.

For *rpm* files: [29]

- **Provides:** libraries or services that the package provides.
- **Requires:** the dependencies of the given package (libraries or other packages that it requires on in order to run correctly). This is a strong dependency, but in addition there are four weak dependencies. These are used by dependency solvers but are not requirements for the package to run: *Recommends, Supplements, Suggests* and *Enhances*.
- **Conflicts:** this package cannot be installed if the other ones are.
- **Obsoletes:** packages that are superseded by the actual one (it is their update).

For this project, Ubuntu distribution has been used. By default, *apt-rdepends* only shows the *Depends* and *Pre-Depends* types, which are the required packages for the installation.

For instance, the dependencies of OpenSSL can be shown by simply running the command (figure 5.1):

apt-rdepends openssl

5.3.2 Assessment

One main principle when talking about software is that no system can be never proved to be absolutely secure. Many vulnerabilities may remain unknown and others may be introduced in future upgrades.

Another aspect to consider is the misuse of these contributions. One project may work properly but not under the conditions of our project. It may be difficult for the developers to understand perfectly the behaviour of all the projects they are using for their code, and therefore some misconfiguration errors can lead to vulnerabilities that were out of the scope of their dependencies.

```
cris@ubuntu:~$ apt-rdepends openssl
Reading package lists... Done
Building dependency tree
Reading state information... Done
openssl
  Depends: libc6 (>= 2.15)
  Depends: libssl1.0.0 (>= 1.0.1)
libc6
  Depends: libgcc1
libgcc1
  Depends: gcc-4.9-base (= 4.9-20140406-0ubuntu1)
  Depends: libc6 (>= 2.14)
  PreDepends: multiarch-support
gcc-4.9-base
multiarch-support
  Depends: libc6 (>= 2.3.6-2)
libssl1.0.0
  Depends: debconf (>= 0.5)
  Depends: debconf-2.0
  Depends: libc6 (>= 2.14)
  PreDepends: multiarch-support
debconf
  PreDepends: perl-base (>= 5.6.1-4)
perl-base
  PreDepends: dpkg (>= 1.14.20)
  PreDepends: libc6 (>= 2.14)
dpkg
  PreDepends: libbz2-1.0
  PreDepends: libc6 (>= 2.14)
  PreDepends: liblzma5 (>= 5.1.1alpha+20120614)
```

Figure 5.1: Dependencies of OpenSSL

Therefore, it is fair to state that the trustworthiness of a project will decrease with the number of dependencies. However, this parameter should not be looked at in isolation, as the quality of the contributions is not the same. Thus, more weight should be given to other parameters more relevant for security issues.

5.4 CVE

CVEs are going to be used to address the trustworthiness of the different contributions.

If only the number of CVE identifiers was considered, this would lead to some unfair comparisons. A package or library would be considered more trustworthy just because **less vulnerabilities were reported**. However, one possible explanation of having less CVEs could be that the software is less used. A popular library could be more secure than other even though more CVE numbers were assigned for it, just because more eyes are scrutinizing the code looking for bugs.

Therefore, another approach can be taken. CVE syntax allows separating the vulnerabilities issues by years. By taking into consideration the year along with the number of CVEs, the evolution of the project can be seen.

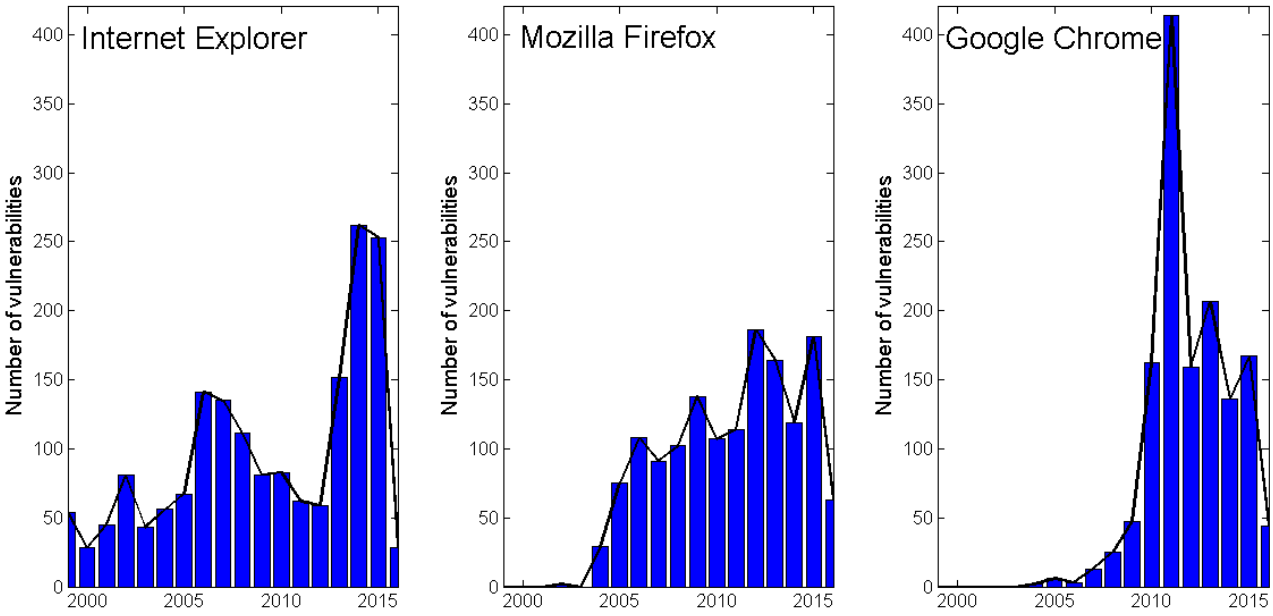


Figure 5.2: Evolution of the number of vulnerabilities reported for a) Internet Explorer b) Firefox Mozilla c) Google Chrome.

For instance, in figure 5.2 three of the most known web browsers have been compared. Only one of this projects is considered to be Open Source (Mozilla Firefox), [1] but this comparison can be useful to understand better how to analyze the CVEs.

Internet Explorer was started in the year 1994 whereas the CVE standard was created in 1999. Therefore, it can only be seen the evolution of the beginning of the project for Mozilla Firefox and Google Chrome. In those 2 cases, there is a increasing on the number of vulnerabilities in the first years. This is an example of what was above mentioned: the increasing number of vulnerabilities does not seem to be related to a decrease of the quality of the software but to an increase of its popularity. As more people start using the new browsers, more effort is put on analyzing the code searching for possible vulnerabilities.

Other of the premises was that the increase of number of vulnerabilities could be related to the improvement of the recordings. This may be one of the reasons for the

[1] Google Chrome is not an Open Source Project. However, it is based in Chromium, which it is.

increasing number of vulnerabilities in the first years for the Internet Explorer case. Also for the following browsers, the amount of vulnerabilities discovered per year are much higher than the ones for Internet Explorer in its beginning.

In this comparison, Google Chrome seems to be more trustworthy as is the only one that has a lowering tendency, even if it is the one that collect more vulnerabilities in a year (2011). Firefox has not any high peak like the former browser, but its tendency remains more constant through the years. The worst case is Internet Explorer since the number of vulnerabilities has been increasing considerably since 2012.

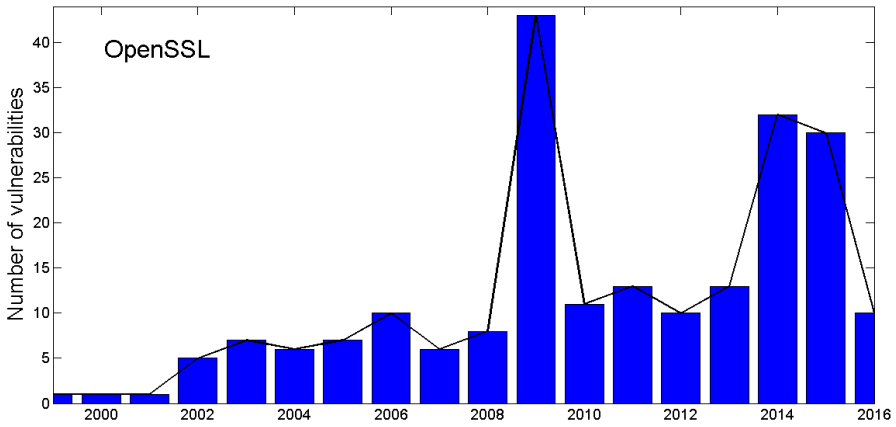


Figure 5.3: Evolution of the number of vulnerabilities reported for OpenSSL.

It may be a bit striking the huge number of vulnerabilities reported each year for all these projects, but when studying the graphs we should take into consideration that they are huge software projects. Therefore it is normal that the number of vulnerabilities found is large. When analysing it for other projects, as figure 5.3 shows for OpenSSL, the number of vulnerabilities decreases considerably. This is also interesting from a trustworthiness point of view. Large projects tend to be more vulnerable:

- Having more lines of code increases the probability of having bugs.
- It may be more tempting for attackers, as more users will be affected because of it.

This example was meant to illustrate that no fair conclusions can be made only based in the number of vulnerabilities. A better indicator is the trend.

5.5 CVSS

As it has been shown, the CVEs can serve as a basis for calculating the trustworthiness, but something else is needed. Once the the number of CVEs as well as the trends of vulnerabilities reported have been analysed, the severity of each of them should be considered.

Continuing with the same example above, Google Chrome, Mozilla Firefox and Internet Explorer have been analysed.

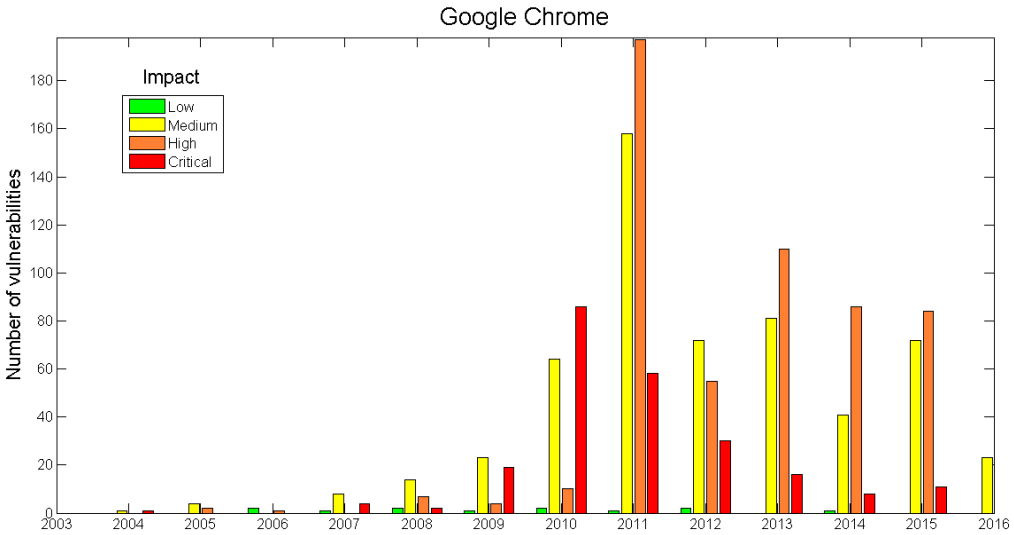


Figure 5.4: Number of vulnerabilities per year of Google Chrome classified by their CVSS score.

Figure 5.4 represents the number of vulnerabilities of Google Chrome per year. For each period of time, the vulnerabilities have been separated depending on their CVSS. Therefore, scores between 0-3.9 are considered to be of low impact (green); between 4.0-6.9, the impact is medium (yellow), high from 7.0 to 8.9 (orange) and finally critical in the range 9.0-10 (red).

It can be noticed that the severity of the vulnerabilities also follow the lowering tendency of figure 5.2, notably for the critical ones, which shows that the software is getting better through the years.

In this case, it does not seem that the decrease is due to less revision of the project, as a large number of vulnerabilities are still discovered every year. Therefore, it could be inferred that the reason is that Google Chrome is a good project in terms of security.

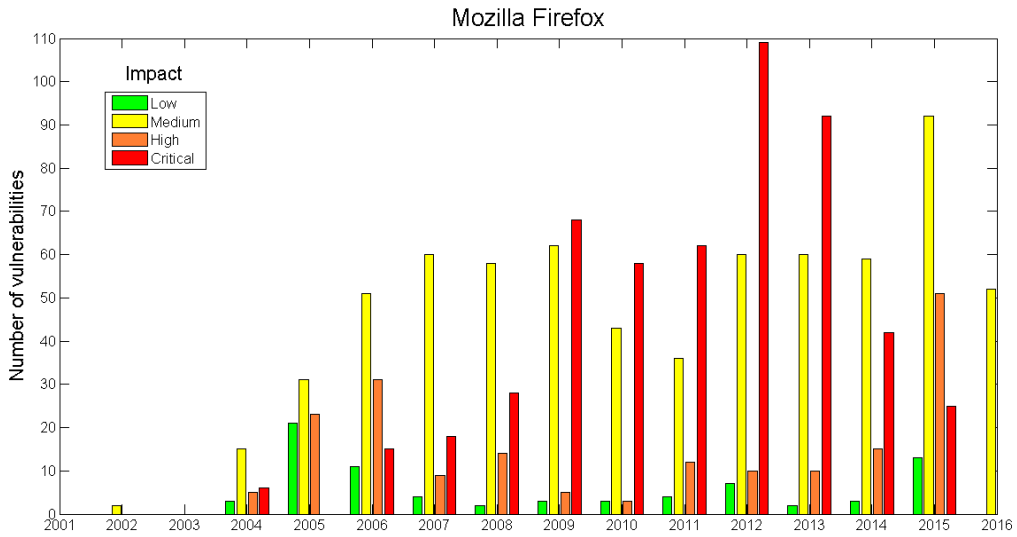


Figure 5.5: Number of vulnerabilities per year of Mozilla Firefox classified by their CVSS score.

When the number of CVE reported was compared in figure 5.2, Mozilla Firefox appeared to have a stabilized pattern. However, in figure 5.5 it is noticeable that the number of critical vulnerabilities has declined since year 2012. Even if the total number of CVE per year is roughly constant, the average impact is lower. This can be also a good sign, as the project is still being constantly inspected but the vulnerabilities found are not so severe.

Finally, Internet Explorer's case, figure 5.6. Unlike Chrome and Firefox, the severity of the vulnerabilities has not diminished, but an increase over the last years can be detected. This is exactly the behaviour to avoid in the dependencies of the project. There is a high probability that a lot of vulnerabilities can still be exploited and therefore, they could be used against the project.

5.6 Score

The final stage of this project is to define how the different metrics are combined to provide a score. As outlined before, several different metrics have to be considered: the number of dependencies, the number of vulnerabilities and its severity. In some cases, the *number of users* is used in deciding whether a project is very good design or it is not being revised.

Before going into details, it is important to recapitulate some major points:

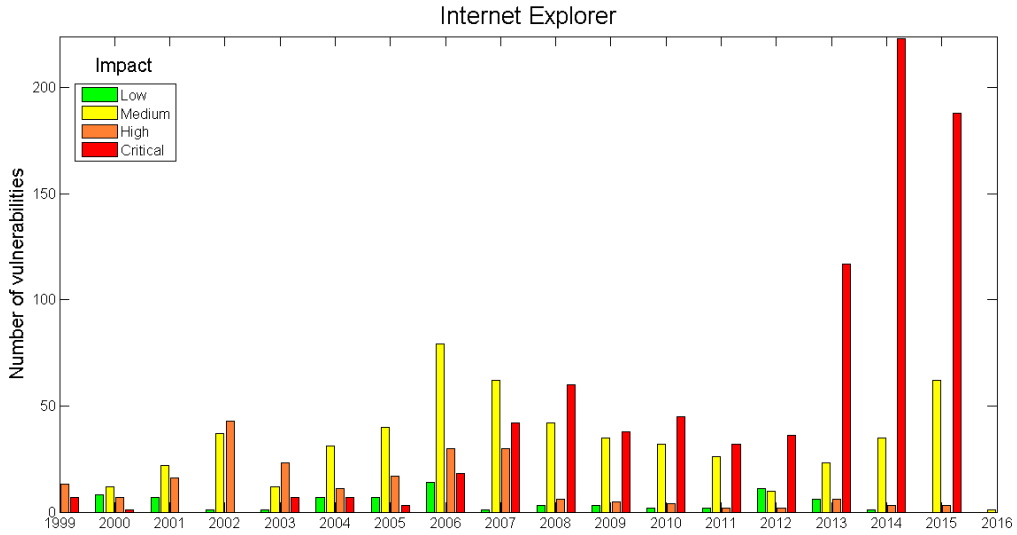


Figure 5.6: Number of vulnerabilities per year of Internet Explorer classified by their CVSS score.

- The aim of this study is to asses the trustworthiness of a project based on its contributions. A score is given to the project based on the quality of its dependencies as well as the quality of the project itself. This rating is based on the CVE and CVSS reported for all the components.
- As the severity of the vulnerabilities is being considered, those which are *critical* and *high* will have more weight for the final score. The reason for this is to highlight vulnerabilities that represent a higher risk for the project.

A function to assess the trustworthiness of a particular dependency should be defined. Before start, it is important to mention that no difference is made between dependencies and transitive dependencies: all of them are treated the same way. The metric obtained should also be aggregated with the ones from other dependencies to finally be able to infer the trustworthiness the whole project. For each dependency, as well as for the project itself, a score will be given based on the CVE and CVSS related to it. The final rating of the project will be based on these individual scores.

5.6.1 Dependency score

Prior to calculate a final score for trustworthiness, the dependencies are analysed individually as well as the project itself. The *dependency score* will be calculated for each of the components and it will reflect the **risk** of that particular element. It is based only in the CVEs and CVSS reported for it: whether it has or not dependencies itself will

not be considered here.

This score is composed of other two: the vulnerability and the severity sub scores.

$$DependencyScore = 0.2 \times VulnerabilityScore + 0.8 \times SeverityScore$$

The **Vulnerability** sub score is based on the CVEs reported and the **Severity** sub score, in the CVSS related to them. As the previous analysis on CVE and CVSS, this last one provides more reliable information, so its weight is bigger.

Note: these scores are not indicating the trustworthiness of the project but its risk. Opposite to the final score, for these ones a high rating means that the component is untrustworthy.

5.6.1.1 Vulnerability sub score

This sub score is based on the average number of CVE reported each year, *NCVE*, and the trend of the vulnerabilities, *TCVE*. Prior to define how these parameters are used to calculate the score, a comparison between real OSS projects has been done and the results can be consulted in table 5.1.

OSS project	NCVE	Users OpenHUB	Active Contributors	Lines of code
KeePass	0,29	230	0	2,28K
tar	4,3	2951	6	23,8K
OpenSSL	11,9	1879	131	438K
Firebug	0,5	5948	9	490K
glibc	5	914	94	1,24M
Apache	48	9452	29	1,8M
MySQL	32	9159	149	2,73M
wireshark	34,9	1269	250	3,35M
Iceweasel	0,1	28	7	13,7M
Firefox	98,6	13215	1057	13,8M
Chrome (Chromium)	107	2148	2004	14,1M

Table 5.1: Comparison of different OSS projects.

The OSS projects in table 5.1 are ordered by the number of lines of code. It should be noted that there is no direct relation between this and the number of vulnerabilities. Of course, larger projects have more probability of having bugs in the code, but it can be seen that the number of users is more directly relevant to the number of vulnerabilities reported. It is important to bear in mind that this number only reflects a small fraction of the real amount of users, but it appears to be a good indicator.

The clearest comparison can be done with the data from the three browsers: Iceweasel, Firefox and Chromium. They are three different alternatives to perform the same task, and therefore a fair comparison can be made. It can be clearly seen that the number of users and contributors is much lower for Iceweasel than for the other two cases, which impacts undoubtedly the number of CVEs reported. It seems reasonable to assume that the Iceweasel project is much more insecure than the other two due to the low number of users that the project has.

However, it may not be necessary to analyse the *number of users* to estimate the involvement of the community for all the projects. The amount of vulnerabilities reported each year for some projects is a good indicator of the revision by itself. This metric becomes important in those cases where the number of CVEs per year is not so high. Therefore, a deeper analysis has been done for projects in which the *NCVE* is lower than 5 and the results can be seen in table 5.2.

OSS project	**NCVE**	**Users OpenHUB**	**Active Contributors**
Iceweasel	0,1	28	7
KeePass	0,29	230	0
Firebug	0,5	5948	9
tar	4,3	2951	6
glibc	5	914	94
tcpdump	1,89	575	17
snort	1,29	86	20
iptables	1,13	331	20
dpkg	1,15	412	25

Table 5.2: Comparison of different OSS projects with NCVE lower than 5.

There are only three projects in which the average vulnerabilities reported is not even 1.0 per year: Iceweasel, KeePass and Firebug. However, the number of users of Firebug in OpenHub is very high, almost 6000 users, so it seems fair to consider it an indicator that the project is in fact very trustworthy, and hence the low number of CVEs reported. In the other two scenarios, this does not apply. The previous comparison of Iceweasel with Firefox and Chromium leaded to the conclusion that this project is not trustworthy. But what about KeePass? The number of users is higher than for Iceweasel, but there are no active contributors for the project. Therefore, the safest estimation is to consider the project untrustworthy. When talking about software security, it is more desirable to have false positives than false negatives, and in this case the information is not conclusive enough to make a clear decision.

Continuing with table 5.2, projects that have more than 500 users are also being reviewed. Even if the number of active contributors is not so high for some of them, this may be because the project is already very well-known and that it has been conscien-

tiously revised before, so they can be considered trustworthy.

Finally, there are some other projects like *Snort*, *iptables* or *dpkg* with less number of users than in the previous case, but they all have more than one vulnerability reported each year. In these cases, it may also be useful to consult the number of contributors: the average for them is 20 or more. It seems reasonable to think that this amount of developers working on improving the code is a good indicator of the code being reviewed. It seems fair to assume that, if the range of developers varies from 15 to 20, or if it is greater than that, the code is being revised. Therefore, this would be the criteria for those projects with less than 500 users.

The previous analysis has been performed for projects with few vulnerabilities per year. In those cases, the trend of the CVEs does not provide useful information, as the fluctuation in the number of vulnerabilities is almost indiscernible. However, for other scenarios it should be studied.

There are 3 possibilities considering the trend: that it is stable, lowering or increasing. An **increasing** trend is not desirable in any case. Therefore the maximum score, 10, is given to the TCVE parameter. However, this is not so straightforward for the other two cases. The TCVE scores should be weighted differently depending on the amount of vulnerabilities that are being considered. A stable trend of *glibc* is less risky than a lowering one of Chrome simply because the number of vulnerabilities in the latter case is much greater, and this should be taken into consideration. However, stable trends are also risky no matter the situation, because they mean that the project is not improving, so a score of 7 is assigned. The NCVE will be used after to weight this value. A lowering trend is desirable, but if the project has a very large number of vulnerabilities, it should still be considered a bit risky. Therefore, not a 0 is given, but a 4. The NCVE can be used to weight differently the trends. If a project with not so many vulnerabilities has a lowering trend, the final score will reflect it, so the TCVE=4 does not impact so much. However, if the project has a large number, it will still receive a vulnerability score of 4, reflecting the NCVE. If the score for lowering trends was lower, untrustworthy projects might be underrated.

By looking at the data from table 5.1, the NCVE can be categorised into four groups:

- **0 - 5**: this special scenario has been explained before. In order to make fair decisions, the number of users should be consulted.
- **5 - 20**: two projects fall in this category: OpenSSL and *glibc*. Even if the number of vulnerabilities is not so low to consider them completely trustworthy, their score should not be high neither. The weight should be higher than 0.1, since the trend of the vulnerabilities should be reflected in the score, but lower than 0.5, as the low number of CVEs indicates that the project is not so risky, even if the trend is increasing. However, it also depends on the severity of the vulnerabilities: if also

this trend is increasing, the project may not be so trustworthy. A score of 0.3 as been given: therefore, if the tendency is lowering, the vulnerability score will be also low (0.12) but not zero, as there are still some vulnerabilities reported. If the tendency is increasing, the score will be 3, which indicates that the project is not completely untrustworthy yet but it also depends on the severity.

- **20 - 70**: some examples are MySQL or Apache. The amount of vulnerabilities is significant in this case, but not so high as Firefox or Chrome. However, the important information to consider which of these projects is trustworthy is the trend. Hence, the weights in these cases should be similar: 0.9.
- **Greater than 70**: any project that has this amount of vulnerabilities is likely untrustworthy. Even if the score should be based in the trend of its vulnerabilities, the maximum weight, 1, should be given based on the NCVE. Therefore, if the trend is increasing, the final score will be the maximum, 10, indicating the high risk of the component.

Table 5.3 summarizes the different scores and weights.

Number CVE/year (NCVE)	0 <NCVE <5	(users)
	5 <NCVE <20	0.3
	20 <NCVE <70	0.9
	NCVE >70	1
Trend (TCVE)	Lowering	4
	Stable	7
	Increasing	10

Table 5.3: Scores based on CVE

The equation used to calculate this sub score is:

$$VulnerabilityScore = NCVE \times TCVE$$

5.6.1.2 Severity sub score

The *severity sub score* will be calculated based on the CVSS and it is very useful to evaluate the evolution of the project. CVSS numbers are calculated based on different parameters, such the exploitability or the impact of the vulnerability. Depending on their score, the CVSS are ranked in four categories: low, medium, high and critical.

As for CVE, the increasing trend is assigned the maximum score, 10, and the stable with 7. It has been mentioned that a stable trend is still untrustworthy, and therefore its score should not be lower. However, in this case the lowering one is associated with 0. The important aspect in this case is to highlight which CVSS are increasing: the critical and the high ones, or the low and medium. Low and medium vulnerabilities

may indicate that the attacker cannot obtain very useful information by exploiting the vulnerability, or that it is very complicated to attack. Therefore, their trend is not so relevant for this analysis, and their weight is very low: 0.05 and 0.1. The important aspect is to notice if the critical and high vulnerabilities are declining or not, and the different scores have been assigned to reflect this. Table 5.4 shows the possible values.

Critical trend (TC)	Lowering	0
	Stable	7
	Increasing	10
High trend (TH)	Lowering	0
	Stable	7
	Increasing	10
Medium trend (TM)	Lowering	0
	Stable	7
	Increasing	10
Low trend (TL)	Lowering	0
	Stable	7
	Increasing	10
N_critical/N_total (NCT)	NCT >0.25	1
	NCT <0.25	0
N_high/N_total (NHT)	NHT >0.25	1
	NHT <0.25	0

Table 5.4: Scores based on CVSS

Two other parameters are also considered: NCT and NHT. They reflect whether the majority of the vulnerabilities reported are severe (critical or high) or not (medium and low). This is also related to the number of CVE that were considered in the previous section: if few vulnerabilities were reported, but most of them corresponded to critical or high CVSS, the score should be higher. This can influence up to 1 point in the sub score. It has not been assigned a bigger weight because the trend is considered to provide more information.

The equation is:

$$SeverityScore = (0.6 \times NCT + 0.4 \times NHT) + 0.45 \times TC + 0.3 \times TH + 0.1 \times TM + 0.05 \times TL$$

Note: there are some exceptions for obtaining the score by applying this formula. Section 6.2.2 explain the motivation in details.

5.6.2 Final score

The FinalScore is based on the scores obtained for the dependencies as well as for the project itself. Different approaches have been considered:

- To calculate the final score as the mean of all the dependencies: the trustworthiness score obtained in this case is very high, due to the large amount of small libraries considered trustworthy.
- To calculate the mean of the scores different than 0: even if the score is lower in this case, the trustworthiness of the project itself is not being considered. Also, if one of the dependencies is very risky, the average may hide this risk.
- The minimum score of the dependencies: this approach is based on the principle *"A chain is only as strong as its weakest link"*. Therefore, the trustworthiness of the dependencies should be the lowest one, as this is the component with higher probability of introducing vulnerabilities into the project. However, the previous result may be useful to notice if the rest of the dependencies are trustworthy or not. If the minimum score is similar to the mean, the trustworthiness of all the dependencies may be considered the same. On the other hand, if the difference is significant, it means that the quality of one of the components is much lower than the quality of the rest.
- The minimum score considering the project: this tool is designed to provide information about a project to decide if it is trustworthy enough to include it in a new software. Therefore, not only the quality of its dependencies should be analysed but also the project itself.

For all of them, the score is calculated as:

$$Score = 10 - DependencyScore$$

Since the value of *DependencyScore* reflects the **risk** of the project and not its trustworthiness.

The comparison of these different approaches can be seen in section 6.2.2. The final score for trustworthiness will be the **minimum score** of the dependencies as well as the project. Otherwise, the value would not reflect the real risk of considering that project. However, as the other values may be useful to analyse more in deep where is the problem, all the results and the different scores for the project would be stored in a file named *Trustworthiness.txt*. Therefore, it can be seen if the problem is due to one of the dependencies, several of them or the project itself.

6 Implementation and evaluation

This tool has been written in *Perl* language. It uses *apt-rdepends* to find the dependencies of a project and a tool named *cve-search*, section 6.1, to find the CVE and CVSS of the abovementioned dependency.

CVEs have been explained in section 3.3 whereas CVSS explanation can be found in 3.4. However, it is important to state that this last explanation has been made for CVSSv3, which was release in June 2015. Therefore, it is possible that some of the CVSS numbers obtained correspond to the previous version in which other equations and parameters were considered.

6.1 Contributions

6.1.1 cve-search

cve-search tool has been utilized in this project to collect information about the vulnerabilities related to a package or library. The main authors are Alexandre Dulaunoy and Pieter-Jan Moreels. The source code is available on GitHub and can be accessed through the following link: https://github.com/cve-search/cve-search

This tool facilitate the search and processing of CVEs. It has been decided to use it inasmuch as there is no API available from the official CVE MITRE website. This project is also supported by organizations such as CIRCL (Computer Incident Response Center Luxembourg): https://www.circl.lu/services/cve-search/

Two scripts have been used to search through the databases:

- *search_fulltext.py*: to find the CVE related to a given package.
- *search.py*: to find the CVSS associated with a CVE

6.1.2 Black Duck Open Hub

Black Duck Open Hub is used to gathered information about the endorsement of the community in a particular OSS project. In those cases where the CVE numbers do not

provide enough data to consider a project trustworthy or not, the *number of users*, and even the *number of contributors* in some special situations, will be taken into consideration.

OpenHub is a website which provides information to compare OSS projects based on their popularity as well as the activity of the contributors. It was founded by Jason Allen and Scott Collison in 2004 and the site lists 672,372 open source projects from 688,219 source control repositories source control repositories (May 2016).

6.1.3 Linear regression

Linear regression has been the approach used to find trends the CVE and the CVSS numbers follow. The idea is to fit the data to the following equation:

$$y = ax + m$$

And to use the slope, a, to find the trends.

The Perl module **Statistics::Regression** by Ivan Tubert-Brohman allows to fit data to an equation of the form:

$$y = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_3 x_3$$

As the desire output is an equation of the form:

$$y = ax + m$$

The appropriate use of this module is:

```
reg->include(y, [1.0, x]);
```

Where y represents the number of CVEs or CVSS that year

6.1.4 Considerations

For a fair analysis of the evolution of the number of vulnerabilities, the tool does not consider the first years after the implementation of the CVE standard. It has been noticed that most of the projects have very few CVEs reported the first years, so if they are consider, they trends obtained will be misleading. This low number of vulnerabilities may not be related to the security of the project but that it was not so known or that scrutinizing the code was not considered to be so important and therefore developers did not put that much effort on it. A declining trend of the last years would possibly be unnoticed because the number of CVEs reported in the last years is higher than at the beginning. The current year should also be ignored as the information is still incomplete and could also be misleading.

The author believes that the best approach is to consider only the last years of the component for obtaining the trends. To be precise, only the CVEs reported the nine past years have been considered, based on the trends of some of the projects analysed.

6.2 Evaluation

6.2.1 Evaluating the DependencyScore

First of all, the weights and parameters used to obtain the dependency score have been tested. The script has been modified to consider only the given project and not its dependencies. This way, it is possible to obtain the score based on its CVE, CVSS and number of users where necessary, and to contrast it for different well known projects (table 6.1). This has been used to adjust the different weights in case some of the projects receive a score that did not seem to fit the reality.

OSS project	DependencyScore (no users considered)	DependencyScore (considering users)
KeePass	10	0
tar	4.54	10
OpenSSL	3.32	3.32
Firebug	6.92	10
glibc	3.08	10
Apache	1.12	1.12
MySQL	4.48	4.48
Iceweasel	10	0
Firefox	0	0
Chrome	4.20	4.20
Explorer	0	0

Table 6.1: Dependency scores obtained for different OSS projects.

For instance, *Iceweasel* obtained a score of 10 in the first tests as the number of CVEs reported was very low. However, when taking into consideration the number of users, it seems unrealistic to consider it trustworthy. Whereas other web browsers have thousand of users and numerous CVEs reported, the lack of them in this case appears to reflect a lack of revision of the code. The scores considering the number of users seem more accurate than the ones obtained in the first function. This also applies for those projects which received a higher score in the second case. For instance, *glibc* and Firebug. The reason for this is that a lot of importance was given to the trends of the vulnerabilities without highlighting the few number of CVEs that were reported each year. When consulting the number of users of each of them, it was shown that they are mature projects with a lot of control from the community, and still, the number of vulnerabilities found

was very little. Therefore, it is fair that they are considered trustworthy.

Some other consideration have been done:

- If the *VulnerabilityScore* is 10 and the *SeverityScore* is greater than 9, the final score of the dependency is 10 (maximum risk). The *VulnerabilityScore* reflects that there is a high number of vulnerabilities reported and that the trend is increasing, whereas the value of the *SeverityScore* is obtained only if the trends for critical and high vulnerabilities are increasing. Therefore, the project should be considered completely untrustworthy.
- If the *VulnerabilityScore* is 10, NCT is 1 and TC is increasing, the final score is also 10. The NCT reflects that a high number of the total amount of vulnerabilities are critical (more than a quarter) and the TC is given based on the trend of the critical vulnerabilities. This situation is also completely untrustworthy.

In these two cases, a warning will also be printed so that the user can be aware if some of the dependencies are troublesome. If a project was to be built depending on it, it is very likely that attackers could find a way to attack the new project through this component. This was the scenario for both Firefox (the first condition) and Internet Explorer (the second).

The severity score for Google Chrome may seem lower than expected. However, when its parameters and trends are compared with other projects similar to it, like Firefox and Internet Explorer (table 6.2), this score is justified. In the other two cases the amount of critical vulnerabilities is significant and also their trend are increasing, which should be considered very untrustworthy.

	Vulnerability Score			**Severity Score**						
Web Browser	**NCVE**	**TCVE**	**Score**	**NCT**	**NHT**	**TC**	**TH**	**TM**	**TL**	**Score**
Firefox	98.6	I	10	1	0	I	I	I	I	10
Chrome	107	I	10	0	1	D	I	I	S	4.75
Explorer	97.1	I	10	1	0	I	D	D	S	10

Table 6.2: Comparison of the parameters obtained for three well-known web browsers.

6.2.2 Evaluating the FinalScore

Four approaches have been considered to calculate the *FinalScore*:

- As the mean of all the dependencies.
- As the mean of the dependencies with scores different than 0.

- The minimum score of the dependencies.
- The minimum score considering the project.

The results obtained for the projects analysed beforehand can be seen in table 6.3. *ND* is the Number of Dependencies.

Project	Final	ND	Average	ND (without score=0)	Average (without score=0)	Minimum Score Dependencies	Score Project
tar	10	8	10	0	10	10	10
glibc	10	4	10	0	10	10	10
KeePass	0	79	9.87	1	0	0	0
OpenSSL	3.32	17	10	0	10	10	3.32
MySQL	0	100	9.31	8	1.41	0	4.48
wireshark	5.32	126	10	0	10	10	5.32
Apache	0	131	9.4	9	1.25	0	1.12
Firefox	0	136	9.93	1	0	0	0
Chrome	0	151	9.89	2	1.66	0	4.20

Table 6.3: Scores obtained for different projects.

It can be seen that the average of the dependencies of KeePass is 0, as well as for Firefox. The components with that score are the library *libgdiplus* and *passwd*. The reason of the low score is that the number of users and contributors in OpenHub is very low. They seem like false positives when consulting the characteristics of the project: there were many commits at the beginning of the projects, so it is likely that the libraries are now mature enough to not need major changes or revision. However, these false positives are very difficult to detect without the risk of not noticing real untrustworthy projects, like Iceweasel.

The advantage of choosing the minimum value to assess about the trustworthiness can be seen in the analysis of Chrome. Even if the minimum score of the dependencies is 0, this is due again to the library *passwd*, the same that for firefox. However, if this one is not considered, the minimum score would be 3.32. The dependency *ca-certificates* depends on OpenSSL and the trustworthiness of this last one is 3.32. It may be possible that the vulnerabilities of OpenSSL affect *ca-certificates* and thus, Chrome may be also vulnerable to them: therefore, its trustworthiness should not be higher than OpenSSL. This is exactly what happened with Heartbleed, so the tool is able to find these weak points through the dependencies.

MySQL also obtains a low score due to its dependencies. In this case, not only the minimum score is 0, but also the average of them is very low: 1.41. This score is mainly because there are several libraries with few users, as explained for KeePass or Firefox.

In order to be sure that they do not represent a threat for the project, they should be studied to decide whether they are really false positives or really untrustworthy.

Even if there are some false positives, the criteria that has been used to identify which components are trustworthy appears to be suitable. This is evidenced by the dependencies of *wireshark*: of 136 elements, none of them was considered risky. The majority of the libraries do not appear in OpenHub, and those which are in its database have a big number of users or contributors associated to them. Therefore, it is assumed that they are secure, which seems appropriate in this case.

7 Conclusions

The goal of this project was to investigate ways to assess the trustworthiness of an open source looking at other software projects reused by it: its **dependencies**. For assessing about their trustworthiness, different metrics have been studied and the number of vulnerabilities, CVE, and its severity, CVSS, have been selected.

The designed tool provides with a *FinalScore*, which ranges from 0 to 10, based on the *DependencyScores* collected from all the dependencies and the project. This one is based on:

- **Vulnerability sub score**: based on the average number of CVE per year and the trend of these vulnerabilities. For some dependencies, the number of vulnerabilities found was very low. To decide whether this was a sign of good design or a lack of revision of the code, the number of users of the project in *OpenHub* was considered.
- **Severity sub score**: based on the CVSS and their trend. Different weights were assigned depending on the risk associated with the vulnerabilities: it is not the same to have one critical security flaw than one which is considered to have low severity.

These different weights have been adjusted after analysing the results for real OSS projects which have been shown in 6.2.

This project has proved how the trustworthiness of the dependencies have a direct relation with the security of the component that uses them. It is possible to detect troublesome dependencies before a severe vulnerability in one of them affects the project. It has also shown how the track record of a software component can be used to infer its functioning in the future.

However, many different considerations have to be addressed to obtain a fair score and, even so, in some occasions it is not possible to discern which elements are trustworthy. The amount of small libraries that are used is one of the main reasons of this problem. Nevertheless, if there is not enough evidence to prove that an element is secure, the sensible decision to take is to consider it untrustworthy.

Finally, it is very important to highlight that a clear understanding of the vulnerable parts of a software project is essential to avoid security holes. If dependencies are not considered as part of this project, many security flaws may not be detected.

7.1 Future work

There are some aspects that could be improved for this tool in future works.

First, some other metrics could also be taken into consideration for a fairer comparison. The idea is to consider different thresholds depending on other parameters, as the ones that were analysed in chapter 3. These additional metrics could also reduce the number of false positives since the the parameters that are being used may not be accurate in some cases.

Consider scalability problems: every year the amount of vulnerabilities reported rises significantly and this should be addressed. This is also an issue for the CVE and CVSS standards: in fact, the CVE syntax has been modified because there were not enough identifiers for all the vulnerabilities reported each year.

Finally, it is possible that one library has many of its vulnerabilities related to only one project. This could happen if the library is not intended to work under the conditions of that project. This would impact its score in an unfair way, because the problem is not the software itself, but how it is used.

Appendix A

Appendices

A.1 CVSS

A.1.1 Base Score

A.1.1.1 Calculations

Exploitability sub score:

$$Exploitability = 8.22 \times AttackVector \times AttackComplexity \times PrivilegeRequired \times UserInteraction$$

Impact sub score (ISC):

$$ISC_{base} = 1 - [(1 - Impact_{Conf}) \times (1 - Impact_{Integ}) \times (1 - Impact_{Avail})]$$

Depending on the Scope:

$$Unchanged \rightarrow ISC = 6.42 \times ISC_{Base}$$
$$Changed \rightarrow ISC = 7.52 \times [ISC_{Base} - 0.029] - 3.25 \times [ISC_{Base} - 0.02]^{15}$$

Base Score:

$$ISC <= 0 \rightarrow BaseScore = 0$$
$$Else,$$
$$ScopeUnchanged \rightarrow BaseScore = Roundup(Minimum[(Impact + Exploitability), 10])$$
$$ScopeChanged \rightarrow BaseScore = Roundup(Minumum[1.08 \times (Impact + Exploitability), 10])$$

A.1.1.2 Metrics and Scores

Base Metrics			
Exploitability Metrics			
Attack Vector (AV)	Physical (P)	The attacker must have physical access to the vulnerable system	0.2
	Local (L)	The The attacker must have physical access to the vulnerable system	0.55
	Adjacent Network (A)	The vulnerable component is bound to the network stack but is limited to the same shared physical or logical network	0.62
	Network (N)	The attacker's path is through OSI layer 3 (network layer). These types of vulnerabilities are often described as remotely exploitable	0.85
Attack Complexity (AC)	Low (L)	There are no special conditions for access to the vulnerability, so the attacker can expect repeatable success against the component	0,77
	High (H)	Specialised access conditions exist, which requires the attacker to prepare against the vulnerable component.	0.44
Privileges Required (PR)	None (N)	The attacker is unauthorized prior to attack	0.85
	Low (L)	The attacker is authorized with privileges that provide basic user capabilities	0.62 / 0.68
	High (L)	The attacker is authorized with privileges that provide control over the vulnerable component	0.27 / 0.50
User Interaction (UI)	None (N)	The system can be exploit without interaction	0.85
	Required (R)	For the exploitation an action has to been taken before the vulnerability can be exploited	0.62
Scope			
Scope (S)	Unchanged (U)	A exploited vulnerability can only affect resources managed by the same authority	-
	Changed (C)	The explotation can affect resources beyond the privileges intended by the vulnerable component. In this case, the vulnerable component and the impact are different	-

Table A.1: CVSS Base Metrics and scores - Exploitability Metrics and Score

Impact Metrics			
Confidentiality Impact (C)	High (H)	There is total information disclosure, providing all resources within the impacted component being divulged to the attacker	0.56
	Low (L)	Access to some restricted information is obtained but how much/what information is of the control of the attacker	0.22
	None (N)	There is no loss of confidentiality	0
Integrity Impact (I)	High (H)	Total loss of integrity, the attacker can modify any files or information	0.56
	Low (L)	Modification of data is possible but the scope is limited	0.22
	None (N)	There is no loss of integrity	0
Availability Impact (A)	High (H)	Total loss of availability, The attacker is able to deny access to resources in the impacted component.	0.56
	Low (L)	There is reduced performance or loss of some functionality	0.22
	None (N)	There is no impact to availability	0

Table A.2: CVSS Base Metrics and scores - Impact Metrics

A.1.2 Temporal Metrics

A.1.2.1 Calculations

Temporal Score:

$$Temporal = Roundup(BaseScore \times ExploitCodeMaturity \times RemediationLevel \times ReportConfidence)$$

A.1.2.2 Metrics and Scores

Temporal Metrics			
Exploit Code Maturity (E)	Not Defined (X)	This metric will be skipped in the scoring equation	1
	High (H)	The vulnerability can be exploit in all situations	1
	Functional (F)	Functional exploit code is available and works in most situations	0.97
	Proof of Concept (P)	Proof-of-concept exploit code is available, not practical for most systems.	0.94
	Unproven (U)	No exploit code is available	0.91
Remediation Level (RL)	Not Defined (X)	This metric will be skipped in the scoring equation	1
	Unavailable (U)	No solution can be applied or is not available	1
	Workaround (W)	Unofficial solution available to mitigate the vulnerability	0.97
	Temporary fix (T)	Official but temporary fix available	0.96
	Official Fix (O)	A complete solution is available, by a patch or an upgrade	0.95
Report Confidence (RC)	Not Defined (X)	This metric will be skipped in the scoring equation	1
	Confirmed (C)	Detail reports exist about the vulnerability	1
	Reasonable (R)	Significant details are published, but either no complete knowledge about the root cause or no access to source code prevent to confirm all the interactions	0.96
	Unknown (U)	There are reports that indicate a vulnerability present but the nature and the cause of the vulnerability is unknown	0.92

Table A.3: CVSS Temporal Metrics and scores

A.1.3 Environmental Metrics

Environmental Metrics			
Security Requirements (CR, IR, AR)	Not Defined (X)	This metric will be skipped in the scoring equation	1
	High (H)	Loss of Condientiality, Integrity or Availability is likely to have a capastrophic effect.	1.5
	Medium (M)	Loss of Condientiality, Integrity or Availability is likely to have a serious effect.	1
	Low (L)	Loss of Condientiality, Integrity or Availability is likely to have only a limited effect.	0.5

Table A.4: CVSS Environmental Metrics and scores

A.1.3.1 Calculations

Modified Exploitability sub score:

$$\begin{aligned} M.Exploitability =& 8.22 \times M.AttackVector \times M.AttackComplexity \\ & \times M.PrivilegeRequired \times M.UserInteraction \end{aligned}$$

Modified Impact sub score (ISC):

$$ISC_{Modified} = Minimum[[1-(1-M.I_{Conf} \times CR) \times (1-M.I_{Integ} \times IR) \times (1-M.I_{Avail} \times AR)], 0.915]$$

Depending on the Scope:

$$Unchanged \rightarrow M.Impact = 6.42 \times ISC_{Modified}$$
$$Changed \rightarrow M.Impact = 7.52 \times [ISC_{Modified} - 0.029] - 3.25 \times [ISC_{Modified} - 0.02]^{15}$$

Environmental Score:

$$
\begin{aligned}
&If M.Impact <= 0 \rightarrow BaseScore = 0 \\
&Else, \\
ScopeUnchanged \rightarrow BaseScore &= \\
&Roundup(Roundup(Minimum[(M.Impact + M.Exploitability), 10]) \\
&\times ExploitCodeMaturity \\
&\times RemediationLevel \\
&\times ReportConfidence) \\
ScopeChanged \rightarrow BaseScore &= \\
&Roundup(Roundup(Minimum[1.08 \times (M.Impact + M.Exploitability), 10]) \\
&\times ExploitCodeMaturity \\
&\times RemediationLevel \\
&\times ReportConfidence))
\end{aligned}
$$

A.1.3.2 Metrics and Scores

Modified Base Metrics
Modified Attack Vector (MAV)
Modified Attack Complexity (MAC)
Modified Privileges Required (MPR)
Modified User Interaction (MUI)
Modified Scope (MS)
Modified Confidentiality (MC)
Modified Integrity (MI)
Modified Availability (MA)

Table A.5: Modified Base Metrics

A.2 Source Code

A.2.1 Directions

Given a software project located in the repository, the script returns a score based on the trustworthiness of its dependencies and of the project itself.

Requirements:

- apt-rdepends
- cve-search: https://github.com/cve-search/cve-search. The fulltext index is necessary. For more details consult the git hubpage.
- Perl modules:
  - List::Util
  - List::MoreUtils
  - Statistics::Regression
  - WWW::Mechanize

Use:
To be able to get the dependencies of a project, it has to be possible to install it with the *apt* tool. For instance, Google Chrome has to first be added to the repository.
Once that it is accessible via *apt*:
./GetTrustworthiness.pl firefox
./GetTrustworthiness.pl chrome google-chrome-stable

This tool creates a folder with the name of the project and in that one, two files are stored:

- In the first one, *Dependencies.txt*, the different dependencies, with their CVEs and scores, are stored.
- In the second one, *Trustworthiness.txt*, the different parameters to obtain the final score are indicated.

The name of the project in the repository may be different than the usual name of the package: for instance, *chrome* and *google-chrome-stable* or *apache* and *apache2*. In this case, both should be included:

- First parameter: the normal name - it'll be used to find the CVEs/CVSS related to it.
- Second parameter: the name of the package in the repository.

A.2.2 Code

```
#!/usr/bin/perl

use strict;
use warnings;
use List::Util qw/ max/;
use List::MoreUtils qw/ uniq/;
use Statistics::Regression;
use WWW::Mechanize;

#Given a software project located in the repository, this script returns a
    score
#based on the trustworthiness of its dependencies and of the project itself

#Requirements:
#apt-rdepends
#cve-search and its requirements: https://github.com/cve-search/cve-search
#The Perl modules:
   #List::Util
   #List::MoreUtils
   #Statistics::Regression
   #WWW::Mechanize

#Note: the name of the project in the repository may be different than the
#usual name of the package. In this case, both should be included:
#  the first parameter: the normal name - it'll be used to find the CVEs/CVSS
    related to it
#  the second parameter: the name of the package in the repository



################################################################################

#Subroutines:


sub mean{
   #This subroutine obtains the mean of an array
   my @data = @_;
   my $sum = 0;
   foreach my $value (@data) {$sum+=$value}
   if ($sum==0) {return 0;}
   return $sum/@data;
}#sub mean


sub trend{
```

```
   #This subroutine calculates the trend of an array and it returns a score
   #based on the results
   #The first parameter: the score to assign when the trend is lowering (4 for
       CVEs' trend, 0 for CVSSs')
   my $reg=Statistics::Regression -> new ("regression",["intercept","slope"]);
   my ($weight_low, $number_years, @array)= @_;
   my @array_backwards= reverse(@array);#Array backwards: the data is analysed
       starting from the last years
   shift(@array_backwards); #delete current year: incomplete information

   for (my $counter=0; $counter<$number_years-2 && $counter<9; $counter++){
   #Without considering current year, the data is obtained from the last 9 ones
   $reg->include($array_backwards[$counter],[1.0,$counter]);
   }

   #Obtain the slope:
   my @theta = $reg->theta();
   my $slope = $theta[1];

   #Thresholds for slope:
   if ($slope < -0.2){ return 10; }#Increasing trend (it is backwards)
   elsif ($slope == 0){ return 0; }#If there are no values, or only 1, there
       is no trend to calculate - otherwise it will be consider stable
   elsif ($slope >= -0.2 && $slope <= 0.2){ return 7; }#Stable
   elsif ($slope > 0.2){ return $weight_low;}#Declining

}#sub trend

################################################################################


#Parameters: it can be only one or two
#If two: the first one corresponds to the name of the project and the second to
#how to look for it in the repository (i.e. chrome and google-chrome-stable)
#Only one parameter is needed if the name is the same for both cases (i.e
    OpenSSL)
my $size=@ARGV;

if ($size!=1 && $size!=2)
{
   die "Illegal number of parameters.\nPlease introduce the correct name/s of
       the package.\n";
}

my $package=$ARGV[0];
my $name_package=$package;

#If there are 2 arguments, the last one corresponds to the name of the package
    in the repository
```

```
if ($size==2)
{
   $name_package=$ARGV[1];
}

#We create a directory where all the necessary files will be stored
my $newdirectory = "./$package";
mkdir $newdirectory;

my $DependencyFile = "$newdirectory/DependencyTree.txt";
my $TrustworthinessFile = "$newdirectory/Trustworthiness.txt";
my $CVEFile="$newdirectory/CVE.txt";
my $CVSSFile="$newdirectory/CVSS.txt";

#To obtain the dependency tree:
my $query = "apt-rdepends $name_package >> $DependencyFile";
#All the information of the dependency tree will be stored in a file
my $results = `$query`;

#To just leave the dependencies: what is after the expression 'Depends:' or
    'PreDepends:'
open my $fileHandle_Reading, "<", $DependencyFile or die "Can't access
    '$DependencyFile'";
my @dependencies;
my $counter=0;
while (<$fileHandle_Reading>){
   chomp;
   my @words = split(' ');
   my $words = @words;
   for (my $i=0; $i<$words; $i++){
   if ($words[$i] eq 'Depends:' || $words[$i] eq 'PreDepends:'){
      $dependencies[$counter]=$words[($i+1)];
      $counter++;
      }#if
   }#for
}#while
close $fileHandle_Reading;

#Remove the file:
unlink $DependencyFile || print "Not possible to delete file
    DependencyTree.txt";;

#First: check if there are dependencies. If not, the final score is 10
my $dependencies_size = @dependencies;
if ($dependencies_size==0)
{
   print "The software project $package has no dependencies.\nThe score is:
       10.";
   die;
```

```
}

#As some dependencies may appear more than one time: sort and uniq
my @dependencies_sorted=sort @dependencies;
my @dependencies_uniq= uniq @dependencies;

#Some parameters and sub scores have to be obtained to calculate the
    FinalScore:
my @DependencyScore;     #The score given to a dependency
my $ND=@dependencies_uniq; #Number of dependencies of the project
my $NRisk=0;       #Number of dependencies with DependencyScore=10



#For each project, we will get the CVEs related as well as the CVSS, and
    calculate the subscores
#The results will be stored in a file: Trustworthiness.txt
open my $fileHandle_writing, ">>", "$newdirectory/Dependencies.txt" or die
    "Can't access Dependencies.txt";

my $name_consult;
for (my $i=0; $i<=$ND;$i++){

   #The last element to consult will not be a dependency, but the project
       itself
   if ($i==$ND){
      $name_consult=$package;
   }
   else{
      $name_consult=$dependencies_uniq[$i];
   }


   #First: print the name of the dependency in the file
   print $fileHandle_writing "---\n";
   print $fileHandle_writing $name_consult . "\n";

   #After: get all the CVE into an array
   my $CVE_query = "./cve-search-master/bin/search_fulltext.py -q
       $name_consult | sort > $CVEFile";
   my $query2 =`$CVE_query`;
   open my $CVE_Reading, "<", $CVEFile or die "Can't access '$CVEFile'";
   chomp (my @CVE_list = <$CVE_Reading>);
   close $CVE_Reading;
   my $CVE_size= @CVE_list;   #Number of CVE
   #In case the libary has no CVEs associated
   if ($CVE_size==0){
      $DependencyScore[$i]=0;
```

```
   print $fileHandle_writing "DependencyScore: " . $DependencyScore[$i] .
       "\n";
}
else{

#To obtain the Vulnerability sub score, we need:
my $VulnerabilityScore;  #Subscore based on the CVE
my $NCVE;        #The average of CVEs reported per year
my $TCVE;        #The tendency: it can be L (Lowering), S (Stable) or I
    (Increasing)

#The years that will be considered will start with the first CVE
# reported: we assume that the year where the project was release
# corresponds to the first year in which CVEs were reported for it
my @years;
my $First_CVE=$CVE_list[0];
my @CVE_split=split("-",$First_CVE); #The syntax is: CVE-YYYY-XXXX
my $year=$CVE_split[1];

my @CVE_year;      #How many CVEs were reported each year: to obtain the
    average
#We create a vector of years, starting in the former one and
#finishing in 2016 (this year)

my $counter_CVE=0;
for (my $y=$year; $y<=2016; $y++){
   push @years, $y; #Add year to the vector

   #We also want to know how many CVEs were reported that year:
   $CVE_year[$counter_CVE]=grep /$y/, @CVE_list; #Count how many times does
       the year appear
   #$reg->include($CVE_year[$counter_CVE],[1.0,$counter_CVE]);
   $counter_CVE++;
}
#The mean:
$NCVE = mean(@CVE_year);


#Ranges:
if ($NCVE <= 5){ #We have to consult the number of users:
my $NumberUsers=0;
my $NumberContributors=0;
my $name_lc=lc($name_consult);
#NOTE: to search in OpenHub, the name of the package has to be lowercase

#Build the request url: to get number of users
my $url= 'https://www.openhub.net/p?query=' . $name_lc;
#Request the url from the server and get the content inside the tags:
#<a href="/p/$name_package/users"></a>
```

```
my $page = WWW::Mechanize->new();
$page->get ( $url ) or die "OpenHub server did not reply";
my @links = $page->links();
my $found=0;
for my $link (@links){
if ($link->url eq "/p/$name_lc/users"){
      $NumberUsers=$link->text; #The content inside the tags
      $found=1;
   }
}
if ($found==1){ #If there is a project in OpenHub
$NumberUsers=~s/,//g; #Delete the comma (for thousand)
$NumberUsers=$NumberUsers+0; #This is needed because otherwise the variable
    is considered as text
}

printf $fileHandle_writing "Number of users: " . $NumberUsers . " \n";

#Note: $NumberUsers is considered text
if ($NumberUsers >= 500){ #Very well known project with few vulnerabilities
   $DependencyScore[$i]=0;
}
elsif ($NumberUsers == 0){
#If the project does not exist in OpenHub or there are no users at all:
    most likely the project is a small library or package: we can consider
    it secure
$DependencyScore[$i]=0;
}

else { #Number users> 1-499
#Depends on number of contributors:
#Get the content inside the tags:
#<a href="/p/$name_package/contributors/summary"></a>

$found=0;
for my $link (@links){
   if ($link->url eq "/p/$name_lc/contributors/summary"){
      $NumberContributors=$link->text; #The content inside the tags
      $found=1;
   }
}

if ($found==1){ #If there is a project in OpenHub
$NumberContributors=~s/,//g; #Delete the comma (for thousand)
$NumberContributors=$NumberContributors+0; #This is needed because
    otherwise the variable is considered as text
}
```

```
printf $fileHandle_writing "Number of contributors: " . $NumberContributors
    . " \n";

if ($NumberContributors >= 15){
#We assume that there are people revising the code
$DependencyScore[$i]=0;
}
else{
#If not: we assume that there may be vulnerabilities undiscovered because a
    lack of revision
$DependencyScore[$i]=10;
}
}#else

}#if NCVE>5
#If NCVE>5, we should calculate the trend as well as the CVSS associated in
    order to provide a score for the dependency
else{
if ($NCVE > 5 && $NCVE <= 20){$NCVE=0.7;}
elsif ($NCVE > 20 && $NCVE <= 70){$NCVE=0.9;}
elsif ($NCVE > 70){$NCVE=1;}

#For trend:
my $length_years=@years;
$TCVE = trend (4, $length_years,@CVE_year);
#Vulnerability sub score:
$VulnerabilityScore=$NCVE * $TCVE;

##############################################

#Now: for each CVE, obtain the CVSS
my @CVSS_list;

#Depending on severity:
my @CVSS_critical;
my @CVSS_high;
my @CVSS_medium;
my @CVSS_low;

#To obtain the average proportion of critical/high vulnerabilities
#compared to the total number: (to obtain NCT, NHT)
my @N_critical_total;
my @N_high_total;

for (my $f=0; $f<$CVE_size; $f++){
#Get the CVSS
#my $CVSS_query = "./cve-search-master/bin/search.py -c $CVE_list[$f] |
    grep -oP '(?<="cvss": )[^,]+' > ${newdirectory}/CVSS.txt";
```

```
my $CVSS_query = "./cve-search-master/bin/search.py -c $CVE_list[$f] >
    $CVSSFile";
my $query3 =‘$CVSS_query‘;
open my $CVSS_Reading, "<", $CVSSFile or die "Can’t access ’$CVSSFile’";

while (my $lineCVSS = <$CVSS_Reading>){
   chomp $lineCVSS;
   my @fields=split ",", $lineCVSS;
   my ($CVSS_field)=grep {/"cvss":/} @fields; #pattern: "cvss": x.x
   if ($CVSS_field){ #There is a small possibility that there is no CVSS
       associated with the CVE
   my @CVSS_fields=split(" ",$CVSS_field);
   my ($CVSS) = $CVSS_fields[1] =~ /"([^"]*)"/; "#Remove double quotes in
       case they exist
   if ($CVSS) {$CVSS_list[$f]=$CVSS;}
   else {$CVSS_list[$f]=$CVSS_fields[1];}
   }
   else {#If there is no CVSS associated: 0.0
      $CVSS_list[$f]=0.0;
   }
}#while

close $CVSS_Reading;
#Print: CVE + CVSS
print $fileHandle_writing $CVE_list[$f] . ": " . $CVSS_list[$f] . "\n";


}#for (CVE)

#When we have all the CVSS, obtain the scores
my $index_CVSS=0; #Index to go through the @CVSS_list array
my $index_severity=0; #Index to go through the 4 arrays assinged for
    severity
for (my $c=0; $c<@CVE_year; $c++){
#The arrray @CVE_year contains how many CVEs were reported each year
my $n_times=$CVE_year[$c]; #The number of CVEs that year
$CVSS_critical[$index_severity]=0;
$CVSS_high[$index_severity]=0;
$CVSS_medium[$index_severity]=0;
$CVSS_low[$index_severity]=0;

while ($n_times>0){ #While at least one CVE reported in that year
#hasn’t been assigned a severity: we consult the associted CVSS
   my $CVSS_value=$CVSS_list[$index_CVSS];
   #How many vulnerabilities of each kind occurred:
   if ($CVSS_value){ #If there is a value
   if ( $CVSS_value > 0.0 && $CVSS_value <= 3.9){
   $CVSS_low[$index_severity]+=1;}
   elsif ( $CVSS_value >= 4.0 && $CVSS_value <= 6.9){
```

```
      $CVSS_medium[$index_severity]+=1;}
      elsif ( $CVSS_value >= 7.0 && $CVSS_value <= 8.9){
      $CVSS_high[$index_severity]+=1;}
      elsif ( $CVSS_value >= 9.0 && $CVSS_value <= 10){
      $CVSS_critical[$index_severity]+=1;}
      }
   $n_times--;
   $index_CVSS++;
   }#while (n_times)

   if ($CVE_year[$c]>0){
   $N_critical_total[$index_severity]=$CVSS_critical[$index_severity]/$CVE_year[$c];
   $N_high_total[$index_severity]=$CVSS_high[$index_severity]/$CVE_year[$c];
   }
   else {
$N_critical_total[$index_severity]=0;
   $N_high_total[$index_severity]=0;
   }

   $index_severity++;
   }#for (CVE_year)

   #Trend: same as CVE
   #Only recent years (from 2006), without considering 2016
   #Backwards
   my $TC = trend(0, $length_years, @CVSS_critical); #Trend critical
   my $TH = trend(0, $length_years, @CVSS_high); #Trend high
   my $TM = trend(0, $length_years, @CVSS_medium); #Trend medium
   my $TL = trend(0, $length_years, @CVSS_low); #Trend low


   #Proportion of critical/high vulnerabilities
   my $NCT = mean (@N_critical_total);
   if ($NCT>0.25) {$NCT=1;}
   else {$NCT=0;}
   my $NHT = mean (@N_high_total);
   if ($NHT>0.25) {$NHT=1;}
   else {$NHT=0;}

   #Now that we have all data, we can calculate the Severity sub score:
   my $SeverityScore = (0.6*$NCT+0.4*$NHT)+0.45*$TC+0.3*$TH+0.1*$TM+0.05*$TL;
       #Subscore based on the CVSS

   #With the vulnerability and the severity score, we can obtain the
       dependency score
   $DependencyScore[$i]=0.2*$VulnerabilityScore+0.8*$SeverityScore;

   if ($VulnerabilityScore==10 && $SeverityScore>=9.0 ||
       $VulnerabilityScore==10 && $NCT==1 && $TC==10){
```

```
      $DependencyScore[$i]=10;
      print "WARNING: considered very risky.\n";
      printf $fileHandle_writing "WARNING: considered very risky.\n";
   }


print $fileHandle_writing "VulnerabilityScore: " . $VulnerabilityScore . "\n";
print $fileHandle_writing "SeverityScore: " . $SeverityScore . "\n";
   }#else: NC>5


print $fileHandle_writing "DependencyScore: " . $DependencyScore[$i] . "\n";

   }#else

}#for (dependencies)

#remove the CVE.txt and CVSS.txt files:
unlink $CVEFile || print "Not possible to delete file CVE.txt";
unlink $CVSSFile || print "Not possible to delete file CVSS.txt";;

#Close the Dependencies.txt file:
close $fileHandle_writing;
#Write the results in the new file:
open $fileHandle_writing, ">>", $TrustworthinessFile or die "Can't access
    Trustworthiness.txt";


#############################################################################
#The final score can be calculated:

#Different aggregation functions are considered:
# 1) the average of all the dependencies
# 2) the average of the dependencies with score != 0
# 3) the minimum score of the dependencies
# 4) the minimum score considering also the project -> the one selected for
    the final score

#The last value of the DependencyScore corresponds to the project:
my $ProjectRisk=pop(@DependencyScore);
my $ProjectScore=10.0 - $ProjectRisk;

#Now we have only the dependency scores:
#We obtain the minimum score for the dependencies (the one that corresponds to
    the maximum risk)
my $maximum = max @DependencyScore;
my $min_score = 10 - $maximum;

#The average: considering all the libraries
```

```
my $average_all=mean(@DependencyScore);

#To obtain the average of those with score !=0
my @Scores_notzero;

my $not_zero=0;
for (my $IndexScore=0; $IndexScore<@DependencyScore; $IndexScore++){
   if ($DependencyScore[$IndexScore] != 0) {
   $Scores_notzero[$not_zero]=$DependencyScore[$IndexScore];
   $not_zero++;
   }
}
#We store the results in the file Trustworthiness.txt
my $average_all_score=10.0 - $average_all;
printf $fileHandle_writing "Average score considering all libraries (" . $ND .
    " dependencies): %.2f\n", $average_all_score;

my $average_notzero = mean(@Scores_notzero);
my $average_notzero_score=10.0 - $average_notzero;
printf $fileHandle_writing "Average score considering only scores different
    than 0 (" . $not_zero . " dependencies): %.2f\n", $average_notzero_score;

printf $fileHandle_writing "Minimum score of the dependencies: %.2f\n",
    $min_score;

printf $fileHandle_writing "Score of the project itself: %.2f\n",
    $ProjectScore;

#To calculate the final score: we select the minimum value
my $Score;
if ($min_score < $ProjectScore){$Score=$min_score;}
else {$Score=$ProjectScore;}

if ($Score == 0) {
print "WARNING: very untrustworthy.\n";
print $fileHandle_writing "WARNING: very untrustworthy.\n";
}

printf "The trustworthiness is: %.2f\n", $Score;
printf $fileHandle_writing "\n\nThe trustworthiness is: %.2f\n", $Score;

close $fileHandle_writing;
```

Bibliography

[1] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 1992.

[2] Ye Yang, Qing Wang, and Mingshu Li. Process trustworthiness as a capability indicator for measuring and improving software trustworthiness. *Lab for Internet Software Technology, Institute of Software Chinese Academy of Sciences.*

[3] Tim Boland, Charline Cleraux, and Elizabeth Fong. Toward a preliminary framework for assessing the trustworthiness of software. *National Institute of Standards and Technology U.S. Department of Commerce*, November 2010.

[4] Edward Amoroso and Carol Taylor. A process-oriented methodology for assessing and improving software trustworthiness. *Proceedings of the 2nd ACM Conference on Computer and communications security, Virginia, USA, pp. 39-50*, 1994.

[5] The heartbleed bug. http://heartbleed.com/.

[6] Thomas J. McCabe. A complexity measure. *IEE Transactions of Software Engenieering, Vol. SE-2, NO. 4*, 1976.

[7] Hassan Raza Bhatti. Automatic measurement of source code complexity. Master's thesis, Lulea University of Technology.

[8] Steve McConnell. *Code Complete.* Microsoft Press, 2004.

[9] Bev Littlewood. Mtbf is meaningless in software reliability. *Reliability, IEEE Transactions on, vol.24, no.1, pp.82-82*, 1975.

[10] Ross Anderson. Why information security is hard - an economic perspective.

[11] *Trusted Computer System Evaluation Criteria (TCSEC).* United States Department of Defense Standard., 1985.

[12] *Common Criteria for Information Techonology Security Evaluation.*

[13] *Systems Security Engineering Capability Maturity Model (SSE-CMM).* International Systems Security Engineering Association.

[14] Neuhaus S and Zimmermann T et al. Predicting vulnerable software components. *Computer and communications security*, 2007.

[15] Debian Policy Manual. Declaring relationships between packages. https://www.debian.org/doc/debian-policy/ch-relationships.html.

[16] Common vulnerability scoring system v3.0: Specification document. https://www.first.org/cvss/specification-document.

[17] National Institute of Standards and Technology (NIST). National vulnerability datadata (nvd). https://web.nvd.nist.gov/.

[18] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs.

[19] Irshad Ahmad Mir and S.M.K Quadri. Analysis and evaluating security of component-based software development: A security metrics framework. *I. J. Computer Network and Information Security,*, 2012.

[20] Siobhan O'Mahony. The governance of open source initiatives: what does it mean to be community managed? *Springer Science+Business Media B.V. 2007*, 14 June 2007.

[21] Xavier Franch, Angelo Susi, and Maria C. Annosi. Managing risk in open source software adoption.

[22] Eric Steven Raymond. The cathedral and the bazaar.

[23] Ross Gardler and Gabriel Hanganu. Governance models. http://oss-watch.ac.uk/resources/governancemodels.

[24] Thomas Hofer. Evaluating static source code analysis tools. Master's thesis, Ecole Polytechnique Federale de Lausanne, 2007.

[25] Thomas Zimmermann Christian Bird, Tim Menzies. *The Art and Science of Analyzing Software Data*. Elsevier, 2015.

[26] Dan Rathbun. Gathering security metrics and reaping the rewards. *SANS Institute*, 2009.

[27] Microsoft. Definition of a security vulnerability. https://msdn.microsoft.com/en-us/library/cc751383.aspx.

[28] MITRE. Common vulnerabilities and exposures. https://cve.mitre.org/.

[29] rpm. Dependencies. http://www.rpm.org/wiki/PackagerDocs/Dependencies.