Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

A methodology and a platform to measure and assess software windows of vulnerability

Tesi in: Sicurezza delle Reti

Relatore:
Gabriele D'Angelo

Presentata da:
Giacomo Venturini

I Sessione
Anno Accademico 2019-2020

Contents

Introduction

In an ideal world software runs smoothly and without problems, there are no malicious people and no one threatens your daily life, especially your data. However this is just a pipe dream and software vulnerabilities are daily discovered. What could go wrong?
Nowadays, when listening to or reading security related news, it is easy to learn about how wrong policies and security flaws, even among big tech leaders, lead to possible data breaches, usually notifying users of the affected services to update their passwords. Actually behind the scenes a set of practices, activities and operations are put into motion to verify the context severity, examining whether the problem is restricted to a more or less limited area.
Since the price to pay for recovering from an outbreak can be enormous, it can often be useful evaluating the risk of software in terms of its window of vulnerability. A window of vulnerability can be defined as the amount of time a software has been vulnerable to an attack that can compromise the system security. Therefore this metric can be used by corporation to:

- assess the security of software, or more in general systems, in terms of vulnerability response, leading to choose a product over another;
- provide additional value to their software solutions;
- monitor their systems degree of risk;
- have a clear representation of the vulnerabilities of their systems, allowing more deep and precise post-incident forensic analysis, which can be useful to determine legal liabilities.

Unfortunately, defining and evaluating a window of vulnerability is not an easy task: in literature many theoretical models have been proposed, but few were actually implemented and their analysis was limited to a restricted subset of software products. One of the main reasons of this lack is related to the required data, that

is provided by heterogeneous sources in different formats.
Therefore this thesis has dual objective:

- on the one hand, to make its readers aware of the importance of applying security patches and the risks caused by careless behaviours
- on the other, to provide a methodology and a tool to compare systems by measuring the window of vulnerability of their software.

For the implementation step it will be necessary to make a system specific choice, as keeping the approach general is not feasible outside of the theoretical model. Therefore, GNU/Linux systems were chosen, since their recent rise in popularity especially in server and Internet of Things contexts, where they are exposed to an higher risk.
This thesis is structured as follows:

- chapter 1 provides a quick overview of computer security concepts necessary to understand the following chapters;
- chapter 2 defines what is a vulnerability and shows the metrics that come into play when doing a vulnerability related research;
- chapter 3 introduces the window of vulnerability topic, highlighting the importance of active system maintenance;
- chapter 4 shows the security implications caused by the use of third-party dependencies and how package management systems can help reducing the number of vulnerable dependencies in use;
- chapter 5 describes the developed tool, focusing on showing and explaining its architecture design choices and all the other meaningful technical details;
- chapter 6 shows some interesting results obtained by the window of vulnerability analysis;
- chapter 7 draws the conclusions, making a recap of the work done and giving some tips for starting future vulnerability related researches using the developed tool.

Chapter 1

Security Concepts Overview

This chapter gives an overview of basic security concepts necessary to understand the following chapters. Most of the terminology comes from “Computer Security: Principles and Practice” [B1], that will surely give a more detailed and in depth perspective of these concepts for more curious and interested readers.

1.1 Key objectives of computer security

The National Institute of Standards and Technology (NIST) Computer Security Handbook [B2] defines the term computer security as follows:

> The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (including hardware, software, firmware, information/data, and telecommunications).

This definition introduces three key security concepts, often referred as the CIA triad:

- **Confidentiality**: this term covers two related concepts:
  - **Data confidentiality**: assuring that private or confidential information is not made available or disclosed to unauthorized individuals.
  - **Privacy**: assuring that individuals control or influence what information related to them may be collected and stored; and by whom and to whom that information may be disclosed.

- **Integrity**: this term covers two related concepts:
  - **Data integrity**: assuring that information and programs are changed only in a specified and authorized manner.
  - **System integrity**: assuring that a system performs its intended functions in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system.
- **Availability**: assuring that systems work promptly and service is not denied to authorized users.

To have a more complete picture, many security fields also add to these definitions authenticity and accountability concepts:

- **Authenticity**: the property of being genuine and being able to be verified and trusted. This means verifying that users are who they say they are and that each input arriving at the system comes from a trusted source. This property focuses on ensuring the validity of transmissions, messages, or message originators.
- **Accountability**: the requirement for actions of an entity to be traced uniquely to that entity. This supports nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and after-action recovery and legal action. Because truly secure systems are not yet an achievable goal, we must be able to trace a security breach to a responsible party. Systems must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes.

1.2 Computer security terminology

The Internet Security Glossary [S1], defines the following computer security terminology:

- **Adversary**: often also called threat agent, it is an entity that attacks, or is a threat to, a system.

- **Attack**: an assault on system security that derives from an intelligent threat. This is an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system.
- **Countermeasure**: an action, device, procedure, or technique that reduces a threat, a vulnerability, or an attack:
  - by eliminating or preventing it,
  - by minimizing the harm it can cause,
  - or by discovering and reporting it so that corrective actions can be taken.
- **Flaw**: an error in the design, implementation, or operation of an information system. A flaw may result in a vulnerability.
- **Risk**: an expectation of loss expressed as the probability that a particular threat will exploit a particular vulnerability with a particular harmful result.
- **Security Policy**: a set of rules and practices that specify or regulate how a system or an organization provides security services to protect sensitive and critical system resources.
- **System Resource** (Asset): data contained in an information system; or a service provided by a system; or a system capability (i.e processing power or communication bandwidth); or an item of system equipment (i.e. a system component); or a facility that houses system operations and equipment.
- **Threat**: a potential for violation of security, which exists when there is a circumstance, capability, action, or event, that could breach security and cause harm. A threat is a possible danger that might exploit a vulnerability.

The vulnerability definition was intentionally omitted from this terminology since it will be discussed with a greater level of detail in chapter 2.

1.3 System resources typologies

The focus of computer security is to protect system resources. These can be categorized as:

- **Hardware**: computer systems and other data processing, data storage, and data communications devices
- **Software**: operating systems, system utilities, and applications
- **Data**: files and databases, as well as security-related data, such as password files
- **Communication facilities and networks**: local and wide area network communication links, bridges, routers, and so on

1.4 Kinds of attackers

The previous section showed which assets are threatened by an attack. In addition there are also different kinds of attackers, that can be grouped by their knowledge and aims:

- **Hacker**: a person who delights in having an intimate understanding of the internal workings of systems, computers and computer networks. With this deep knowledge, he is able to locate and exploit possible flaws in their design. The term is often used with a negative connotation, where "cracker" would be the correct term instead. As a matter of fact, his intentions are not necessarily malicious, and his discoveries can be employed to achieve more secure systems.
- **Cracker**: an individual who attempts to access computer systems without authorization. These individuals, as opposed to hackers, are usually malicious and have many means at their disposal for breaking into a system.

- **Script kiddie**: an individual who uses scripts or software written by someone else to exploit or break into a computer system. It is a derogatory term, describing someone who uses malicious tools without knowing how they work under the hood or being skilled enough to create them. [S2]

Chapter 2

Vulnerabilities, metrics and involved parties

Nowadays, doing a vulnerability related work, is not a simple matter: actually, it is easy to be overwhelmed and confused by many related acronyms and classification tools. Moreover, to furtherly hinder the task, there is not a unique place where to find the information needed to face the topic clearly.
After a brief introduction on what vulnerabilities are, this chapter goal is to group and describe the most important concepts related to the subject, showing the main available public and free to use sources.

2.1 What is a vulnerability?

In computer security, the "vulnerability" concept has been described with many degrees of detail, that could be appropriate for a specific use case but imprecise or vague for another. For this reason, to have an overall idea of what is a vulnerability, below are reported some useful definitions given by different authorities:

> A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [S1]

> A vulnerability is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source. [S3]

> A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative

> impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety) [S4]

With this definitions in mind, section 2.2 will show a brief historical introduction on what led to the standardization of the tools used nowadays to refer and describe vulnerabilities and their characteristics.

2.2 Identifying and classifying vulnerabilities

From the given vulnerability definitions, it is quite obvious that vulnerabilities differ from each other, not only in terms of affected products but also in terms of exploitation strategies and harmful potential. Before 1999 security tools used proprietary names and different metrics for these classifications: the consequences were potential gaps in security coverage and no effective interoperability among the disparate databases and tools, with difficulties into determining whether different entries were referring to the same problem [S5]. Nowadays these problems are luckily solved thanks to the work started in 1999 by the Massachusetts Institute of Technology Research Establishment (MITRE) Corporation, that led to the following identification and classification factors:

- Common Vulnerabilities and Exposures (CVE)
- Common Platform Enumeration (CPE)
- Common Vulnerability Scoring System (CVSS)
- Common Weakness Enumeration (CWE)

2.3 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a list of common identifiers for publicly known cybersecurity vulnerabilities. CVE is a dictionary rather than a database which provides:

- a unique identifier and a standardized description for each vulnerability or exposure
- a method with which different databases and tools can "speak" the same language
- a way to interoperability and better security coverage
- a basis for evaluation among services, tools, and databases

CVE is nowadays the industry standard for vulnerability and exposure identifiers and it is free for public download and use.

2.3.1 CVE Entries

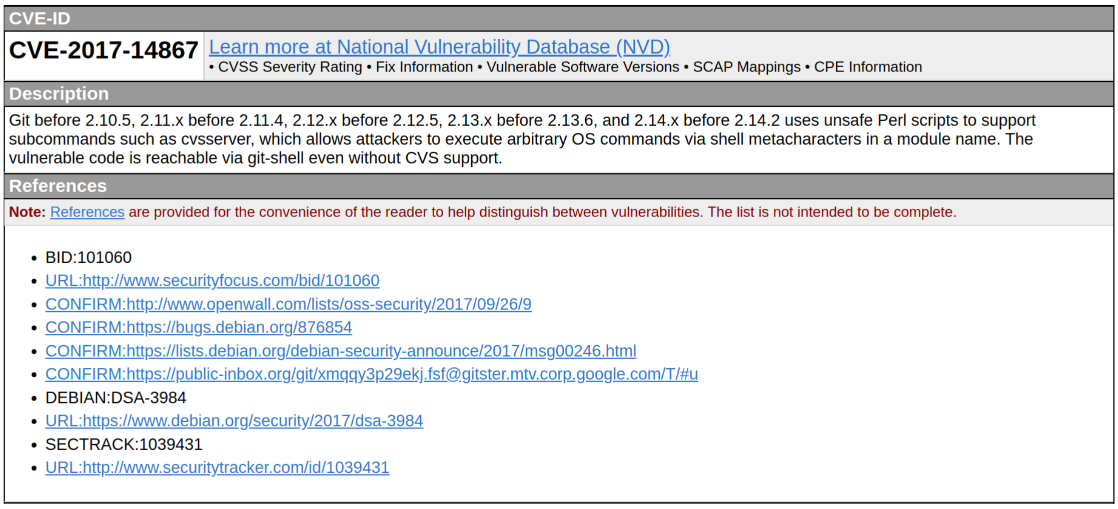
CVE Entries, also called CVEs, CVE IDs, and CVE numbers by the community, provide:

- reference points for data exchange, so that cybersecurity products and services can "speak" with each other
- a baseline for evaluating the coverage of tools and services, so that users can determine which tools are most effective and appropriate for the needs of their organization

An entry is created when a potential security vulnerability is discovered and, as shown in Figure 2.1, contains the following information:

- CVE ID: an identifier composed by the "CVE" prefix followed by an year field and at least 4 digits number (i.e. "CVE-1999-0067", "CVE-2014-10001", "CVE-2014-100001").

- Description: a brief description of the security vulnerability or exposure.
- References: any pertinent references (i.e. vulnerability reports and advisories).



CVE-ID

CVE-2017-14867 | Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Git before 2.10.5, 2.11.x before 2.11.4, 2.12.x before 2.12.5, 2.13.x before 2.13.6, and 2.14.x before 2.14.2 uses unsafe Perl scripts to support subcommands such as cvsserver, which allows attackers to execute arbitrary OS commands via shell metacharacters in a module name. The vulnerable code is reachable via git-shell even without CVS support.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BID:101060
- URL:http://www.securityfocus.com/bid/101060
- CONFIRM:http://www.openwall.com/lists/oss-security/2017/09/26/9
- CONFIRM:https://bugs.debian.org/876854
- CONFIRM:https://lists.debian.org/debian-security-announce/2017/msg00246.html
- CONFIRM:https://public-inbox.org/git/xmqqy3p29ekj.fsf@gitster.mtv.corp.google.com/T/#u
- DEBIAN:DSA-3984
- URL:https://www.debian.org/security/2017/dsa-3984
- SECTRACK:1039431
- URL:http://www.securitytracker.com/id/1039431

Figure 2.1: CVE Entries information displayed on MITRE site.

States of CVE Entries

While analysing the CVE dictionary, it can happen to face entries marked with the following special states:

- **RESERVED**: when the entry has been reserved for use by a CNA or a security researcher, but its details are not yet populated. A CVE Entry can change from the *RESERVED* state to being populated (the normal one) at any time, based on a number of factors both internal and external to the CVE List.
- **DISPUTED**: when there is a disagreement between parties on the assertion that a particular issue in software is a vulnerability. In these cases, further references are provided to better inform those trying to understand the facts of the issue.

- **REJECT**: when the entry is not accepted as a CVE Entry. The reasons for the decision are usually stated in its description. Possible *REJECT* cause examples are duplicates, withdrawal by the original requester or incorrect assignments. As a rule, *REJECT* CVE Entries should be ignored, however there may be cases where such entries might be moved back to *RESERVED* or populated states.

2.3.2 CVE Community

The CVE project is an international cybersecurity community effort. Its success comes from the contributions of CVE Board, CVE Numbering Authorities (CNAs) and the numerous organizations which made their products and services compatible with CVE, and/or adopted or promoted their usage.

CVE Board

The CVE Board includes numerous cybersecurity related organizations such as commercial security tool vendors, academic and research institutions, government departments and agencies, and other prominent security experts, as well as end-users of vulnerability information. Through open and collaborative discussions, the Board provides critical input regarding the data sources, product coverage, coverage goals, operating structure, and strategic direction of the CVE Program.

CVE Numbering Authorities

CNAs are vendors, vulnerability researchers, national and industry Computer Emergency Response Teams (CERTs), and bug bounty programs that assign CVE Entries to newly discovered issues without directly involving the CVE Team in the specific vulnerabilities details.

CNAs are currently organized in the four following categories [B3]:

1. **Sub-CNA**: the most common and basic level of CNA. Each Sub-CNA assigns CVE IDs for vulnerabilities in their own products or their domain of responsibility (called scope). Sub-CNAs also submit vulnerability information to the CVE List when they make a vulnerability public.

2. **Root CNA**: CNA that administers and manages a group of Sub-CNAs within a given domain or community. They are also entrusted of admitting new Sub-CNAs, CNAs-LR, and Root CNAs within their scope.

3. **CNA of Last Resort** (CNA-LR): CNA created by a Root CNA to manage the vulnerabilities of its scope that are not already covered by the other Sub-CNAs.

4. **Program Root CNA**: a special type of Root CNA that oversees the entire CNA Program. It:

    - acts as the final arbiter for all disputes between CNAs and content-related decisions,
    - develops the CNA Rules with approval from the CVE Board,
    - recruits and onboards new CNAs,
    - ensures that all other CNAs are following CNA Rules.

In addition to these categories there is the **Secretariat** role, that supports many of the CNA functions (such as publishing to the CVE List). To better understand the described CNA relationships, Figure 2.2 represents their hierarchical organization.

2.3.3 MITRE's Role

The MITRE Corporation currently maintains the CVE standard and its public website, oversees the CNAs and CVE Board and provides impartial technical guidance throughout the process to ensure CVE serves the public interest. In addition, the MITRE CVE Team currently functions as the CVE Program Root CNA.

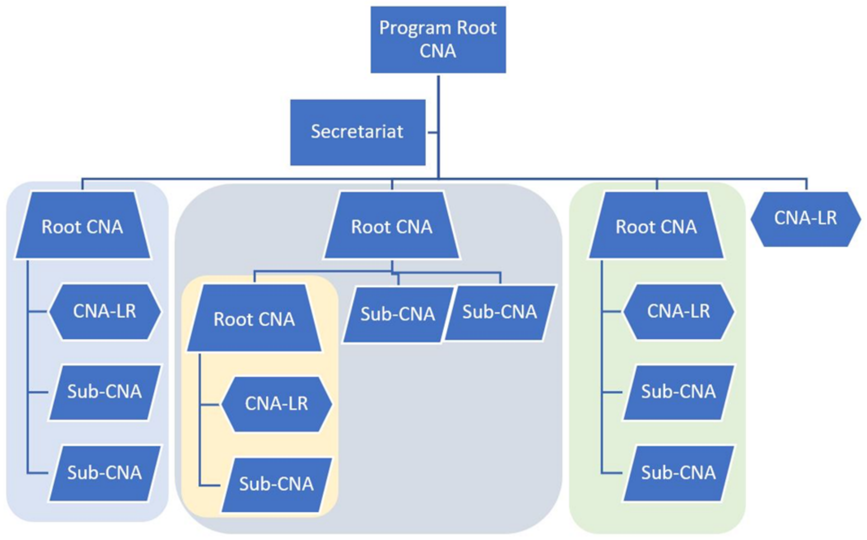


Figure 2.2: CNA hierarchical organization structure. Each colored box indicates a different Root CNA scope. Root CNA sub-hierarchies can be created. (From MITRE CNA Rules)

2.4 Common Platform Enumeration

Common Platform Enumeration (CPE) is a standardized method of describing and identifying classes of applications, operating systems, and hardware devices present among an enterprise's computing assets. CPE identifies abstract classes of products, such as XYZ Visualizer Enterprise Suite 4.2.3, XYZ Visualizer Enterprise Suite (all versions), or XYZ Visualizer (all variations). Identifying products using their CPE names allows IT management tools to collect information about them and then make fully or partially automated decisions regarding the assets.

2.4.1 CPE Specifications

The current version of CPE is 2.3 and it is defined through the following set of specifications in a stack-based model (Figure 2.3):

- **Naming specification**: defines the logical structure of *Well-formed Names* (WFNs), *Uniform Resource Identifier* (URI) and *formatted string* bindings, and the procedures for converting WFNs to and from the bindings. A WFN is a logical construct that constitutes an unordered list of Attribute-Value (A-V) pairs that collectively describe or identify one or more operating system, software application, or hardware device. Unordered means that there is no prescribed sequence in which A-V pairs should be listed, and there is no specified relationship (i.e. hierarchical, set-theoretic) among attributes.

- **Name Matching specification**: defines the procedures for WFNs comparison and so determine whether they refer to the same products.
- **Dictionary specification**: defines the concept of CPE dictionary, which is a repository of CPE names and metadata, with each name identifying a single class of an IT product. It also defines processes for using the dictionary, such as how to search for a particular CPE name or how to look for dictionary entries that belong to a broader product class. Moreover it outlines all the rules that dictionary maintainers must follow when creating or updating its entries.
- **Applicability Language specification**: defines a standardized structure for forming complex logical expressions out of WFNs. These expressions, also known as applicability statements, are used to tag checklists, policies, and other documents with information about the product(s) to which the documents apply. For example, a security checklist for Mozilla Firefox 72.0.2 running on Microsoft Windows 10 could be tagged with a single applicability statement that ensures only systems with both Mozilla Firefox 72.0.2 and Microsoft Windows 10 will have the security checklist applied.

The CPE stack is designed to be open for innovation opportunities and its reference page will be updated as soon as new specifications become available.

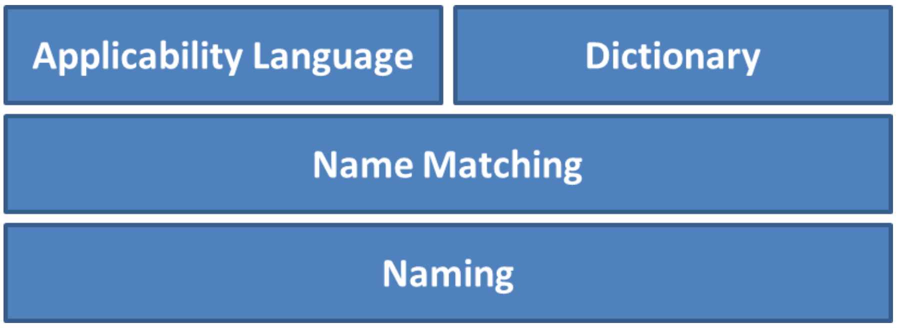


Figure 2.3: The current CPE 2.3 stack. Each higher layer builds on top of the ones below it, so the Naming layer is most fundamental one. (from NVD CPE stack)

Well Formed Names attributes and values

The CPE 2.3 standard [B4] defines the following WFN A-V pairs:

- **part**: this attribute can assume one of these string values:
  - "a": when the WFN is for a class of applications
  - "o": when the WFN is for a class of operating systems
  - "h": when the WFN is for a class of hardware devices
- **vendor**: values for this attribute should describe or identify the person or organization that manufactured or created the product.
- **product**: values for this attribute should describe or identify the most common and recognizable title or name of the product.
- **version**: values for this attribute should be vendor-specific alphanumeric strings characterizing the particular release version of the product.
- **update**: values for this attribute should be vendor-specific alphanumeric strings characterizing the particular update, service pack, or point release of the product.
- **edition**: deprecated attribute, which should assume the logical value ANY except where required for backward compatibility with CPE 2.2 version.

- **language**: values for this attribute shall be valid language tags as defined by Request For Comments 5646 [S6], and should be used to define the language supported in the user interface of the product being described.
- **sw_edition**: values for this attribute should characterize how the product is tailored to a particular market or class of end users.
- **target_sw**: values for this attribute should characterize the software computing environment within which the product operates.
- **target_hw**: values for this attribute should characterize the instruction set architecture (i.e. x86) on which the product being described or identified by the WFN operates.
- **other**: values for this attribute should capture any other general descriptive or identifying information which is vendor or product specific and which does not logically fit in any other attribute value.

Each attribute may be used at most once in a WFN. If an attribute is not used, it is said to be *unspecified* and its value defaults to the logical value ANY.

URI and formatted string formats

URI and formatted string formats are the machine-readable representation of WFNs. With the CPE 2.3 standard the URI format is designed to be backward compatible with prior CPE versions, while the formatted string one (Listing 1) has been introduced to relax the requirements that typically apply to URIs. Both representations bind the attributes of a WFN in a fixed order, separating them by the colon character.

```
cpe:2.3: part : vendor : product : version : update : edition :
    language : sw_edition : target_sw : target_hw : other
```

Listing 1: CPE 2.3 formatted string attributes format.

2.5 Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) is an open framework owned and managed by FIRST [S7], an US-based non-profit organization, whose mission is to help computer security incident response teams across the world. CVSS provides a way to capture the main characteristics of a vulnerability and produce a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations to properly assess and prioritize their vulnerability management processes.
CVSS scores are commonly used:

- for calculating the severity of vulnerabilities discovered on a system,
- as a factor in prioritization of vulnerability remediation activities.

CVSS current version is 3.1 and its scoring system is composed of three metric groups [S8]:

- **Base metric**: produces a score ranging from 0 to 10, which represents the innate characteristics of a vulnerability. This score can then be modified by the Temporal and Environmental metrics.
- **Temporal metric**: changes the score over time due to events external to the vulnerability.
- **Environmental metric**: customizes the score to reflect the vulnerability impact on an organization.

2.6 Common Weakness Enumeration

Common Weakness Enumeration (CWE) is a community-developed list of common software and hardware weakness types that have security ramifications [S9]. "Weaknesses" are defined as flaws, faults, bugs, vulnerabilities, or other errors

in software or hardware implementation, code, design, or architecture that if left unaddressed could result in systems, networks, or hardware being vulnerable to attacks.
The list was initially focused on software weaknesses because organizations of all sizes wanted assurance that the software products they acquire and develop are free of known types of security flaws. However, in 2020 the support was also extended to hardware weaknesses, since the recent rising concern in this kind of security issues (i.e. LoJax [B5], Rowhammer [B6], Meltdown/Spectre [S10][S11][S12]).

2.6.1 CWE objectives

CWE helps developers and security practitioners to:

- describe and discuss software and hardware weaknesses in a common language,
- check for weaknesses in existing software and hardware products,
- evaluate coverage of tools targeting these weaknesses,
- leverage a common baseline standard for weakness identification, mitigation, and prevention efforts,
- prevent software and hardware vulnerabilities prior to deployment.

2.7 National Vulnerability Database role

The National Vulnerability Database (NVD) is the U.S. government repository of vulnerability data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. The NVD includes databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics. Originally created in 2000 (called Internet - Categorization of Attacks Toolkit or ICAT), the NVD has undergone multiple iterations and improvements and, as stated on their website [S13], it will continue to do so to deliver high quality services.

> **Security Content Automation Protocol**
> The Security Content Automation Protocol (SCAP) is a synthesis of interoperable specifications derived from community ideas. Community participation is a great strength for SCAP since it ensures the broadest possible range of use cases is reflected in its functionality.

2.7.1 NVD data

The NVD is the result of the analysis performed on entries published to the MITRE's CVE dictionary. By aggregating information from description, references supplied, and any supplemental public data, the analysis enhances CVE data with:

- impact metrics (Common Vulnerability Scoring System - CVSS),
- vulnerability types (Common Weakness Enumeration - CWE),
- applicability statements (Common Platform Enumeration - CPE),
- as well as other pertinent metadata.

The NVD does not actively perform vulnerability testing and relies on vendors, third party security researchers and vulnerability coordinators to provide the information used to assign these attributes. However, as additional information becomes available or existing one is subject to changes, the NVD endeavors to ensure that the information offered are up to date.
As part of its enhanced information, the NVD site also provides the following advanced searching features:

- by OS,
- by vendor name,

- by product name and/or version number,
- by vulnerability type, severity, related exploit range and impact.

Entries states in NVD

Since NVD entries are the result of many data sources they have different states compared to the MITRE ones:

- **Received**: CVEs which have been recently published to the CVE dictionary and have been received by the NVD.
- **Awaiting Analysis**: CVEs which have been marked for analysis. Normally once in this state the CVEs will be analyzed by the NVD staff within 24 hours.
- **Undergoing Analysis**: CVEs which are currently being analyzed by the NVD staff. This process results in the association of reference link tags, CVSS scores, CWE typologies, and CPE applicability statements.
- **Analyzed**: CVEs which have been analyzed and all their data associations have been made.
- **Modified**: CVEs which have been amended by a source (CVE Primary CNA or another CNA). Analysis data supplied by the NVD may be no longer be accurate due to these changes.
- **Deferred**: CVEs which are not going to be analysed or re-analysed by the NVD team due to resources problems or other concerns.
- **Rejected**: CVEs which have been marked as "REJECT" in the CVE dictionary. These CVEs are in the NVD, but currently do not show up in search results.

Chapter 3

Information Systems and Windows of Vulnerability

Software and information systems inevitably evolve during their lifetime: they introduce new features and add functionalities to improve the overall user experience. During their development, however, it is nearly impossible to keep them bug safe and failure proof. For this reasons, this chapter focus is showing information systems from a vulnerability perspective, defining how they transition between different states and phases. Moreover this chapter will introduce the "Windows of Vulnerability" topic, showing the importance of security activities and the implications of naive attitudes.

3.1 States of information systems

From a vulnerability point of view, information systems transition between hardened, vulnerable, and compromised states during their lifetime [B7]. A system:

- attains a *hardened state* when all security related corrections have been installed
- becomes *vulnerable* when at least one security related correction has not been installed
- enters a *compromised state* when it has been successfully exploited

A system during its lifetime typically oscillates between the hardened and vulnerable states, and, if fortunate, it will never enter the compromised one. Active system management seeks to reduce the total time a system remains in the vulnerable and compromised states.

3.2 Phases of information systems

The vulnerability detection and fix effort changes during the lifetime of an information system. In particular Alhazmi and Malaiya in their research [B8] pointed out how vulnerabilities discovery depends largely on its user base. Their model, called Alhazmi Malaiya Logistic Model (AML), identified the following three phases:

- **learning phase**: during this phase software testers (including hackers and crackers) begin to understand the target system and gather the knowledge needed to break it successfully.

- **linear phase**: when the new system, by attracting a significant number of users, reaches the peak of its popularity. During this phase it begins to face stronger security challenges since the number of vulnerabilities tends to grow linearly. The system will remain in this phase until it starts getting replaced by a newer one.

- **saturation phase**: when the technical support and hence the frequency of update patches will begin to decline. The users start to switch or upgrade to a more modern system. At the same time, attackers start to lose interest in the system, since fewer and fewer targets will be using it.

From a security perspective, the linear phase is the most important one since it is when most of the vulnerabilities will be found.
Even if the AML model refers only to information systems, it can actually be easily applied to software in general, since, in this situation, information systems can just be considered as a complex software able to run other software.

3.3 Window of Vulnerability

The term "window of vulnerability" is labelled in the Oxford Dictionary as "An opportunity to attack something that is at risk (especially as a cold war claim that America's land-based missiles were easy targets for a Soviet first strike)" [S14]. This definition does not differ too much from the computer security one:

> In computer security, a Window of Vulnerability (WoV) — also called Window of Exposure — is a time frame within which a certain software is vulnerable to an attack. This period ranges from the flaw discovery date to the patch installation date.

In reality there is not just a single WoV, but rather the superposition of many Windows of Vulnerability (WoVs). Moreover it is important to underline that not just applications are software but also operating systems. So, considering the bigger picture, systems become vulnerable when just one of their application has an exploitable flaw.

3.3.1 Vulnerability life cycle

During software lifetime it is almost unavoidable the (hopefully unintentional) introduction of security flaws. The process that starts with their discovery and leads to their fix is not straightforward and it is influenced by many factors. In literature, researchers proposed many vulnerability life cycle models, with different abstraction levels and different exploitation risk classifications. However none of these models represented the problem considering also the end system actual patching procedure (that is where the WoV gets actually closed). For this reason in Figure 3.1 is proposed a new model filling the lack. The model, compared to others [B9][B10][B11], does not use a time axis to represent the vulnerability "phases", since it is not well suited to represent their "concurrency".

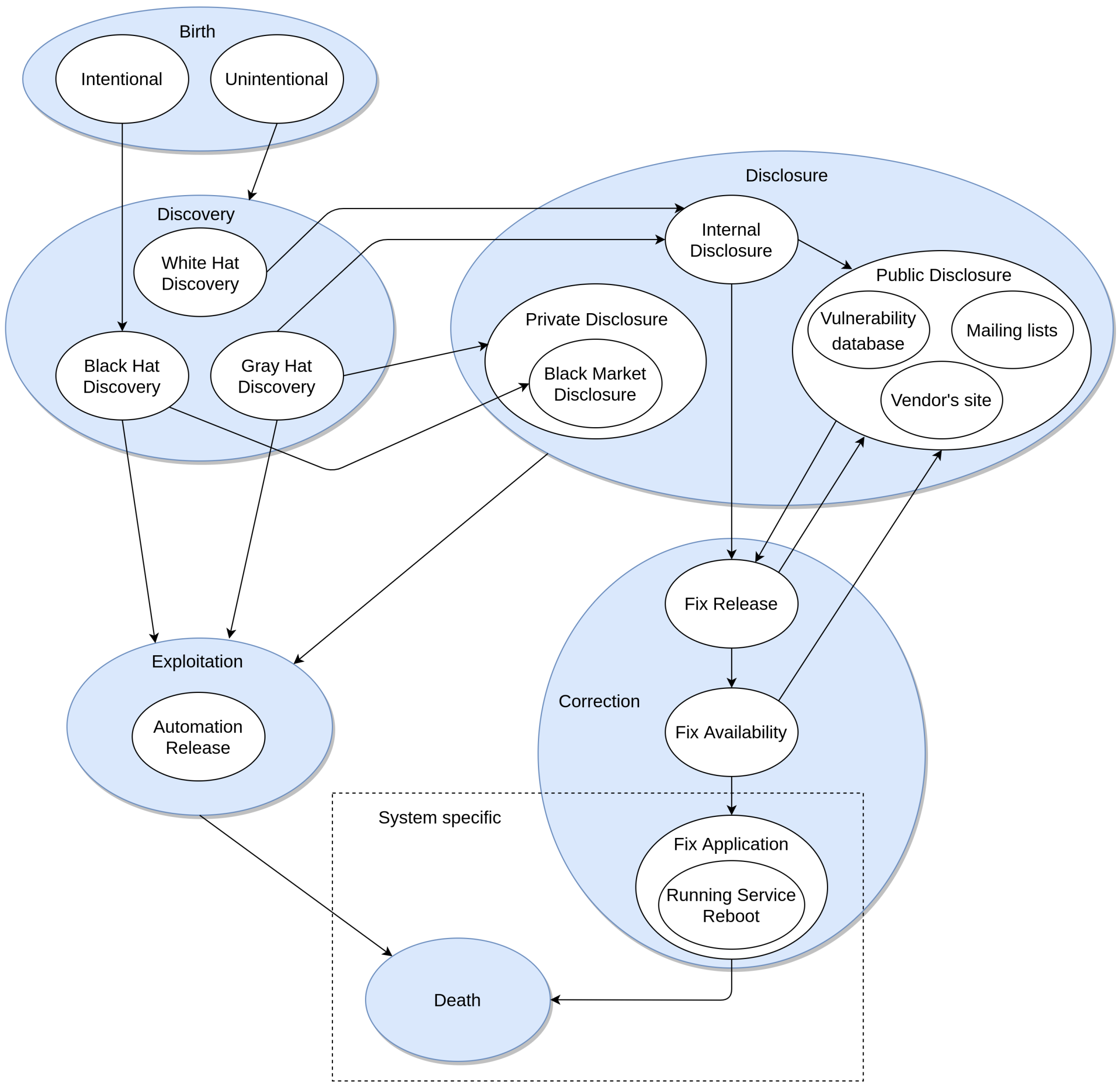


Figure 3.1: Representation of the possible vulnerability life cycle phases. The vulnerability death is represented only considering a specific system, in reality the actual death happens when the number of exploitable systems decreases to insignificance.

The life cycle presents the following phases:

- **Birth**: denotes the flaw's creation. It usually occurs unintentionally. If the birth is malicious and thus intentional, discovery and birth coincide.

- **Discovery**: when someone discovers that a product has security implications, the flaw becomes a vulnerability. It should be pointed out that in this phase only who discovered the vulnerability knows about it. In some cases, original discovery is not an event that can be known: the discoverer may never disclose his finding. Based on whether the discoverer's intentions are malicious or benign, the discovery can be further split into:
  - **Black Hat discovery**: the vulnerability will be used only by its discoverer or it will be sold on black market. The vulnerability will not be revealed to the vendor or to the public.
  - **White Hat discovery**: the vulnerability will be reported (disclosed) to the vendor. Then, after a reasonable amount of time in which the vendor should develop a patch, it will be reported to the public.
  - **Gray Hat discovery**: neutral situation between White Hat and Black Hat discoveries, where the vulnerability can be used just by its discoverer, reported to the involved vendor or sold to a private institution.
- **Disclosure**: the vulnerability and its exploitation details become known to a wider audience. Depending on the discoverer's target, the disclosure audience can be divided into:
  - **private institution disclosure**: the vulnerability details are revealed to private institutions that will not reveal them to the vendor or the public. The vulnerability will be employed for private means and often for malicious purposes.
  - **internal disclosure**: the vulnerability is reported to the vendor, to allow the development and the release of a corrective patch. Not releasing immediately the vulnerability details to a wider audience reduces the number of malicious people that could exploit it. However it should be noted that once the fix is available, the vulnerability details should be publicly disclosed, making the community aware of all the possible implications. Furthermore, a time limit is usually given to the vendor

  before the vulnerability public disclosure, avoiding neglectful attitudes and producing a quicker threat response (fearing a bad press).

  - **public disclosure**: the vulnerability is described on a channel (i.e mailing list[1], vulnerability database[2], vendor's site[3]) where its information and details are freely available to the public.

- **Exploitation**: once the vulnerability becomes known, it can be employed to harm vulnerable systems. Since different vulnerability levels of detail can be revealed at their disclosure, there is a gap between people that can employ them without the proper knowledge. However, once the exploit procedure is automated and released, even those with little or no skill can exploit the vulnerability, increasing therefore the systems at risk. In literature, the exploit automation is often referred also as "scripting"; but, since this term can be misleading, it is not used in this work.

- **Correction**: a vulnerability is correctable when the vendor or a developer releases a patch, a software modification or a configuration change that fixes the underlying flaw. The correction phase is composed by:

  - **fix release**: the date when the fix has been released.
  - **fix availability**: the date when the fix is publicly available for the download for the desired platform. Usually the fix availability happens as soon after the fix release; however, sometimes it could be delayed due to required platform specific tunings. For this reason different platforms could have different fix availability dates.
  - **fix application**: the date when the fix is installed and "actively" working on the system. If the fix concerns a running service, its reboot should be required to make the fix actually take effect.

[1]Debian security announce: https://lists.debian.org/debian-security-announce/
[2]VulDB: https://vuldb.com/
[3]Mozilla Vulnerabilities: https://www.mozilla.org/en-US/security/known-vulnerabilities/

- **Death**: usually, a vulnerability dies when the number of systems it can exploit decreases to insignificance. The death reasons can be:
  - the majority of the systems have installed the vulnerability patch,
  - the vulnerable systems have been retired from the market,
  - the attackers and the media have lost interest in the vulnerability.

  From a system specific perspective, this phase coincides with the fix application.

3.3.2 Zero day vulnerabilities in the life cycle

A zero day vulnerability is a vulnerability that is unknown to those who should be interested in its mitigation. The term zero day refers to the fact that the vendor has "zero days" to fix the problem, since it is already being exploited. In the proposed vulnerability life cycle this corresponds to a Black Hat or Gray Hat discovery followed by a private disclosure.

3.3.3 Reducing the attack chance

From the proposed model appears that users and in particular system administrators are the "passive" element of the chain, since they can only wait for fixes to be available and ready to be applied. Actually good system administrators know that prevention, detection, response and recovery techniques can be implemented while waiting for their publication. Some examples could be using a firewall to identify and deny suspicious network activities or using containers to provide an additional isolation layer between applications and host system.
Since the window of vulnerability grows as more people learn about the vulnerability, there are two available options to make it as small as possible:

- Reducing the window in space: by limiting the number of people who know about the flaw and the information available to the public. Even if this idea could work in theory, in reality it just promotes the "security through obscurity" bad practice:

> The argument that secrecy is good for security is naive, and always worth rebutting. Secrecy is beneficial to security only in limited circumstances, and certainly not with respect to vulnerability or reliability information. Secrets are fragile; once they're lost, they're lost forever. Security that relies on secrecy is also fragile; once secrecy is lost there's no way to recover security. Trying to base security on secrecy is simply bad design. [S15]

  Usually a better approach is reporting the flaw to the vendor and making it public only when the fix is available or a certain time limit is elapsed (i.e. Google's Project Zero research team gives the vendor 90 days to fix).

- Reducing the window in time: by increasing the speed at which vendors patch software and how fast those fixes are installed. However, even if vendors reactively publish patches, the installation process is in the hands of system administrators, that often avoid it fearing breaking behavioural changes.

3.3.4 The dilemma of security patches application

From a security perspective, security patches[4] should always be applied to increase the organization's resilience to attacks. The patch application however is not a straightforward step:

- servers or devices may need to be restarted (causing temporary disservices),
- previously reliable services may encounter errors due to the introduced changes.

For these reasons, it is common practice delaying or totally avoiding security fixes application, especially (and unfortunately not only) among small organizations with limited resources employed in system administration roles. More responsible institutions in fact usually choose to apply at least security patches fixing Important or Critical flaws.

[4]Even if the term "fix" is technically more appropriate, it is common practice using the term "patch" with the same meaning: thus from now on they will be used interchangeably.

Indeed hoping that security breaches never happen is a very cost efficient short term "solution", but it should be remarked that the price to pay for recovering from an outbreak can be enormous:

- Recovery from getting hacked generally implies hosts formatting, operating systems re-installation (with patches applied), restoring data from a previous "sane" backup, applying the remaining patches and perform forensics analysis.
- Recovery from a bad patch may simply be a reinstall, or at least does not involve the cost of dealing with malice.

Given these perspectives, when lacking of specialized staff, outsourcing security functionalities is often a better solution: "security is a process and not a product" [S16].

Chapter 4

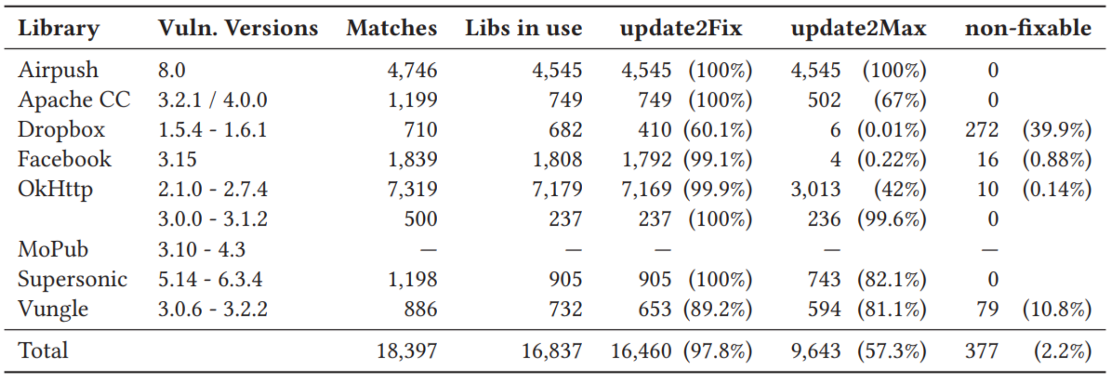
Security concerns in software third-party dependencies

Nowadays, it is common practice for developers to leverage third-party tools, systems and code to make their applications: avoiding reinventing the wheel allows to put the focus on the solution of the main problem and speed up the software release itself. However, the advantage of relying on third-party dependencies unfortunately has security implications, increasing the attack surface of the final products.
This chapter starts with an overview of the possible approaches that can be used to introduce software dependencies into an application, explaining strengths and their related security weaknesses.
After that, it will be shown how package management systems can help reducing the number of vulnerable dependencies in use. To this end, we will describe Debian's update release model, which has been recognized (by the community) among the most reliable ones in terms of system stability and security.
Finally we will discuss the recent direction taken by some Linux distributions, which encourages the usage of system agnostic package management systems.

4.1 Applications and software dependencies

As previously mentioned, any non simple application usually relies on third-party software. A third-party software component, often simply called dependency or library, is a reusable module developed to be either freely distributed or sold by an entity other than the one who uses it [S17]. However, even though its development is up to an external party, the duty to maintain the applications who uses it re-

mains to their developers: so, to the problem of end users not updating software on their devices (discussed in the previous chapter), is added the lack of third-party library updates by developers.
One of the most clear example is the Android market, where different researches highlighted how, even among the top Play Store apps, there were security vulnerabilities concerning outdated third party libraries [B12][B13]. As a proof of matter, Figure 4.1 shows some of their frightening discoveries.



Library	Vuln. Versions	Matches	Libs in use	update2Fix	update2Max	non-fixable
Airpush	8.0	4,746	4,545	4,545 (100%)	4,545 (100%)	0
Apache CC	3.2.1 / 4.0.0	1,199	749	749 (100%)	502 (67%)	0
Dropbox	1.5.4 - 1.6.1	710	682	410 (60.1%)	6 (0.01%)	272 (39.9%)
Facebook	3.15	1,839	1,808	1,792 (99.1%)	4 (0.22%)	16 (0.88%)
OkHttp	2.1.0 - 2.7.4	7,319	7,179	7,169 (99.9%)	3,013 (42%)	10 (0.14%)
	3.0.0 - 3.1.2	500	237	237 (100%)	236 (99.6%)	0
MoPub	3.10 - 4.3	—	—	—	—	—
Supersonic	5.14 - 6.3.4	1,198	905	905 (100%)	743 (82.1%)	0
Vungle	3.0.6 - 3.2.2	886	732	653 (89.2%)	594 (81.1%)	79 (10.8%)
Total		18,397	16,837	16,460 (97.8%)	9,643 (57.3%)	377 (2.2%)

Figure 4.1: Vulnerable libraries found in 18000∼ Android applications. Among those almost, 17000 were actively used, the 97.8% could be upgraded to the first non-vulnerable version without code adaption (update2Fix), the 57.3% could be upgraded to the last available version without code adaption (update2Max) and only the 2.2% could not be upgraded to a fixed version without code modification (non-fixable). (from [B12])

When releasing an application there are three possible ways to introduce the necessary dependencies:

- static linking
- dynamic linking
- dynamic loading

We will now discuss and review the advantages and disadvantages of the approaches.

4.1.1 Static linking

With static linking, all the necessary dependencies are bundled within the application, producing a stand-alone executable. A classic example are "fatjars" or the so called "portable applications".
Straightforward advantages of this approach are:

- the application distribution is simplified, since no additional files need to be present on the system to be able to run it
- it is easier to remove the application from the system, since everything is packaged inside its executable

Disadvantages are:

- the same library can not be shared between different applications, since each one will have its own copy bundled within the executable, leading to disk space waste
- any third-party library update requires the whole program to be recompiled and redistributed

4.1.2 Dynamic linking

Dynamic linking allows storing all the necessary libraries as separate files outside of the executable file. For this reason, when the program is compiled, additional information will be provided to locate the dependencies: in this way, when executing the application, the operating system will know where to find them, correctly loading and linking the required libraries.
As opposed to static linking, this approach allows multiple applications to share the same library without falling into duplicates. Another benefit is that an update in an external library does not require the main application to be recompiled.
The main issues of this approach are:

- libraries corruptions, that will inhibit all the programs that use them
- incompatibilities among different versions of the same library

4.1.3 Dynamic loading

Dynamic loading allows a computer program to start up in the absence of its required libraries. When launched, the program (and not the operating system like in dynamic linking) is responsible for their retrieval and load. The retrieval procedure usually implies dependencies download and caching on the end user device, to avoid wasting bandwidth. Dynamic loading is most frequently used when implementing software plugins, overall allowing more modular applications. The main issues of dynamic loading are:

- it is not supported by every system
- developers are forced to use special constructs to invoke the libraries Application Programming Interface (API), eventually obstructing and/or limiting the natural implementation logic flow

4.1.4 Dependencies strategies overview

From the presented dependency inclusion typologies, it is quite clear that dynamic approaches are superior in terms of third-party library vulnerability patching: as a matter of fact, the original application should not be recompiled and redistributed to fix their security issues:

- with dynamic linking, the affected third-party libraries should be replaced with the fixed ones
- with dynamic loading, the library retrieval procedure should implement a strategy that checks whether more up to date compatible versions are available, eventually downloading them

Unfortunately both approaches have another notable weak point: integrity and authenticity checks need to be performed on the required libraries, to prevent attackers to replace them with compromised ones. While for dynamic loading everything is up to application developers, for dynamic linking package management systems were fortunately born exactly to solve such problems. Next section will give an overview of their working principles, showing the reasons of their success in many Unix distributions.

4.2 Package management systems

A package management system (often simply called package manager) is a tool (usually composed by several layers) that automates installation, upgrade and removal of software, dealt in the form of a package. A package is an archive file containing a program backed up with additional metadata. The metadata typically includes: software description, version, vendor, license, list of required dependencies, and all installation specifications necessary to make it run [B14]. Thanks to these information, package managers can easily solve the previously mentioned dynamic linking problems:

- upon installing a new package, its missing dependencies are downloaded
- dependencies upgrades can be easily installed (and eventually automated), since they are also treated as packages
- dependencies conflicts are promptly identified and showed to the user, which will choose how to solve them (usually by canceling the current installation or by removing the conflicting packages)
- dependencies integrity and authenticity checks are automatically performed, by verifying digital certificates and checksums

To achieve the above tasks, packages need to be maintained in centralized and trusted repositories. Publishing a package in a repository needs to follow certain rules and it can be subject to reviews. Examination policies can be more or less strict and can delay updates releases in favor of ulterior guarantees of not breaking the end user environment.
Usually package managers have a default repository, which can be extended with additional ones, like vendor specific or personal repositories. It should be noted that, for obvious reasons, those repositories will have their own policies, which for

example could grant more up to date software at the expense of system stability. Centralizing software management is a great advantage especially for large enterprises which rely on numerous software, eliminating the need for manual updates installation and simplifying the package flow tracking.

4.2.1 Package Manager Typologies

As previously mentioned, package managers are widely used in Unix systems, where many distributions ship with a default one. Most famous are Debian's Advanced Package Manger (apt) [S18] or RedHat's RedHat Package Manager (RPM) [S19]. However package managers are not a Unix exclusive:

- for systems without a preinstalled one there are usually many unofficial solutions, like OneGet [S20] or Chocolatey [S21] for Windows systems
- developers largely use language specific package managers to build their applications, like JavaScript's Node Package Manager (npm) [S22]

App Stores can also be viewed as a sort of modern package management systems, since they provide a centralized way to install and update systems applications. The main differences are:

- it is not possible to add to the main repository alternative ones
- the dependency management information is hidden, since they were born to be more user friendly

4.3 Debian's effective package management model

The Debian distribution nowadays is known by the community as one of the most stable ones and currently consists in around 30000 packages [B15]. To achieve its stability while maintaining such a large amount of packages, Debian is organized in three different main distributions (often also called suites): unstable, testing and stable.

Unstable, is the active development distribution, where every Debian developer can update their packages at any time. No effort is done to make sure everything is working properly, leading to possible instabilities.
The **testing** distribution is generated automatically by taking packages from unstable if they satisfy certain criteria. This ensures that good quality packages are present within testing. After a period of development, under the guidelines of the release manager, the testing distribution is frozen: the policies which control how packages flow from unstable to testing are tightened, allowing only bug fixes. When the open issues are solved to the satisfaction of the Release Team, a new distribution is released. Upon a release:

- the testing distribution is renamed to **stable**,
- a new testing distribution is created by copying the new stable one,
- the previous stable is renamed to oldstable. Debian's security team tries to support oldstable distribution with security fixes for at least one year, except when another stable distribution is released within the same year [S23].

This development cycle is based on the assumption that the unstable distribution becomes stable after spending a certain amount of time in testing.
Even once a distribution is considered stable, a few bugs inevitably remain and, for this reason, the stable distribution is supplied with the following suites:

- **security updates**: suite used by Debian security team to release security updates. Updates are announced on the debian-security-announce mailing list.
- **stable-updates**: suite to release updates that are not security related, but that should be urgently installed (without waiting for the next Point Release[1]). Possible examples are antivirus databases or timezone-related packages.

[1]Point Releases are stable distribution updates which incorporate fixes of important bugs and security issues. Point Releases usually happen about every two months and update the stable installation image distributed on Debian's website.

- **backports**: suite for packages taken from the testing suite, which are adjusted and recompiled for usage on stable [S24]. Its main purpose is to provide more up to date software than the one available on stable at the expense of stability. Therefore it is recommended to select only the backport version of packages which features are missing in the stable suite.

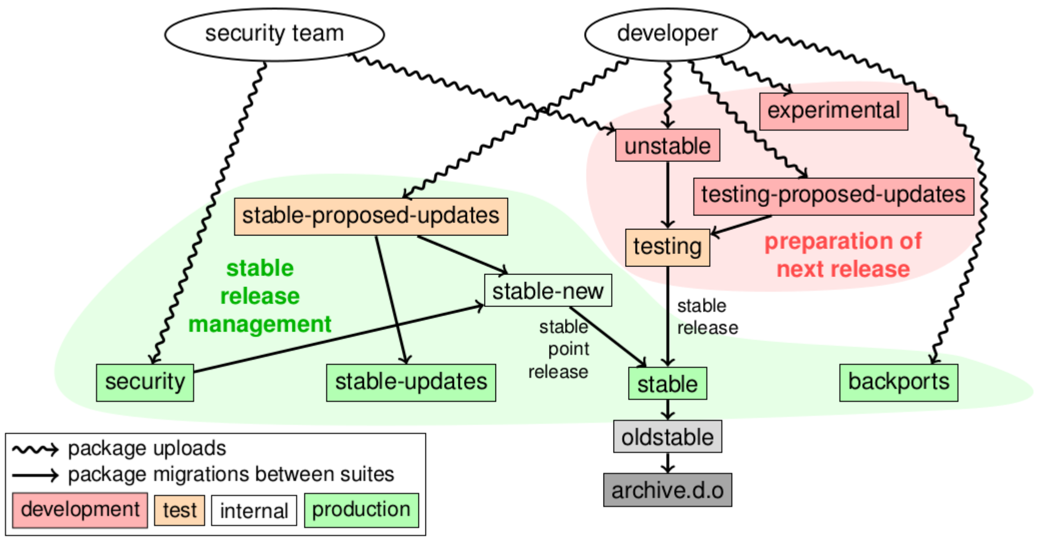


Figure 4.2: Debian package release management. After the current stable suite becomes unstable, the previous unstable is moved to the Debian archive, to serve history purposes. (from Debian Packaging Tutorial [S25])

4.3.1 Security-related bugs handling

Due to their sensitive nature, Debian handles security related bugs very carefully:

> Debian takes security very seriously. We handle all security problems brought to our attention and ensure that they are corrected within a reasonable timeframe. [...] Experience has shown that security through obscurity does not work. Public disclosure allows for more rapid and better solutions to security problems. [S26]

The Debian Security Team exists to coordinate this activity by:

- keeping track of outstanding security problems,
- helping maintainers with security problems or fixing them themselves,
- maintaining Debian Security Tracker,
- sending security advisories.

Debian Security Tracker

Debian Security Tracker is a centralized database that contains all the public information known about security issues: which packages and versions are affected or fixed, and thus whether stable, testing and/or unstable are vulnerable. Information that are still confidential are not added to the tracker.

Security advisories

Security advisories are only issued for stable distribution, excluding therefore testing and unstable ones. When released, advisories are sent to Debian's security announce mailing list[2] and posted on the security web page. Security advisories are written and posted by the security team and include the following information:

- A description of the problem and its scope, including:
  - The type of problem (privilege escalation, denial of service, etc.)
  - What privileges may be gained, and by whom (if any)
  - How it can be exploited
  - Whether it is remotely or locally exploitable
  - How the problem was fixed

  allowing users to assess the threat to their systems.
- Version numbers of affected packages

[2] debian-security-announce@lists.debian.org

- Version numbers of fixed packages
- Information on where to obtain the updated packages (usually from Debian security suite)
- References to upstream advisories, CVE identifiers, and any other information useful in cross-referencing the vulnerability

4.4 Agnostic alternatives to platform specific package managers

GNU/Linux operating system has experienced a continuous growth in the last years, especially in server and Internet of Things (IoT) contexts. Since hundred of different Linux distributions exist, being bound to their default platform specific package manager could not be feasible for certain working environments: for example a small company could not afford to make its products compatible with all the main Linux distributions and deploy them to their respective repositories. To solve this problem system agnostic tools were born, unifying application releases for different environments.
Nowadays the most famous formats are AppImage, Flatpak and Snap which share, each with their own strategies, the possibility to install[3] applications without the need to grant them root access. This feature is particularly useful to:

- prevent users having administrator privileges, consequently exposing systems to minor risks
- allow users to customize the systems with apps for their needs

Of course everything is not granted without paying a price: to have a format compatible with every system, third-party dependencies are bundled within the application, inheriting the previously discussed static linking problems.

[3]It should be noted that AppImage applications do not actually require any installation and can be just run wherever they are.

Chapter 5

A model to compare Windows of Vulnerability

When choosing to base a business model on an operating system, vulnerability patching responsiveness is as much important as factors like system stability and software availability. For this reason, I developed DiVulker, a platform that is able to track Windows of Vulnerability in GNU/Linux software distributed via package managers. This chapter shows at first the requirements needed for its realization, and then it will explain its design choices and all the meaningful technical details.

5.1 Requirement analysis

For developing this kind of tool it is not feasible thinking to have a unique vulnerability data source: in fact, it is quite unlikely to find an institution that is able to keep track of every GNU/Linux system vulnerability. Moreover, even if such institution would exist, having more unrelated vulnerability data sources is usually a better practice for:

- reducing the number of possible errors in the provided data
- avoiding possible favoritism, leading to inaccurate and unfair statistics

achieving an overall better reliability.
Regarding vulnerability data, the following information are required:

- unique identifiers: to be able to match and merge data coming from different sources (i.e. CVE IDs)
- date of public disclosure: used to mark the start of the WoV

- date of fix availability: used to mark the closure of the WoV
- affected package(s): a vulnerability could affect more than one package, moreover a package could be vulnerable only on a certain platform

In addition to these requirements, it could also be useful, but not mandatory, that vulnerability data sources had the support for:

- bulk downloads of their data, speeding up the retrieval procedure
- providing their data in machine readable formats, like JavaScript Object Notation (JSON) or Extensible Markup Language (XML), reducing the effort with creating ad-hoc data parsers
- web feed or publish-subscribe like systems, avoiding polling based strategies to retrieve data updates

5.1.1 Vulnerability data sources selection

To understand if the system realization was possible, I first conduced an assessment analysis to discover if there were enough data sources available to make comparisons at least between two different distributions. The obtained results were not the most reassuring: many of the most used distributions do not have a CVE tracker and their security advisories[1] are not available for a bulk download. Even worse, some distributions do not even issue security advisories (or some other method to track security fix releases): for instance Alpine Linux, which is a distribution widely used in container environments, despite declaring in its motto being security-oriented, dismissed its security mailing list at least from June 2017 [S27].

> [...] We don't issue our own advisories if that's what you mean. That would require more man power which I think we prefer to spend on fixing the security issues. [S28]

[1]Security advisories are one of the main sources used by Unix distributions to notify their users of security fixes. They provide information on when the fix is available for download and which packages and platforms are affected.

Fortunately Debian and Red Hat offer documented and CVE compatible vulnerability data of their products, fulfilling the previously identified fundamental requirements and allowing matches with MITRE and NVD vulnerability lists.

5.2 Architecture design

To allow the system to be scalable, modular and as much as possible technology independent, I chose to realize it using a microservice based architecture. In this way:

- the different microservices which compose the system can be dislocated on many devices, allowing to distribute the computational load and the resource usage
- each microservice is designed to be independent and can be easily plugged in/out without restarting the entire system
- a microservices subset can be reused outside of DiVulker, reducing the amount of work necessary in future vulnerability related researches
- future additions and extensions are not bound to the implementation language used to develop the system, since they only need to adapt to the exchanged messages format

5.2.1 Microservices

DiVulker is composed by four main microservices typologies:

- **Services Registry** (SR): microservice used to keep track of the other entities of the system. At startup, the other microservices will send to the SR all the information necessary for their identification.
- **Data Retriever** (DR): microservice which retrieves and parses publicly available vulnerability data, storing them in its knowledge base. There are two kinds of DR:

  - **Distro Specific Data Retriever** (DSDR): microservice which data are specific to a certain distro
  - **General Data Retriever** (GDR): microservice which data are common to more distros

- **Data Aggregator** (DA): microservice that retrieves data from the other microservices, matching and merging them to obtain more complete vulnerability data. To know how to contact the other microservices a DA will first interact with SR microservice to obtain such information. There are two kinds of DA:
  - **Distro Data Aggregator** (DDA): microservice that retrieves data only from DSDR microservices related to a specific distro
  - **General Data Aggregator** (GDA): microservice that retrieves the data from DDA and GDR microservices

- **Data Comparator** (DC): microservice that retrieves vulnerability data from GDA microservices and offers methods to obtain useful statistics and make comparisons between different distros

From the given perspective, it can be noticed that, except for SR microservice, the others (called from now on Vulnerability Data Services) act both as a client and a data host:

- when acting as client, the service will retrieve the vulnerability related data from a source, parsing only the useful information

- when acting as data host, the service will act as data source, sending its data to anyone who requests it

To better understand how the microservices relationships are organized Figure 5.1 gives a basic (only one GDA is used) system configuration.

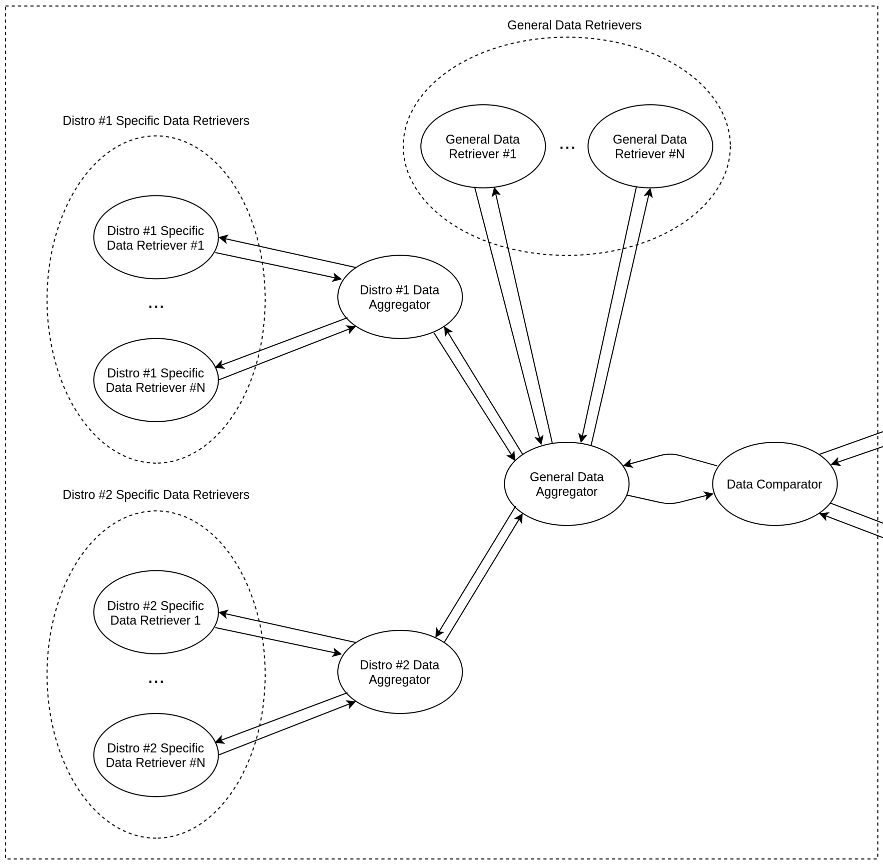

General Data Retrievers
General Data Retriever #1
...
General Data Retriever #N
Distro #1 Specific Data Retrievers
Distro #1 Specific Data Retriever #1
...
Distro #1 Specific Data Retriever #N
Distro #1 Data Aggregator
General Data Aggregator
Data Comparator
Client
Client #1
...
Client #N
Distro #2 Specific Data Retrievers
Distro #2 Specific Data Retriever 1
...
Distro #2 Specific Data Retriever #N
Distro #2 Data Aggregator

Figure 5.1: DiVulker microservice architecture. For readability reasons, the Sources Registry microservice, the external data sources and all the communications involving them are not represented.

5.3 Detailed design

This section will show the relevant design choices made to achieve the previously described architecture.

5.3.1 Vulnerability Data Services interactions

To be able to reuse DiVulker microservices even outside of the system, each Vulnerability Data Service (VDS) is designed to be a standalone data source, that will send its data to anyone who requests them (called from now on subscriber).

For obvious reasons, when using the "standalone" term, it is excluded the data retrieval procedure, in which the services necessary depend on an external source. The data is sent across the system via WebSockets:

- granting the possibility to notify each VDS subscriber when updates or new vulnerability data are available
- avoiding a VDS to know anything about how his subscribers are implemented and consequentially force some kind of API

5.3.2 Vulnerability Data Services hierarchy

To speed up the VDSs realization and to apply the Don't Repeat Yourself (DRY) principle, I designed the hierarchy structure represented in Figure 5.2. In this way, new microservices can be added to the system saving time, just by defining:

- the info sent to the registry to uniquely identify the service
- the necessary models to store the data
- the strategy used to retrieve and update the data, for DR microservices
- the microservices to which connect to and how to merge their data, for DA microservices

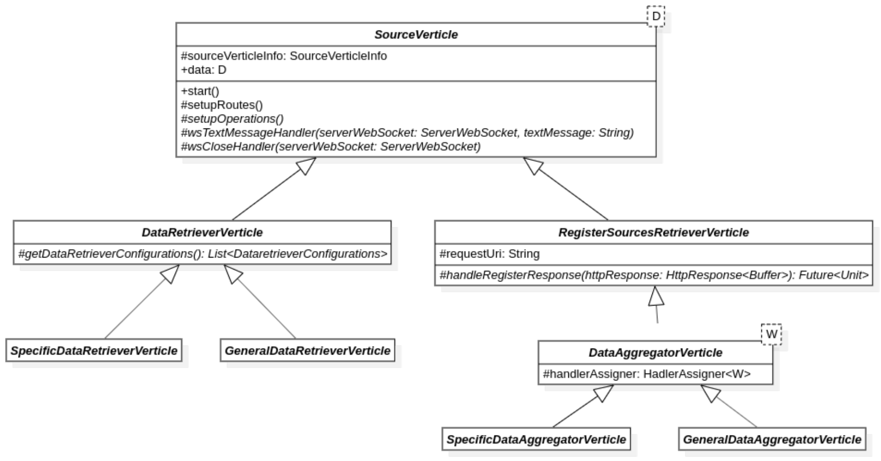

D
SourceVerticle
#sourceVerticleInfo: SourceVerticleInfo
+data: D
+start()
#setupRoutes()
#setupOperations()
#wsTextMessageHandler(serverWebSocket: ServerWebSocket, textMessage: String)
#wsCloseHandler(serverWebSocket: ServerWebSocket)
DataRetrieverVerticle
#getDataRetrieverConfigurations(): List<DataretrieverConfigurations>
RegisterSourcesRetrieverVerticle
#requestUri: String
#handleRegisterResponse(httpResponse: HttpResponse<Buffer>): Future<Unit>
SpecificDataRetrieverVerticle
GeneralDataRetrieverVerticle
W
DataAggregatorVerticle
#handlerAssigner: HadlerAssigner<W>
SpecificDataAggregatorVerticle
GeneralDataAggregatorVerticle

Figure 5.2: Vulnerability Data Service hierarchy.

5.3.3 Data Aggregators: messages and data handling

Since each source is designed to be independent, each Data Aggregator should:

- know how to interact with the host sources to retrieve their data
- define how to handle the received data

Figure 5.3 shows the interface hierarchy to implement each time a Data Aggregator needs to subscribe to a new microservice and handle its data:

- *SpecificDataSourceWsHander* is used by Specific Data Aggregators for handling Specific Retriever data
- *GeneralRetrieverWsHander* is used by General Data Aggregators for handling General Retriever data
- *SpecificAggregatorWsHander* is used by General Data Aggregators for handling Specific Data Aggregators data

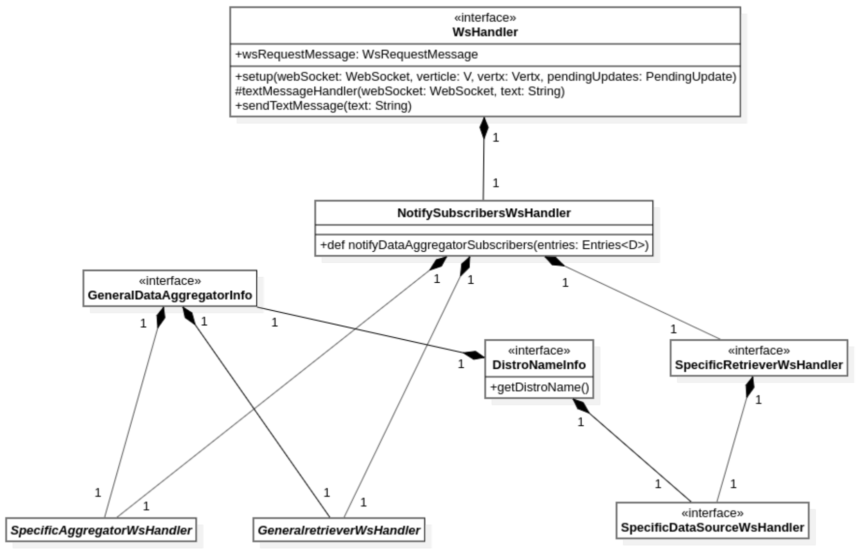

«interface»
WsHandler
+wsRequestMessage: WsRequestMessage
+setup(webSocket: WebSocket, verticle: V, vertx: Vertx, pendingUpdates: PendingUpdate)
#textMessageHandler(webSocket: WebSocket, text: String)
+sendTextMessage(text: String)
NotifySubscribersWsHandler
+def notifyDataAggregatorSubscribers(entries: Entries<D>)
«interface»
GeneralDataAggregatorInfo
«interface»
DistroNameInfo
+getDistroName()
«interface»
SpecificRetrieverWsHandler
SpecificAggregatorWsHandler
GeneralretrieverWsHandler
«interface»
SpecificDataSourceWsHandler
1

Figure 5.3: Data Aggregator WebSocket handlers. SpecificAggregatorWsHandler, GeneralRetrieverWsHandler and SpecificDataSourceWsHandler abstract classes implement most of the interfaces methods reducing the effort needed to create a new WsHandler.

5.3.4 Data retrieval procedure

Each Data Retriever microservice needs to retrieve its data from one or more external source before being ready to send anything to its subscribers. This procedure could take quite long time, especially for General Retrievers which need to download and parse a large amount of data. The activity diagram in Figure 5.4 shows the strategy adopted to speedup services setup when frequently updating and testing their functionalities.

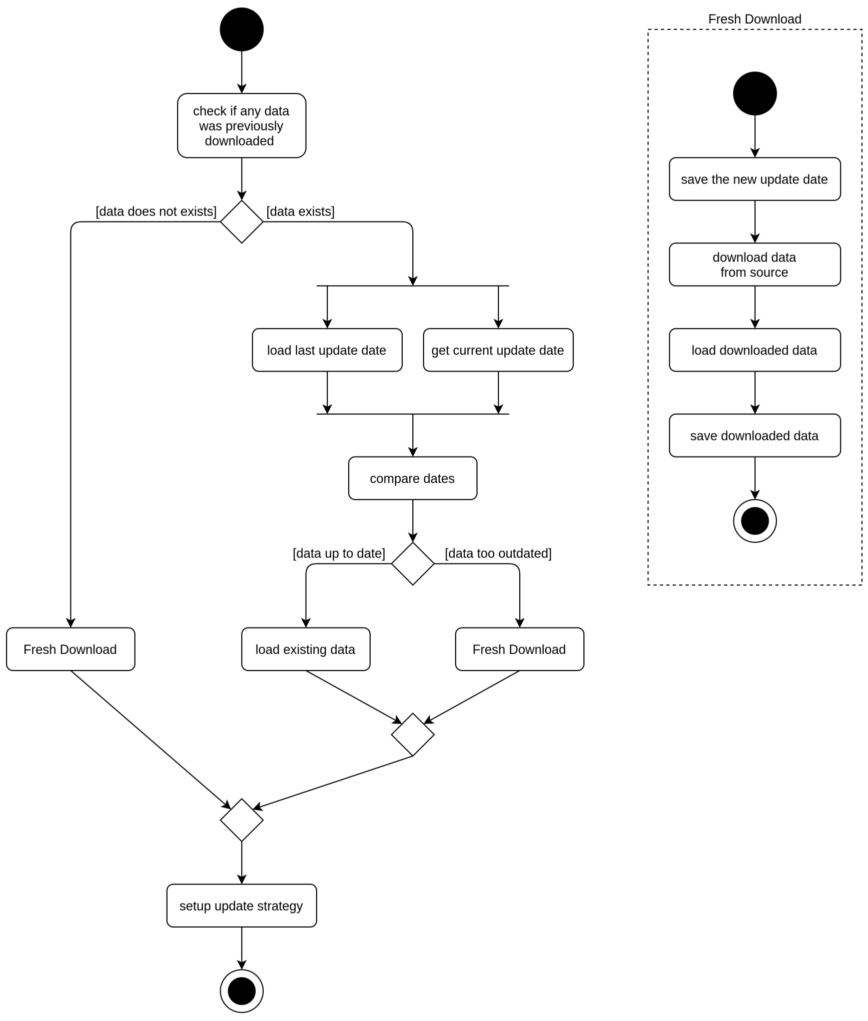


Figure 5.4: Activity diagram that shows the data retrieval procedure.

This procedure takes advantage of a local cache, which can be stored on the filesystem or on an internal database. In this way:

- the DR avoids to contact and download the data from an external source
- the parsing effort is reduced, since the vulnerability data of the local cache is already in the required format (sources could provide raw data) and contains only the meaningful information (often sources provide more data than the required one)

Both "download data from source" and "setup update strategy" activity nodes implementation depend on how the external source makes its data available: for sources that provide them through their website, the polling approach is the only one available.

5.4 Implementation

This section will give a brief overview of the technologies used to build DiVulker as well as showing significant details and aspects on the microservices realization.

5.4.1 Technologies

DiVulker was implemented with the following technologies:

- **Eclipse Vert.x**: Eclipse Vert.x, or often simply called Vert.x, is a polyglot event-driven and non blocking application framework that runs on the Java Virtual Machine (JVM). It is designed to let applications scale with minimal hardware requirements. Vert.x can be used with multiple languages (Java, Kotlin, JavaScript, Groovy, Ruby and Scala), allowing teams to choose the one more suitable for a specific task. Moreover it is very flexible and, by supporting many different messaging protocols out of the box, allows to easily build HTTP/REST microservices based applications.

- **Scala**: Scala is a general purpose programming language that combines object-oriented and functional programming in one concise, high-level language. Being concise is one of the main focuses of the language, speeding up the overall development and allowing external readers to better understand the solution proposed. Other strengths of the language are:

  - **static typing**: that helps avoiding bugs when building complex applications
  - **seamless Java interoperability**: that allows Java and Scala stacks to be freely mixed and easily access to huge ecosystems of libraries
  - **pattern matching**: a mechanism for checking a value against a pattern that provides a great abstraction to write algorithms, improves readability and makes easier to statically intercept bugs
  - **flexibility**: by combining interfaces and classes, allowing complex hierarchies

- **Jackson**: Jackson is a suite of data-processing tools for Java (and the JVM platform). The project was originally designed as a JSON parsing library, but now supports many more data formats (like XML, CSV and YAML). Its usage is built around annotations, that allow to quickly define which fields need to be serialized/deserialized without the need to implement custom parsers.

- **Gradle Build Tool**: Gradle is an open-source build automation tool focused on flexibility and performance. Gradle allows to configure projects on fresh environments without depending on specific Integrated Development Environment (IDE) plugins, ensuring that all the required dependencies are correctly downloaded. Moreover it allows to create complex project dependency hierarchies, especially useful to build modular applications based on custom libraries.

5.4.2 Vulnerability Identifiers and Data Structure

Vulnerabilities across the system are identified by their CVE ID. In addition to this identifier, Specific Aggregators could use other means to associate a vulnerability related info to the correct one: for example, distros usually associate each CVE ID to a custom security advisory ID.
Since the amount of data to store tends to grow very quickly especially in the last microservices of the system, it was necessary to adopt a data structure that is able to quickly locate the entries and speedup update operations. For this reason data is structured similarly to how it is organized on *CVEproject/cvelist* GitHub repo [S29], where each vulnerability is categorized using its CVE ID, by first using the year portion and then by truncating the id portion by 1000. Listing 2 shows a YAML formatted example on how CVE-1999-0001, CVE-1999-1044 and CVE-2020-13458 would be inserted inside the proposed data structure.

```yaml
"1999":
  "0xxx": [
    {id: CVE-1999-0001},
    #...
  ]
  "1xxx": [
    #...
    {id: CVE-1999-1044},
    #...
  ]
  #...

#...
"2020":
  #...
  "13xxx": [
    #...
    {id: CVE-2020-13458},
    #...
  ]
  #...
```

Listing 2: Data structure used to memorize vulnerabilities that uses CVE IDs as identifiers.

5.4.3 General Retrievers

The implemented General Retrievers use data publicly available on MITRE [S30] and NVD [S31] sites. This two sources are used to mainly provide the date of first disclosure of vulnerabilities. Even if the NVD data could technically be enough to have the necessary vulnerability information, I still decided to also use the ones provided on the MITRE site for the following reasons:

- the NVD data partially come from the MITRE CVE list and it often happens that such data falls out of sync
- implementing more than one General Retriever microservice helped with the design more generic microservices, simplifying the addition of future ones

Unfortunately both sources do not provide a way to be notified only when new entries are added or updated. For this reason, all the data need to be periodically downloaded. However the NVD provides a "META" file which stores information on when the provided data was lastly updated, useful to avoid pointless downloads.

5.4.4 Debian Retrievers

Vulnerability data for Debian comes from two sources:

- the Debian Security Tracker [S32]
- the snapshot archive [S33]

Debian Security Tracker

Debian Security Tracker is a Git repository containing vulnerabilities information of Debian distribution and scripts to generate reports. The files used by Debian Retriever microservices are:

- **CVE/list** to identify which vulnerabilities affect or affected Debian
- **DSA/list** to identify when a Debian Security Advisory (DSA) was released to notify the community of fixes related to one or more security issues

For my research I focused only on Debian stable release, therefore I excluded **DLA/list** file from the analysis, since its entries target Debian Long Term Support (LTS) release. Listing 3 and Listing 4 show some examples on how the vulnerabilities data are stored in those files.

```
CVE-2019-14287 (In Sudo before 1.8.28, an attacker with access ...)
    {DSA-4543-1 DLA-1964-1}
    - sudo 1.8.27-1.1 (bug #942322)
...
CVE-2014-9680 (sudo before 1.8.12 does not ensure that the TZ ...)
        {DSA-3167-1 DLA-160-1}
        - sudo 1.8.12-1 (bug #772707)
        [jessie] - sudo 1.8.10p3-1+deb8u2
...
CVE-2011-0010 (check.c in sudo 1.7.x before 1.7.4p5, when ...)
        - sudo 1.7.4p4-6 (bug #609641)
        [lenny] - sudo <not-affected> (Only affects 1.7.x)
        [squeeze] - sudo 1.7.4p4-2.squeeze.1
```

Listing 3: CVE/list entries.

```
[14 Oct 2019] DSA-4543-1 sudo - security update
        {CVE-2019-14287}
        [stretch] - sudo 1.8.19p1-2.1+deb9u1
        [buster] - sudo 1.8.27-1+deb10u1
...
[22 Feb 2015] DSA-3167-1 sudo - security update
        {CVE-2014-9680}
        [wheezy] - sudo 1.8.5p2-1+nmu2
```

Listing 4: DSA/list entries.

The information merging (performed by Debian Aggregator microservice) turned out to be an hard task due to many inconsistencies and missing information, visible in the proposed entries:

- CVE/list marks CVE-2014-9680 as affecting "Jessie" release, while DSA/list DSA-3167-1 marks it affecting "Wheezy".
- CVE/list CVE-2014-9680 cross-references two advisories IDs, DSA-3167-1 issued for stable release, DLA-160-1 for oldstable one, but only "Jessie" release is listed in its fields.
- CVE/list CVE-2019-14287 cross-references DSA-4543-1 and DLA-1964-1 but no affected stable or oldstable release. Usually, when no distribution name is referenced in a CVE/list entry, it is because the issue affected only the unstable release: however, DSA/list DSA-4543-1 shows that both "Stretch" and "Buster" were actually affected.
- some CVEs, like CVE/list CVE-2011-0010, do not present any security advisory and so the date information of their fix is lost.

Other secondary problems are:

- the format requires non simple and error prone regex parsing, especially for CVE/list file
- CVE/list tracks every security issue, even those not related to Debian. This implies undesired data downloads that must be later filtered out

Snapshot archive

The snapshot archive is a wayback machine that allows access to old packages based on dates and version numbers. Its usage was necessary to fill the lack of the fix availability date information for Debian CVEs without a DSA: the service, among all the stored information, provides the first date a certain package was seen in Debian stable or security repositories.

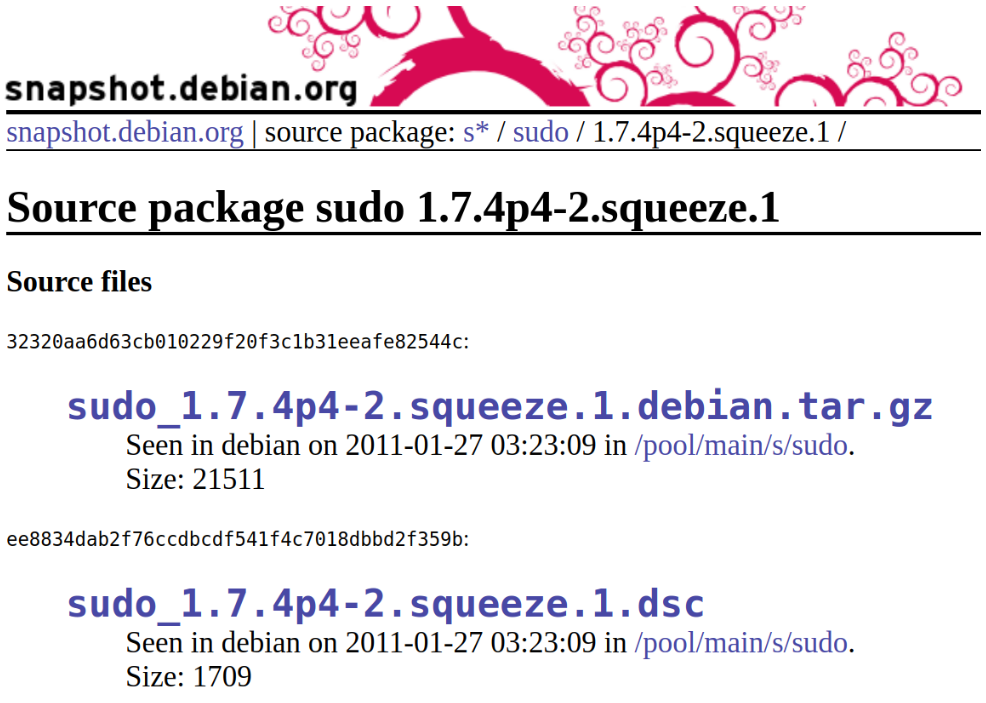
The main problem encountered with accessing to its data is that they are not available for bulk downloads. For this reason:

- a lot of HTTP requests are required to obtain all the missing data
- the snapshot archive refuses many connection attempts, since it is accessed by the same host many times in a short timeframe, slowing furtherly down the data retrieval

To retrieve the "first seen" information of each vulnerable package, the following REST APIs can be used:

1. **/mr/package/<package>/**: lists all the available source versions of the specified package. Having this information avoids requesting for package versions not present in the snapshot archive.
2. **/mr/package/<package>/<version>/allfiles**: lists all the files associated to the specified version of a package. Within this information there is the hash needed to perform the last query.
3. **/mr/file/<hash>/info**: returns the information of the file with the specified hash. Among this information there is the "first_seen" field used as first fix availability date.

Actually, this process has been furtherly improved by directly requesting and parsing the HTML page of a package version at /package/<package>/<version>/, avoiding the request necessary to retrieve the package hash. Figure 5.5 shows the (rendered) page to parse to obtain the release date of sudo 1.7.4p4-2.squeeze.1 which was missing for CVE-2011-0010.



snapshot.debian.org | source package: s* / sudo / 1.7.4p4-2.squeeze.1 /

Source package sudo 1.7.4p4-2.squeeze.1

Source files

32320aa6d63cb010229f20f3c1b31eeafe82544c:

sudo_1.7.4p4-2.squeeze.1.debian.tar.gz
Seen in debian on 2011-01-27 03:23:09 in /pool/main/s/sudo.
Size: 21511

ee8834dab2f76ccdbcdf541f4c7018dbbd2f359b:

sudo_1.7.4p4-2.squeeze.1.dsc
Seen in debian on 2011-01-27 03:23:09 in /pool/main/s/sudo.
Size: 1709

Figure 5.5: Snapshot info page for **1.7.4p4-2.squeeze.1** version of **sudo** source package.

5.4.5 Red Hat Retrievers

Vulnerability data for Red Hat comes from their Security Data page [S34]. More precisely Red Hat Retrievers use the following files:

- **cve_dates.txt**: file that lists all the CVE IDs that affect/affected Red Hat products
- **release_dates.txt**: file that lists every Red Hat Security Advisory (RHSA), Red Hat Bug Advisory (RHBA) and Red Hat Enhancement Advisory (RHEA), providing the information on the date of when a fix is available
- **rhsamapcpe.txt**: file that maps Red Hat Advisories with the related CVE IDs, products and packages

Listing 5, Listing 6, and Listing 7 show some examples of how the files are structured. Even if once again the files use a custom plain text format, in comparison to the one used by Debian, it is so much easier to identify and extract the data of interest since:

- each entry is stored in a single line
- a CPE URI format is used to track affected packages and products
- no data duplication is present in the provided files, except for CVE and advisories IDs which are necessary to perform mappings among the entries

```
CVE-2011-0010 impact=low,public=20110111,reported=20110111,
↪  source=debian,cvss2=1.2/AV:L/AC:H/Au:N/C:N/I:P/A:N
...
CVE-2014-9680 impact=moderate,public=20141016,reported=20150210,
↪  source=gentoo,cvss2=3.0/AV:L/AC:M/Au:S/C:N/I:P/A:P
...
CVE-2019-14287 impact=important,public=20191014:1500,
↪  reported=20191010,source=distros,
↪  cvss3=7.0/CVSS:3.0/AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:H/A:H,
↪  impact(RHBA-2019:3248)=moderate
```

Listing 5: cve_dates.txt entries.

```
RHBA-2015:2424 20151119:0621 (cdn)
...
RHSA-2012:0309 20120221:0220 (rhn)
...
RHSA-2015:1409 20150720:1406 (cdn)
...
RHSA-2020:0388 20200204:1256 (cdn)
```

Listing 6: release_dates.txt entries.

```
RHBA-2015:2424 CVE-2014-9680
↪  cpe:/o:redhat:enterprise_linux:7::client/sudo,
↪  cpe:/o:redhat:enterprise_linux:7::computenode/sudo,
↪  cpe:/o:redhat:enterprise_linux:7::server/sudo,
↪  cpe:/o:redhat:enterprise_linux:7::workstation/sudo
...
RHSA-2012:0309 CVE-2011-0010
↪  cpe:/o:redhat:enterprise_linux:5::client/sudo,
↪  cpe:/o:redhat:enterprise_linux:5::server/sudo
...
RHSA-2015:1409 CVE-2014-9680
↪  cpe:/o:redhat:enterprise_linux:6::client/sudo,
↪  cpe:/o:redhat:enterprise_linux:6::computenode/sudo,
↪  cpe:/o:redhat:enterprise_linux:6::server/sudo,
↪  cpe:/o:redhat:enterprise_linux:6::workstation/sudo
...
RHSA-2020:0388 CVE-2019-14287
↪  cpe:/o:redhat:rhel_e4s:8.0::baseos/sudo
```

Listing 7: rhsamapcpe.txt entries.

Red Hat Errata (RHSA, RHBA, and RHEA)

Red Hat Errata, often called advisories or errata advisories, help users determine what updates are available and how important they are based on analysis performed by Red Hat engineering [S35]. These advisories come in three types:

- **Red Hat Security Advisory (RHSA)**: RHSAs contain one or more security fixes and might also contain bug or enhancements fixes. RHSAs are ranked using a severity rating (Low, Moderate, Important, or Critical) based on the severity of the vulnerability.

- **Red Hat Bug Advisory (RHBA)**: RHBAs always contain one or more bug fixes and might contain enhancements, but do not contain security fixes.
- **Red Hat Enhancement Advisory (RHEA)**: RHEAs contain one or more enhancements or new features and do not contain bug fixes or security fixes. Essentially, a RHEA is released when new features are added and an updated package is shipped.

From the given definitions only RHSAs should be considered for security related work. However, sometimes RHEAs or RHBAs (like RHBA-2015:2424) can also address a security flaw: this happens when, after their issue, it is discovered that they also fix vulnerabilities of previous software version(s). So, to avoid confusion determined by sudden ID changes, the advisories are not re-labelled as RHSAs.

5.4.6 Considerations on tools and data provided by the distros

Disclaimer: this section contains considerations based on my experience with the tools and the data used to implement DiVulker. Debian and Red Hat are very big ecosystems: all the proposed suggestions and possible improvements will probably need to be reviewed and adapted considering other aspects I am unaware of. On the whole, I have a few criticisms that are to be considered constructive, as their aim is to grant an overall higher quality experience.

Debian

Starting with the tools offered by Debian Security Tracker, even if they could probably solve many parsing related problems, I decided to not use them for the following reasons:

- they are built with a weakly typed language (Python) and, for this reason, often it is not clear how the models are defined
- they are not provided as libraries downloadable through a package manager (i.e. pip), making difficult to track updates and possible breaking changes

- they lack modularity: the scripts are mostly 500-1500 lines long, hindering an external user to easily approach and understand the API

For what concerns Debian snapshot archive, even if the retrieval procedure is very slow due to the high quantity of HTTP requests needed to gather its data, I do not think anything has to be modified, since its goal is way wider than simply tracking security related information.
However, considering the exposed problems, I think Debian Security Tracker data (CVE/list, DSA/list and DLA/list) needs to be redesigned. Probably a first step could be the creation of a tool to identify all the inconsistencies in the files, so that they can be manually fixed. A second step could be the conversion of the data to a new easier to use format, maybe using JSON instead of a custom plain text format. Such format should try to avoid as much as possible information duplication (i.e. storing only in one file the data of the affected Debian releases) reducing the possibility to introduce inconsistencies among different files. Moreover I think that new IDs need to be introduced to track release dates of fixes that did not come with a security advisory.

Red Hat

Even if the provided Red Hat security data are quite easy to use, there is still margin for improvements:

- A JSON version (or other machine readable format) could be introduced, clarifying immediately how the data is structured.
- A change tracking system is not present: Debian uses Git to give users transparency of the applied changes.
- Few very old issues do not have a RHSA/RHBA/RHEA and, for this reason, they appear to be unfixed. However I do not think that even minor security problems gets ignored for such a long time.

- Since the data are only available via HTTP requests, “META” files like the one provided by NVD avoid useless parsing when the files have not been modified.
- Older entries use superseded formats: vulnerability scores using CVSS v3 should be added to the entries that only use CVSS v2, keeping the information backward compatible with other tools.
- CPE URIs are stored with different formats: the parsing effort is consequently increased, since it needs to match the values against multiple possibilities.

Chapter 6

Vulnerability data analysis

This chapter explains how a Window of Vulnerability of a CVE Entry is computed and shows some results obtained by analyzing the vulnerability data.

6.1 CVE Window of Vulnerability formula

To compute a Window of Vulnerability of a distro for a certain CVE Entry it is necessary to have:

- the date of disclosure (DD)
- the date of fix (FD)

The formula to compute a WoV is:

$$CVE\_WoV = FD - DD$$

If its result is a number less than or equal to 0 it means that the fix was released before or the same day as the vulnerability was disclosed: systems promptly updating their software should not worry about such vulnerability being exploited in the wild (obviously this is only true if the details of the vulnerability remain known only to those entrusted with its fixing).
Since DiVulker supports more than one vulnerability data source for both disclosures and fixes, for each CVE Entry having multiple dates information I considered only the **earliest** one (respectively called EDD and EFD):

$$CVE\_WoV = EFD - EDD$$

Moreover, it may occur that a CVE Entry affects more than one package, which fixes are released at different times. Therefore, the resulting Window of Vulnerability will be obtained by taking the average WoV of all the affected packages:

$$CVE\_WoV = \frac{1}{n}\sum_{p=1}^{n}(EFD_p - EDD)$$

where "n" represents the number of packages affected by the vulnerability and EFD_p is the earliest fix date for a certain package.

6.2 Vulnerability data analysis results

This section will show some data inferred from the Data Comparator microservice. The WoV data metrics are always:

- expressed in days, since the window usually ranges from few weeks to 3-4 months;
- computed only on fixed packages (so CVEs without any fix are excluded), since older unfixed issues largely impact on the final results.

Table 6.1 makes an overview on the whole dataset of the two analyzed distributions. As expected from the data sources issues showed in the previous chapter, Debian has a lot of CVEs without a fix: however this does not necessarily mean that they are actually unfixed, but more probably that no security advisories or Debian Snapshot data were found. As a matter of fact, later charts will show that many old CVEs remain unfixed for more than 10 years, and that is quite unlikely.

Distro	Pkgs Number	CVEs Number	Fixed CVEs Number	Average WoV
Debian	3735	27715	22891	115.7
Red Hat	4647	13457	13424	83.1

Table 6.1: Table showing the dataset available for the two distributions.

Another peculiarity of Debian distro is that it has almost a 4 months wide average WoV, despite its team declares putting effort in tracking and fixing security issues:

therefore, I divided the dataset by CVE ID year, discovering that the WoV of older CVEs have an high impact on the overall WoV (Table 6.2). This can be explained historically: the CVE standard was defined in 1999 and with it arose a greater interest and effort with security tracking and so, it is reasonable to think that distributions fully adapted to it with some delay, leading older data to be less precise. To have a better representation of Table 6.2 data, Figure 6.1 shows a graphical representation of its WoV column excluding:

- CVEs with the ID year portion before 2002, since they significantly impact on the chart scale
- CVEs with 2020 ID year portion, since their data is partial (other IDs can be assigned during the current year)

The analysis of this data reveals that, despite the number of CVEs affecting Debian packages is often more than twice the ones affecting Red Hat, their average WoV tends to be almost always lower. Thus, considering the entire dataset could be misleading.

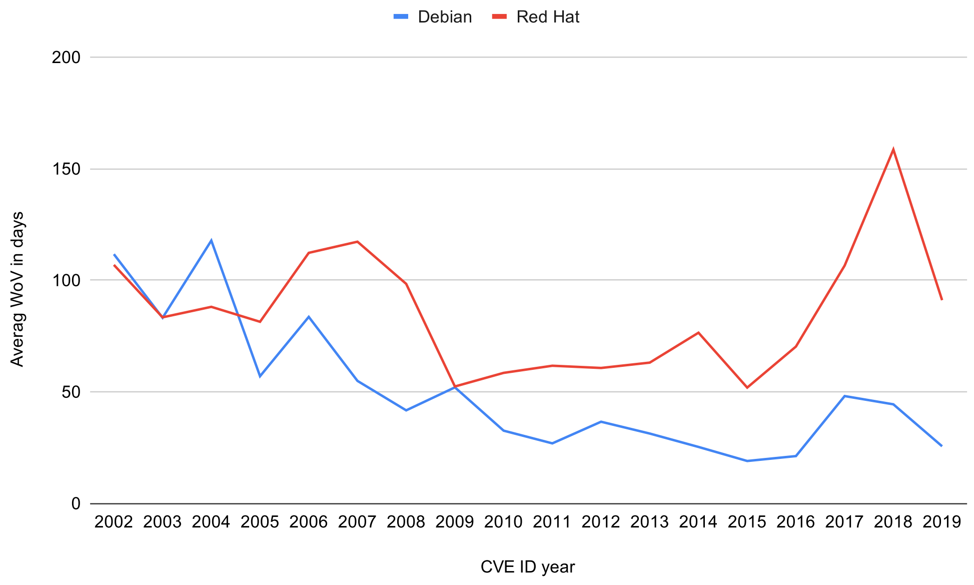


Figure 6.1: Average Window of Vulnerability divided by CVE ID year.

	Debian			**Red Hat**		
Year	Total CVEs	Fixed CVEs	Avg WoV	Count CVEs	Fixed CVEs	Avg WoV
1999	5	4	1913.3	7	2	2643.5
2000	9	3	2844.0	96	85	38.2
2001	24	9	829.7	155	150	63.0
2002	312	160	111.6	220	215	106.8
2003	399	254	83.1	196	193	83.3
2004	711	375	117.7	311	310	88.0
2005	1001	871	56.9	446	445	81.3
2006	1000	918	83.5	364	363	112.2
2007	1063	935	54.8	439	438	117.2
2008	1150	1011	41.6	467	467	98.3
2009	1045	968	51.9	529	529	52.3
2010	1252	1135	32.5	699	699	58.4
2011	1256	1127	26.8	640	640	61.6
2012	1431	1234	36.5	729	729	60.6
2013	1631	1456	31.2	953	953	63.0
2014	1758	1527	25.2	950	950	76.4
2015	1701	1510	18.9	1230	1230	51.8
2016	2306	2005	21.1	1342	1342	70.2
2017	3399	2840	48.0	1100	1100	106.5
2018	2835	2271	44.3	1221	1221	158.5
2019	2599	1954	25.5	1103	1103	91.0
2020	828	324	7.0	260	260	15.0

Table 6.2: CVE dataset divided by CVE ID year.

For what concerns the dataset splitting, the year portion of a CVE ID does not actually represent the date when vulnerabilities have been disclosed: for example CVE-2017-11137[1] still has the "RESERVED" state and so, its disclosure year date will surely differ from the CVE ID year portion.

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11137

Therefore, as shown in Figure 6.2 and Figure 6.3, I grouped CVEs by the year of their earliest disclosure. The charts display data considering a 10 year range of CVE IDs common to both distros:

- The first compares the total number of CVEs in common and the number of fixed CVEs of each distro. As expected, Debian has a slightly inferior number of fixes, due to the previously highlighted problems concerning the lack of information.
- The latter compares their average WoV. It can be observed that, except from 2009 to 2011, the trends have a similar curvature, however, it is clear how Debian presents a faster response.

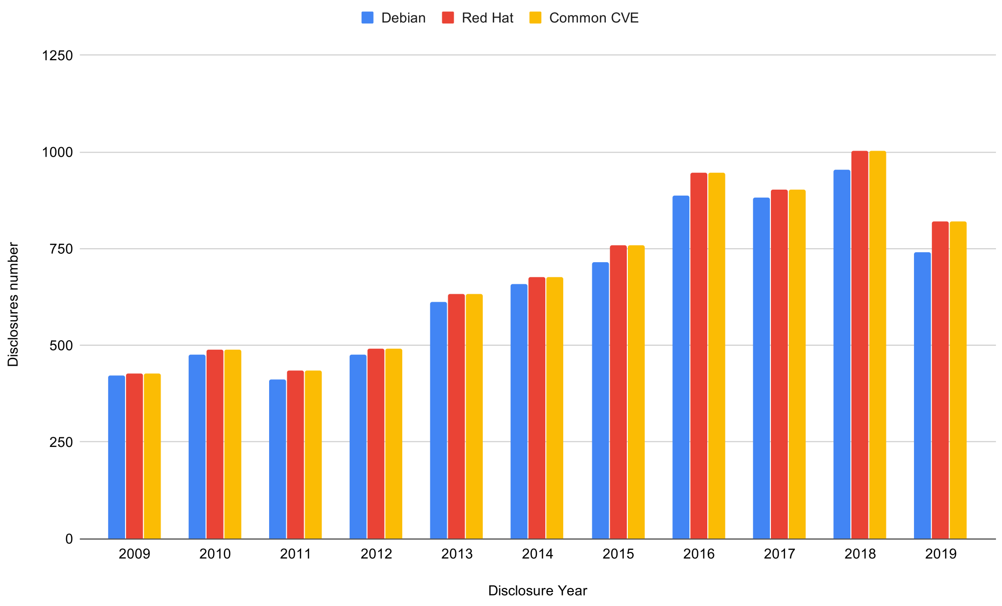


Figure 6.2: Fixed disclosures number of common CVE IDs.

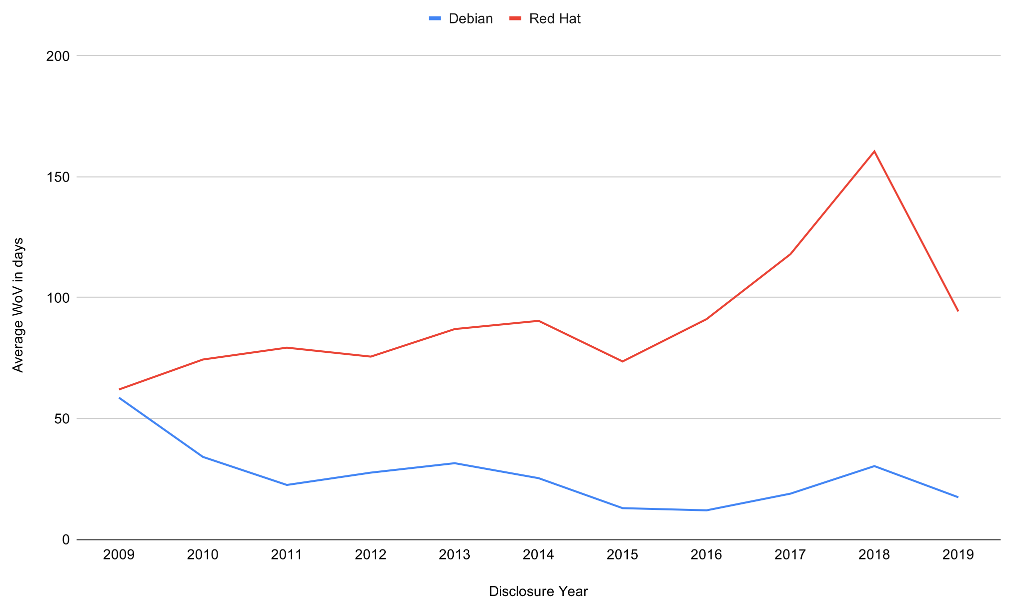


Figure 6.3: Average WoV of common CVE IDs.

Other interesting data can be observed by relating the average WoV with the CVEs severity:

- For Debian (Figure 6.4) the trend shows that:
  - from 2009 to 2015 there is no correlation between the closure of the WoV and the CVEs severity
  - from 2016 higher severity CVEs tend to have a lower WoV
  - the WoV tends to decrease over the years
- For Red Hat (Figure 6.5) the trend shows from the start an overall increased effort in closing the WoV of higher severity issues. Unexpectedly, the average WoV increases over time: this might be due to the fact that nowadays Red Hat maintains its products for at least 10 years, resulting in a slower vulnerability patching speed.

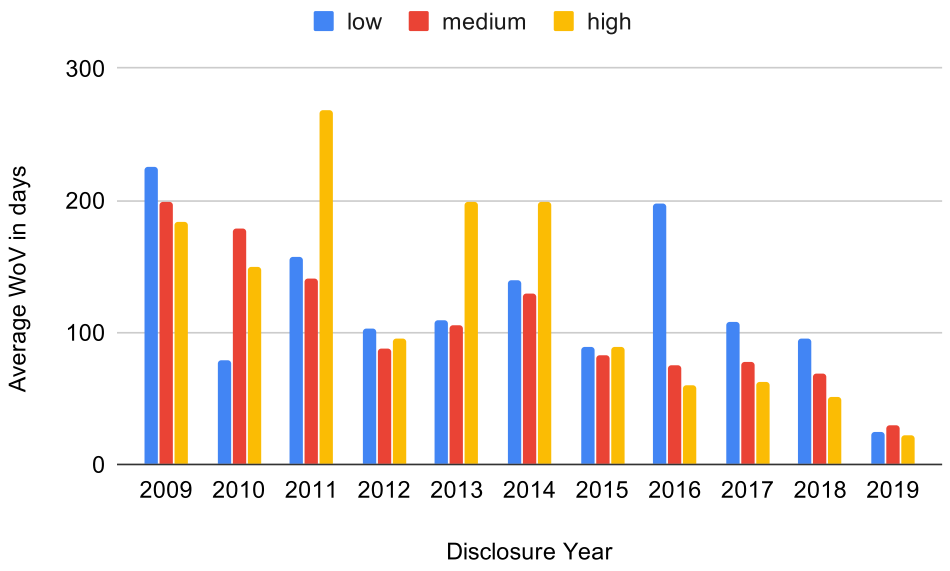


Figure 6.4: Debian's average WoV divided by vulnerability severity.

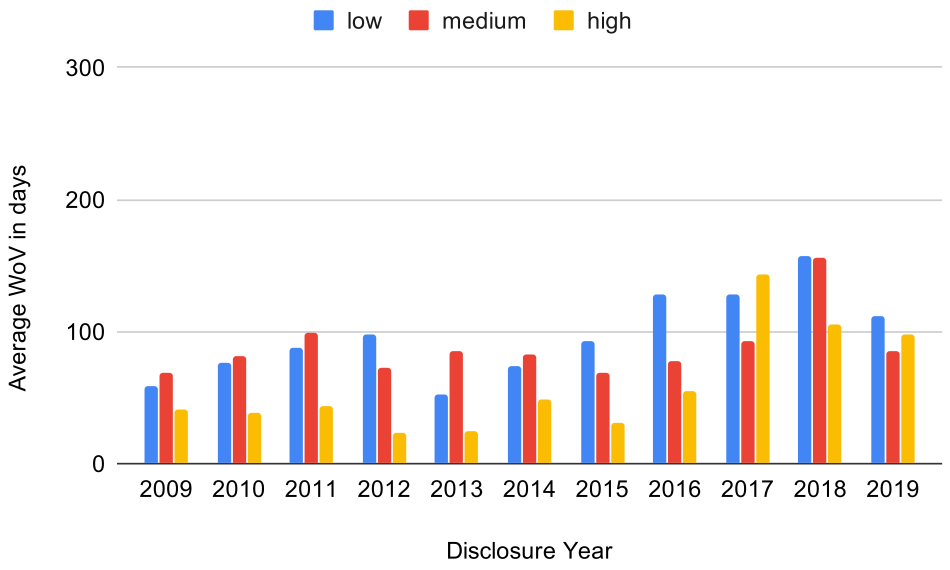


Figure 6.5: Red Hat's average WoV divided by vulnerability severity.

I also tried to find out whether there is some sort of correlation between disclosures release dates and the period of the year. To do so, I considered all the distros related CVEs: in Figure 6.6 it can be observed how on May, August and November there are usually less vulnerabilities disclosures (probably due to festivities).

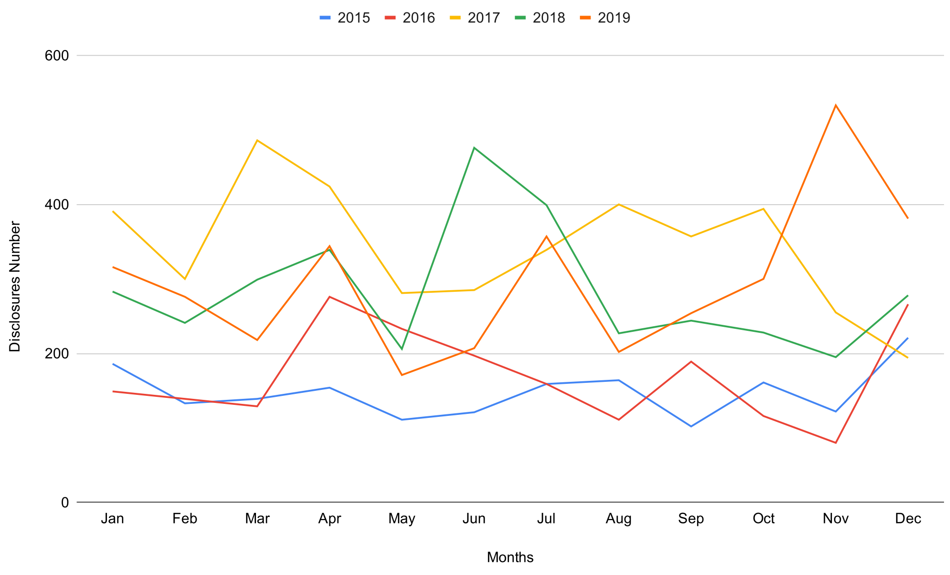


Figure 6.6: Number of disclosures per months from 2014 to 2019.

As an ulterior proof, in Figure 6.7 the same CVEs are grouped by the day of the week they have been disclosed, actually showing that the number of disclosures tend to decrease over the weekend.
Lastly the most used packages are another interesting aspect to take under statistical analysis: Debian weekly publishes on the Debian Popularity Contest page [S36] statistics on which are its user's most used packages. Unfortunately most of the top used packages are:

- Utility tools, like grep, gzip or sed, which have almost no vulnerability reports
- Debian specific tools, like dpkg or debconf, which are not present in Red Hat based products

making actual comparisons meaningless.

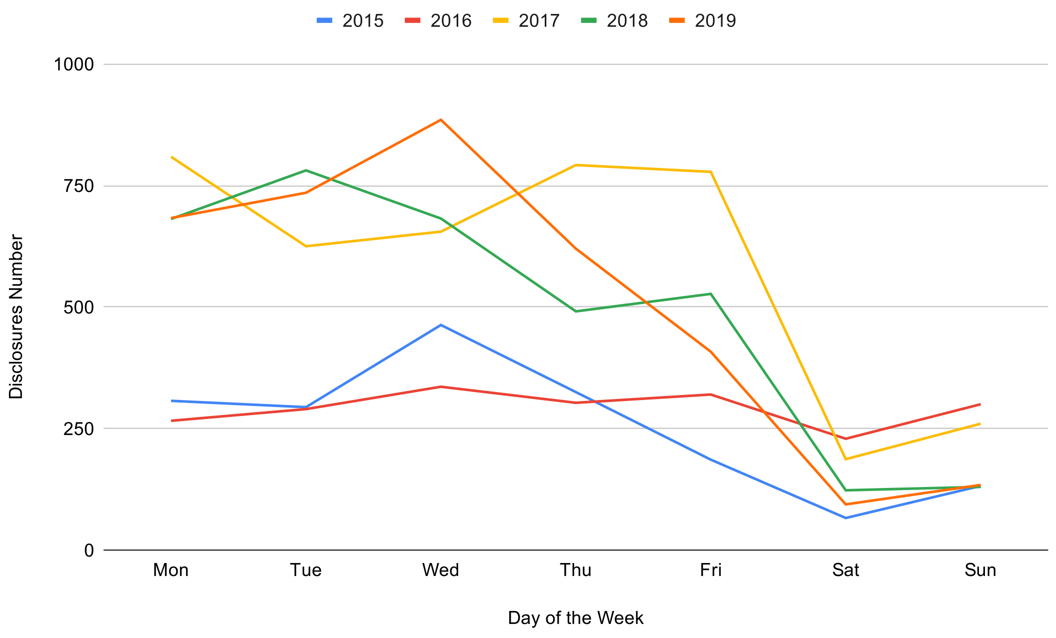


Figure 6.7: Number of disclosures per day of the week from 2014 to 2019.

For this reason, I took the top 10 vulnerable packages for each distro and computed the respective average WoV. As shown in Table 6.3, the packages with the most number of vulnerabilities are web browsers, mail clients and the Linux kernel, showing how the attacks mainly target highly used user applications or the system itself.

Sometimes performing this kind of analysis considering the overall WoV of a product reveals some complications: outside of package naming discrepancies between the distributions, the same product could be identified by different package names. Clear examples are the products referring to programming languages, like php5, openjdk-6 or java-1.7.0-openjdk, which major and sometimes minor versions appear in the package name.

Debian			**Red Hat**		
Package Name	Count	Avg WoV	Package Name	Count	Avg WoV
chromium-browser	1471	45.3	rhel_extras	3186	18.6
linux	1072	87.4	kernel	1230	109.1
iceweasel	607	231.5	firefox	1092	8.2
firefox	543	76.6	flash-plugin	925	3.4
icedove	536	239.4	thunderbird	862	9.0
imagemagick	492	31.6	rhel_productivity	592	13.2
wireshark	484	19.4	java-1.6.0-ibm	514	67.2
php5	482	169.8	xulrunner	471	7.7
openjdk-6	472	220.2	rhel_extras_oracle_java	408	44.3
xulrunner	378	204.8	java-1.7.0-openjdk	402	42.9

Table 6.3: Top 10 most vulnerable packages. Note that Debian identifies the Linux the kernel vulnerabilities as “linux”, while Red Hat labels them as “kernel”.

To sum up the data analysis, it is quite clear that the Debian distribution, especially in the last years, is the winner in terms of fix response. However, it is remarkable that the introduction of the missing fix dates information (approximately the 17.4% of Debian total CVEs) could largely change the obtained results. Moreover, when choosing a distribution, it should be also evaluated only the WoV of the packages that will be highly used by the organization, focusing the comparison between a restricted set of products. Finally the number of discovered CVEs is another factor to take into account: having a slower fix response and a lower number of discovered CVEs could be preferable for small businesses, which can not afford to frequently invest their resources in the maintenance required by the patching operations.

Chapter 7

Conclusions

The main objective of this thesis was the design and the implementation of a tool to measure the Window of Vulnerability of GNU/Linux distributions software. I will now review the topics I faced to come to its realization.
At first, it was necessary to identify the available vulnerability metrics standards, required to uniquely identify vulnerabilities across different systems, as well as their properties, such as affected products and issue severity. The challenging side of this step was retrieving all the required documentation (maintained by different institutions) to understand the purpose of each standard and how they come into play when considering the bigger picture.
Then, I defined what is a software Window of Vulnerability by proposing a new vulnerability life cycle model: this model, compared to the ones proposed in other related researches, defines with a greater level of detail the vulnerability correction phase, considering also the patching procedure on the final end systems. This is remarkable since it determines when the Window of Vulnerability caused by a vulnerable software gets actually closed.
After showing a brief overview of security patches application problems created by:

- system administrators, that often neglect them to avoid dealing with troublesome maintenance operations,
- software developers, that mainly put their effort in developing new functionalities for their products, and so overlook the review of security implication borrowed by the third-party library they use,

I showed how using a package management system to install, update and delete system software can improve the overall system security:

- at first by explaining their intrinsic advantages, such as automatic integrity and authenticity checks of downloaded software or easier dependencies management;
- then by presenting Debian's effective package management model, which is acknowledged and appreciated by the community for its stability and security.

For this reason and also due to the fact that package managers represent nowadays the standard way to manage software in GNU/Linux distributions, the developed tool is designed around computing and assessing the Window of Vulnerability of software distributed through them. The tool was designed using a microservice based architecture, allowing it to be modular and to scale by distributing its workload across different devices. Other benefits of this design choice are:

- future extensions are not bound to the implementation language used for its development;
- the microservices data can be used outside of the system, reducing the amount of work necessary in future security related researches.

Finally, the vulnerability data analysis results showed how, even though the amount of discovered vulnerabilities increases over the years, the effort put in their fixing has fortunately risen as well. In addition, these results, together with other considerations aimed to improve the quality of the data used to develop the project, will be sent to the Debian security team, that expressed interest in the discoveries of this research.

7.1 Future works

The designed system has been realized with the aim to support other vulnerability related researches. Surely adding more distributions to the analysis can produce even more interesting results. It should be noted that, even if the system was designed on GNU/Linux systems, it can be adapted with little or no modification to any system using a package manager to install, update and remove its software.

To this end, I suggest considering also:

- Unix-like distributions, for example as FreeBSD[1] and OpenBSD[2];
- rolling distributions, like Arch Linux[3] or Gentoo Linux[4];

which, in my first analysis, seemed to have all the requirements needed to perform a Window of Vulnerability related research.
Otherwise, it could be interesting to compare the Window of Vulnerability of software provided via distribution official repositories with the one provided via system agnostic repositories like Snapcraft and Flathub.
Finally, my research did not take into account the distribution code name affected by a vulnerability and which package version fixed the issue, as many issues emerged while parsing the available data. If anything changes in the data provided by the analyzed distributions, or other distributions have such information, they can be used to:

1. observe if the fix response decreases when the distro release switches to long term support or approaches to its end of life
2. determine the local system Window of Vulnerability, analyzing how long it takes to install security fixes in respect to their availability

[1]https://www.freebsd.org/security/security.html
[2]https://www.openbsd.org/security.html
[3]https://security.archlinux.org/issues/all
[4]https://security.gentoo.org/glsa/

Appendix A

Project Deployment

The project is publicly available for download and use on BitBucket[1] under the GNU General Public License v3.0.
The project is configured as a Gradle multi-project, where each microservice is treated like a module. The main modules of the project are:

- **common**: module used as an internal library to share common code among all the other project modules
- **debian**: module containing Debian distribution Data Retrievers and Data Aggregator microservices
- **distro-data-comparator**: module containing the Data Comparator microservice
- **general-retrievers**: module containing MITRE and NVD Data Retrievers and General Aggregator miscroservices
- **launcher**: utility module that provides a quick way to setup and run the entire system
- **red-hat**: module containing Red Hat distribution Data Retrievers and Data Aggregator microservices
- **registry**: module containing Registry microservice

In addition to these modules, inside the folder **web-gui** there is a very simple frontend developed in Angular that allows to show some statistics by contacting the distro-data-comparator microservice.

[1]Repository: https://bitbucket.org/FlamingTuri/divulker/src/master/

System requirements

The project requirements are:

- Java 8 or greater, to run the microservices
- npm and npx, to run the web-gui frontend

Microservices execution order

At the current system state, its microservices need to be executed in a specific order:

1. Registry Service, necessary to make the other microservices discoverable
2. Distro Specific Data Retrievers
3. Distro Data Aggregators
4. General Data Retrievers
5. General Data Aggregators
6. Data Comparator

Since in the actual system version the microservices keep all the data in memory, they can be shut down when all the necessary data has been retrieved by their "client" microservices, thus avoiding memory saturation problems.
To ease the system deployment burden, the project provides inside **launcher** module an application that will run the microservices in the correct order, by waiting the subscription of a microservice to the Registry Service before launching a new one. The application relies on MultiModuleDeployer[2], a library I developed to setup and execute projects composed by multiple applications. The library is still a work in progress:

- on GNU/Linux systems with Gnome Terminal set as default terminal application everything should work smoothly,

[2]Repository: https://github.com/FlamingTuri/multi-module-deployer

- on Mac Os systems, due to recent changes on permission settings, it might be required to allow AppleScript to send keystrokes (used to open different terminal applications)
- on Windows systems the library should work, but is not deeply tested

At its first usage the library will create the ".multi-module-deployer" folder inside the user's home directory. In this directory will be stored the script used to open and run commands in a new terminal, which can be easily modified to fix possible bugs or to change the default terminal application used to run the commands.

Project files and distributed deployment

The project files will be saved inside ".divulker" directory located in the user's home folder. To deploy the system over multiple devices the "registryLocation-Config.json" file needs to be modified, specifying the address at which the Registry Service can be found.

Bibliography

[B1] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. 3rd. USA: Prentice Hall Press, 2014. ISBN: 0133773922.

[B2] Barbara Guttman and Edward A. Roback. *SP 800-12. An Introduction to Computer Security: The NIST Handbook*. Tech. rep. Gaithersburg, MD, USA, 1995.

[B3] CVE Board. *Common Vulnerabilities and Exposures (CVE®) Numbering Authority (CNA) Rules V.3*, pp. 1–2, 6. URL: https://cve.mitre.org/cve/cna/CNA_Rules_v3.0.pdf.

[B4] Brant A. Cheikes, David Waltermire, and Karen Scarfone. *Common Platform Enumeration: Naming Specification Version 2.3*. NIST, Aug. 2011, pp. 8–13. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf.

[B5] Dan Goodin. "Eight months after discovery, unkillable LoJax rootkit campaign remains active". In: (Jan. 2019). URL: https://arstechnica.com/information-technology/2019/01/8-months-after-its-discovery-unkillable-lojax-rootkit-campaign-remains-active/.

[B6] Lily Hay Newman. "An Ingenious Data Hack Is More Dangerous Than Anyone Feared". In: (Nov. 2018). URL: https://www.wired.com/story/rowhammer-ecc-memory-data-hack/.

[B7] W. A. Arbaugh, W. L. Fithen, and J. McHugh. "Windows of vulnerability: a case study analysis". In: *Computer* 33.12 (Dec. 2000), pp. 52–59. DOI: 10.1109/2.889093. URL: https://ieeexplore.ieee.org/document/889093.

[B8] O. H. Alhazmi and Y. K. Malaiya. "Quantitative vulnerability assessment of systems software". In: *Annual Reliability and Maintainability Symposium, 2005. Proceedings.* Jan. 2005, pp. 615–620. DOI: 10.1109/RAMS.2005.1408432.

[B9] Stefan Frei et al. "Large-Scale Vulnerability Analysis". In: *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense.* LSAD '06. Pisa, Italy: Association for Computing Machinery, 2006, pp. 131–138. ISBN: 1595935711. DOI: 10.1145/1162666.1162671. URL: https://doi.org/10.1145/1162666.1162671.

[B10] P. K. Kapur, V. S. S. Yadavali, and A. K. Shrivastava. "A comparative study of vulnerability discovery modeling and software reliability growth modeling". In: *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE).* 2015, pp. 246–251.

[B11] Jeffrey Jones. "Estimating Software Vulnerabilities". In: *Security & Privacy, IEEE* 5 (Aug. 2007), pp. 28–32. DOI: 10.1109/MSP.2007.81.

[B12] Erik Derr et al. "Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2187–2200. ISBN: 9781450349468. DOI: 10.1145/3133956.3134059. URL: https://doi.org/10.1145/3133956.3134059.

[B13] Michael Backes, Sven Bugiel, and Erik Derr. "Reliable Third-Party Library Detection in Android and Its Security Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 356–367. ISBN: 9781450341394. DOI: 10.1145/2976749.2978333. URL: https://doi.org/10.1145/2976749.2978333.

[B14] D. Spinellis. "Package Management Systems". In: *IEEE Software* 29.2 (2012), pp. 84–86.

[B15] *Debian Developer's Reference - Resources for Debian Members*. URL: https://www.debian.org/doc/manuals/developers-reference/resources.html.

Sitography

[S1] Network Working Group. *Internet Security Glossary, Version 2.* May 2007. URL: https://tools.ietf.org/html/rfc4949.

[S2] *Script kiddie definition.* URL: https://www.computerhope.com/jargon/s/scriptki.htm.

[S3] Joint Task Force Transformation Initiative. *Guide for Conducting Risk Assessments.* Sept. 2012. URL: https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final.

[S4] *NVD vulnerabilities.* URL: https://nvd.nist.gov/vuln.

[S5] *MITRE - About CVE.* URL: https://cve.mitre.org/about/index.html.

[S6] *RFC 5646.* URL: https://tools.ietf.org/html/rfc5646.

[S7] *Common Vulnerability Scoring System SIG.* URL: https://www.first.org/cvss/.

[S8] *Vulnerability Metrics.* URL: https://nvd.nist.gov/vuln-metrics/cvss.

[S9] *About CWE.* URL: https://cwe.mitre.org/about/index.html.

[S10] *Meltdown and Spectre.* URL: https://meltdownattack.com/.

[S11] Moritz Lipp et al. *Meltdown.* 2018. arXiv: 1801.01207 [cs.CR]. URL: https://arxiv.org/abs/1801.01207.

[S12] Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution.* 2018. arXiv: 1801.01203 [cs.CR]. URL: https://arxiv.org/abs/1801.01203.

[S13] *NVD General Information.* URL: https://nvd.nist.gov/general.

[S14] *Lexico window of vulnerability definition.* URL: https://www.lexico.com/definition/window_of_vulnerability.

[S15] Bruce Schneier. *The Non-Security of Secrecy.* Oct. 2004. URL: https://www.schneier.com/essays/archives/2004/10/the_non-security_of.html.

[S16] Bruce Schneier. *The Process of Security.* Apr. 2000. URL: https://www.schneier.com/essays/archives/2000/04/the_process_of_secur.html.

[S17] *Third-party software component.* URL: https://www.semanticscholar.org/topic/Third-party-software-component/174394.

[S18] *apt man page.* URL: https://manpages.debian.org/stretch/apt/apt.8.en.html.

[S19] *RPM Package Manager homepage.* URL: https://rpm.org/.

[S20] *OneGet GitHub.* URL: https://github.com/oneget/oneget.

[S21] *Chocolatey homepage.* URL: https://chocolatey.org/.

[S22] *Npm home page.* URL: https://www.npmjs.com/.

[S23] *Debian oldstable FAQ.* URL: https://wiki.debian.org/DebianOldStable.

[S24] *Debian Backports homepage.* URL: https://backports.debian.org/.

[S25] Lucas Nussbaum. *Debian Packaging Tutorial.* URL: https://www.debian.org/doc/manuals/packaging-tutorial/packaging-tutorial.en.pdf.

[S26] *Debian Security Information.* URL: https://www.debian.org/security/.

[S27] *Alpine Linux: Mailing lists.* June 2017. URL: https://wiki.alpinelinux.org/w/index.php?title=Alpine_Linux:Mailing_lists&oldid=13584.

[S28] *[alpine-devel] Alpine security tracker discussion.*
URL: https://lists.alpinelinux.org/~alpine/devel/%3CCANtECw8gn93aiQ4qGoGcM3TJ4C6vLAK45cp62_nmHEL1usfg7A%40mail.gmail.com%3E.

[S29] *CVEProject/cvelist GitHub.*
URL: https://github.com/CVEProject/cvelist.

[S30] *MITRE Download CVE List page.*
URL: https://cve.mitre.org/data/downloads/index.html.

[S31] *NVD Data Feeds - JSON Feeds.*
URL: https://nvd.nist.gov/vuln/data-feeds#JSON_FEED.

[S32] *Debian Security Tracker.*
URL: https://security-team.debian.org/security_tracker.html.

[S33] *Debian Snapshot Archive.* URL: https://snapshot.debian.org/.

[S34] *Red Hat Security Data.*
URL: https://www.redhat.com/security/data/metrics/.

[S35] *Explaining Red Hat Errata (RHSA, RHBA, and RHEA).*
URL: https://access.redhat.com/articles/2130961.

[S36] *Debian Popularity Contest.*
URL: https://popcon.debian.org/stable/index.html.

Ringraziamenti

Ringrazio innanzitutto il professor Gabriele D'Angelo per avermi dato l'opportunità di affrontare un'interessante e non banale tematica di sicurezza informatica e per avermi guidato nella realizzazione di questa tesi con estrema disponibilità ed attenzione. Ringrazio inoltre Stefano Zacchiroli e Salvatore Bonaccorso per le delucidazioni fornite riguardo ai dati offerti dagli strumenti Debian, che mi hanno consentito di ottenere risultati sicuramente più significativi. Un grazie ai miei familiari che, nonostante i momenti difficili, specialmente dell'ultimo periodo, mi hanno sempre sostenuto e creduto in me. Infine un grazie anche ai miei amici, universitari e non, che mi hanno aiutato a vivere più serenamente questo percorso. Grazie veramente a tutti, potrà non sembrare, ma anche nelle piccole cose il vostro supporto ha contribuito a fare la differenza.