Ca' Foscari
University
of Venice

Master's Degree program

in Computer Science

LM-18 Class D.M. 270/2004

Final Thesis

Analysis of Cryptographic Vulnerabilities in Docker Images

Supervisor
Ch. Prof. Riccardo Focardi

Graduand
Filippo Camoli
Matriculation Number 871380

Academic Year
2021 / 2022

I would like to thank Professor Riccardo Focardi, supervisor of this thesis, for allowing me to undertake this thesis work whose work was facilitated by his spin-off company, Cryptosense, in particular the collaboration with the CEO and founder Graham Steel. Also, an important mention to the Cryptosense's Software Engineer Ian Barnes, for having provided support and availability for every need related to the company's products.

Last but not least, a special thanks to my cat, Bizet, for keeping me company from the beginning of my university career, despite his paws on the keyboard, writing apparently nonsense sentences amid the reports and source codes in these last 5 years.

Summary

1. Introduction

In the enterprise IT[1] world there is a strong push and interest in virtualization[2], especially for the many advantageous reasons that benefit both business and customer users. A fundamental pillar of the world of virtualization is the solution proposed by Docker[3], an open-source project that speeds up the deployment of applications in software containers that offer an additional level of abstraction by operating at the Application level. These containers, often light and small in size, are based on images that contain reduced operating systems including Linux and Windows distributions. In Docker, each container is derived from a docker image, which can be seen as an environment that can be created, modified, and deleted without affecting the main core of the base system image used.

This leads the research into looking for non-conformities in widely used products by the industry that could potentially put many users at risk. The current setup is focusing only on the cryptographic vulnerabilities and the purpose is to learn about them by answering some questions like “Why does this vulnerability exist? Is there any way to fix this permanently?”.

In the context discussed, thus the vulnerabilities in Docker images apply to any Linux environment and distribution, however, this thesis aims to give additional and direct support to the images curators given the great variety of distributions and applications present in the Docker community. All Docker images that have Windows, as their basic operating system, are not included in this thesis work, as they are incompatible with the software used in the experimental phase.
This thesis work provides a current[4] overview of some of the docker images, in particular, an extended look into vulnerabilities related to certificates, keys, and keystores[5]. Thanks to this overview, it is possible to understand the reasons behind the existence of these vulnerabilities and their fixes to avoid hypothetical attack scenarios.

---

[1] Information Technology,
[2] Computer Science can be referred to as the act of abstracting something, thus virtualizing Hardware and/or Software layers.
[3] Docker is not classified into Emulation and Para-Virtualization, but into Container-based Virtualization.
[4] Some of the results are gathered from images between March and May 2022.
[5] In particular Java Keystores, referred from now on with the acronym “JKS”.

The methods applied in the experimental phase of this thesis will be explained, from which it was possible to observe some collected statistics that provide thoughts for both end users and publishers/curators of Docker Images, to make them more aware of security risks and stimulate them to improve their solutions.

The results obtained show that there are some criticalities, at least minor than those reported by the analysis tools, which, however, it is always good to be able to take into account if the reader in the future wants to build or contribute to the development of applications distributed through Docker. The conclusions will have some open questions to which it has not been possible to obtain an official answer, this could emphasize that the lack of transparency can create not only ambiguities but also concerns.

Following the conclusion of the work carried out, the reader can find a section not strictly linked to the topic discussed, which is, therefore, a section of backgrounds relating to various information from the IT world, Docker, and Cryptosense, as well as the problems arising in the researching process of which I invite readers to read to understand the mechanisms behind these organizational realities better.

1.1. Cryptosense

Cryptosense is a software company located in France backed by some European-American secured capital firms whose core business is creating Cryptographic automated tools **[1]**, in particular the *Cryptosense Analyzer Platform (CAP)*[6] which I used to perform analysis on Docker Images. The CAP consists of a suite of different products like the Analyzer Platform, Application Tracer, Network Scanner, FileSystem Scanner, and HSM Scanner. The CAP is *easily accessible by everyone* and a free version is given to new subscribers. All the Cryptosense tools used in this thesis are part of the CAP suite.

In Section 6.1 the reader can find more information about the CAP product.

[6] The tools used in the experimental part are essentially the Analyzer Platform, via its APIs and the FileSystem Scanner via Host Scanner; more details are provided in Chapter 3.

2. Cryptography and Cyber Security

Before diving into the explanation of the experimental process or its analysis results, it is important to understand why and how Cryptography is essential in Security applied to Computer Science, starting with the basics fundamentals of Cryptographic concepts and scenarios used in this project.

2.1. The Basis of the IT Security

Cyber Security (CS)[7], aims to protect computer systems, networks, and their derivatives, starting from the Hardware to the Software layer. As with the rest of IT, Cyber Security covers a great variety of macro areas, however, all these areas have in common the fundamental key concept of IT security: the **"CIA"** *triad*. All the following explanations are elaborated from the original FIPS[8] publication n.199 [2].

"CIA" is nothing more than the anagram of *Confidentiality*, *Integrity,* and *Availability*. Terms that accompany every component of the computer world and that we will take into consideration throughout the thesis.

By *Confidentiality*, we mean restricting the authorization limits on sensitive data, in particular, in the context of data confidentiality, the aim is to not disclose such data to unauthorized entities.

Integrity, on the other hand, ensures that these aforementioned data are, over time, consistent, accurate, and reliable despite the changes that will have to follow a specific way with specific authorizations for being manipulated. In the context of *System Integrity*, the statement is similar but applied to a system that operates with its intended functions without exhibiting unauthorized manipulations.

The last one, *Availability*, ensures that the system will always provide information only for its authorized users.

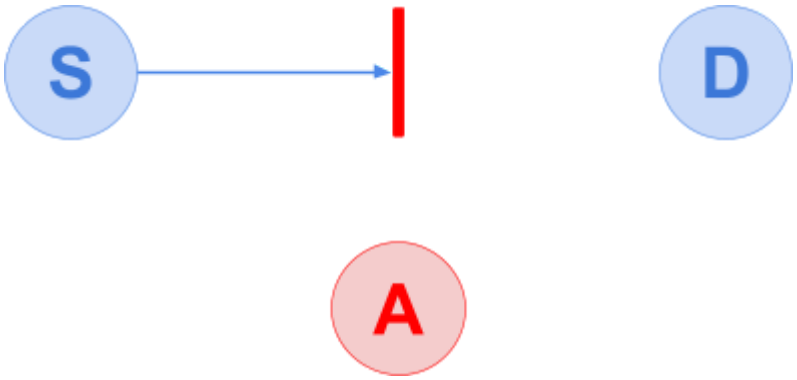
There are other terms like *Authenticity* (An entity must be identified correctly) and *Accountability* (Events are traced by unique entities) [3], but among the aforementioned terms, the most important ones for this research work are *Confidentiality*, *Integrity,* and *Authenticity*.

[7] Formally known as IT Security.
[8] Federal Information Processing Standards

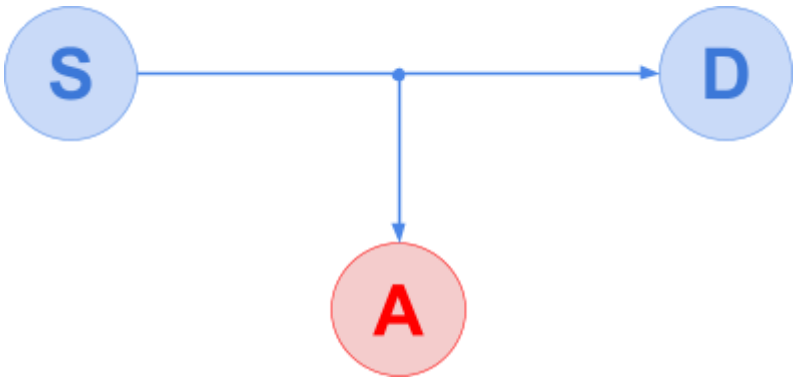
2.1.1. Some Attack Scenarios

Achieving these properties is almost trivial, here are listed the most common attack scenarios that illustrate the attack and what property will be broken [3][4]. As an assumption, we state that there exists an information flow from a Source (S) to a Destination (D).

a) *Interruption*: The attacker (A) actively[9] interrupts the flow of information.
→ It breaks *System Integrity* and *Availability.*

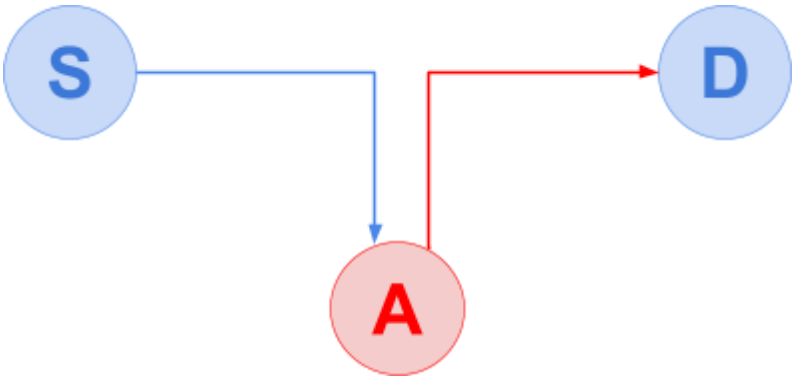


b) *Interception*: The Attacker (A) passively[10] intercepts information with unauthorized access.
→ It breaks *Confidentiality*.

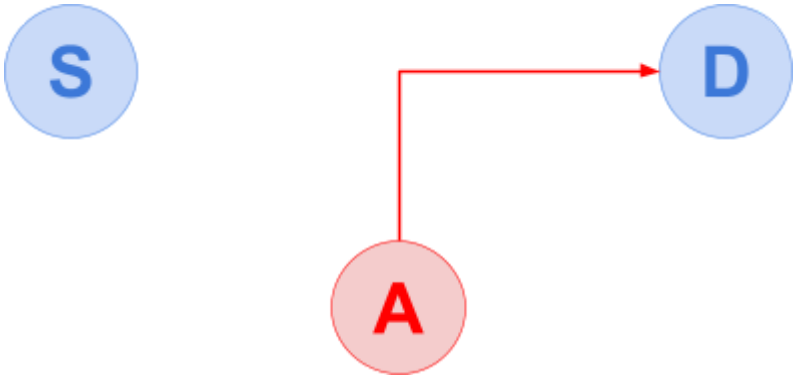


[9] Active attacks aim on altering existing resources.
[10] Passive attacks aim on learning information without altering the resources.

c) *Modification*: The Attacker (A) intercepts passively and actively manipulates both flow and information.
→ It breaks *Integrity*.



d) *Forging*: The Attacker (A) actively forges, without authorization, new information in the system.
→ It breaks *Confidentiality*, *Integrity,* and *Authenticity*.



e) *Corruption*: The Attacker modifies system functions.
→ It breaks *System Integrity*.

f) *Misappropriation*: The software of the Attacker abuses hardware and software resources without authorization.
→ It breaks *System Integrity*.

If any of these properties is broken in our system, then it is possible that the other ones will be broken, opening to more complex attack scenarios.

2.2. Why Cryptography?

By definition [4][5], Cryptography helps us to protect secret information in an insecure environment (i.e: the presence of a malicious entity), thus by applying some mechanism and protocols, the previously discussed properties can be safely archived but not maintained[11] if we do not perform some manutention over the time.

The following notions and mechanisms need to be explained to understand how these useful tools require some care and attention since their incorrect use or the administrator's negligence could lead to the loss of important security properties.

2.2.1. Cryptographic Keys

The main core of cryptography is algorithms and keys. When algorithms were not enough to protect messages exchanged between two parties (i.e.: the algorithm was leaked and became public knowledge), keys are the best alternative, not only for security reasons (i.e.: they are simple to maintain).

The security strength of a key [6] lies in its conformation, like its algorithm, length[12], generation[13], and process of exchanging keys[14]. Nowadays the standard is to use, RSA encryption, at least 2048 bits for current systems [11]. We will see that many non-conformed keys found in Docker Images have short key length issues.

[11] By maintaining, we mean following the international standards that keep evolving through time.
[12] Referred also as Size, it is measured in bits and it is decided by the algorithm adopted.
[13] Keys must be generated randomly in order to avoid guessing.
[14] Dubbed RSA exchange key method does not preserve the forward secrecy property, instead, it is preferred to use a Diffie-Hellman algorithm in a security context [7][8].

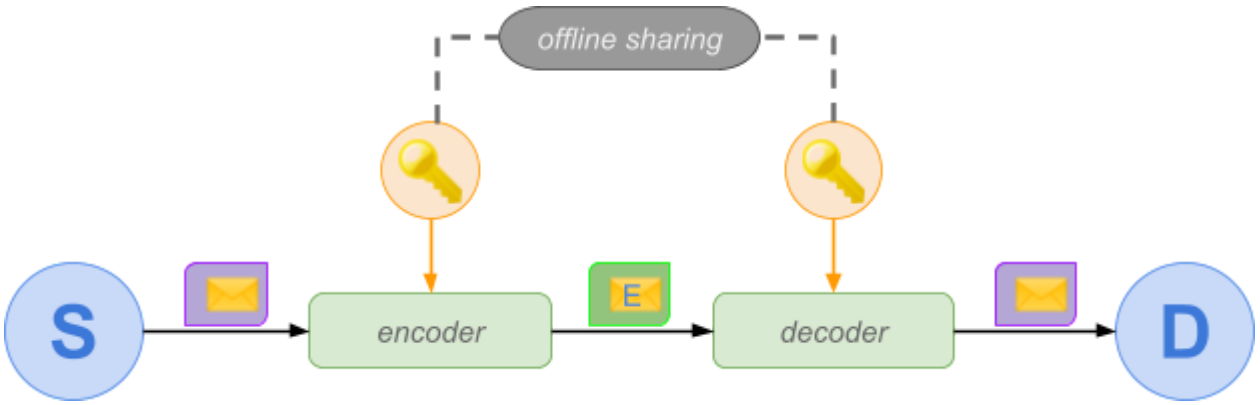
There are three main types of keys:

a) *Elliptic Curves* [13][14]

Elliptic Curves Cryptography (ECC) is a type of Public Key Cryptography where EC is both a geometric and algebraic object defined over Finite Fields. In general, EC keys benefit from a shorter key length, while providing the same security strength by assuming that a public point is known (ie: Public Key) and finding its discrete logarithm element is infeasible (ie: infeasible to find an associated Private Key given the Public one). ECC is mostly used for *Key Agreements* and *Digital Signatures*, so many existing algorithms for Symmetric and Asymmetric Keys were adapted to ECs.

b) *Symmetric Keys*

The encryption[15] and decryption[16] phases use the same key, making it efficient but it requires an offline sharing for establishing the key, moreover, encrypted messages are not uniquely making it difficult to distinguish the parties who sent the message.
Most famous algorithms used for Symmetric Key Cryptography according to [10]: *AES, DES, 3DES*[17]*, IDEA, Blowfish, RC4, RC5, RC6.*
Other algorithms may be deprecated due to known vulnerabilities on ciphers or due to insufficient support on key length complexity over its performance.

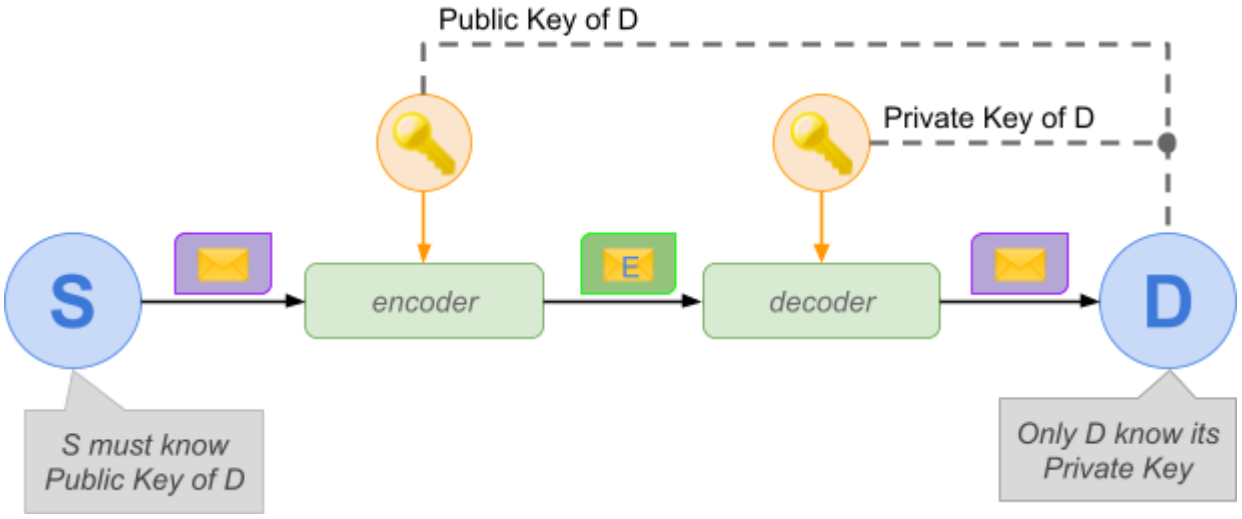


Symmetric Keys are now used only to ensure confidentiality and integrity of exchanged messages in a session[18].

[15] Encryption of a plain text into a ciphertext through an encryption algorithm.
[16] Decryption of a cipher text into a plain text through a decryption algorithm.
[17] 3DES is actually a variant of DES.
[18] In TLS connections, the Symmetric Key is used as a Session Key [9].

c) *Asymmetric Keys*

Each party has a pair of keys, a Public one, known to everyone, which is derived from a Private key, that is only known by its owner and infeasible to compute using its associated Public key. This kind of cryptography is now used everywhere for everything.
By using the Public key in encryption, the decryption can be only made by using the Private key.



If the Private Key is used for encryption, thus the Public Key for decryption, then we are dealing with *Digital Signatures*.

Thus, Asymmetric Cryptography is used for establishing[19] a new Symmetric Key, called Session Key, whose purpose is to exchange information ensuring *Confidentiality* and *Integrity,* but temporarily. *Authenticity* is not achieved yet since the key exchange does not guarantee that a Server owns a Public Key. This can be achieved by using *Certificates*.

[19] Establishing a session key through Dubbed RSA will not preserve the *forward secrecy* property, it is suggested using a Diffie-Hellman key exchange method to achieve this property.

NIST provided [12] a table of recommended key lengths (in bits):

In Asymmetric Algorithms with *Finite-Fields Cryptography*[20], **L** is the *public* key length, while **N** is the *private* key length. Rows in red are no anymore used for security reasons.

Security Strength[21]	**Symmetric Key Algorithms**	**FFC**[22] **(DSA, Diffie-Hellman)**	**IFC**[23] **(RSA)**	**Elliptic Curves**
≤ 80	2TDEA	L=1024 N=160	1024	160-223
112	3TDEA	L=2048 N=224	2048	224-255
128	AES-128	L=3072 N=256	3072	256-383
192	AES-192	L=7680 N=384	7680	384-511
256	AES-256	L=15360 N=512	15360	≥ 512

[20] Group-based cryptography done over the integers modulo a prime [13].
[21] It is the estimated maximum-security strength in bits provided by the algorithms and their key lengths in its row.
[22] Finite-Field Cryptography, see note #15 above.
[23] Integer-Factoring Cryptography, RSA is based on performing integer factorization [13].

2.2.2. Certificates and Digital Signatures

Authenticity is easily achieved by an *Identity Certificate*, also known as *Public Key Certificate*, which validates the identity of the owner through the data digitally signed on the certificate itself. *Digital Signature (DS)* is a scheme to verify the *authenticity* of a document. An implicit effect of DS is to detect tampered or forged data. It is based on *Asymmetric Cryptography* and, as mentioned in the previous section, a document can be signed by using the private key of an entity and the latter must provide its public key for signature check (thus the certificate).

A standard certificate **X.509**[24] (v3) includes several pieces of information [15]:

> *Subject* (The identity of the owner)
>
> *Public Key* (Of the owner)
>
> *Issuer* (Who released the certificate)
>
> *Validity* (Not Before and Not After DateTime)
>
> *Serial Number* (A unique ID for revocation tracking when a subject is compromised)

The most important ones for achieving the *Authenticity* property are the:

- **Signature Algorithm** (It contains the hashing[25] algorithm used in signature and the encryption algorithm).
- **Signature** (the body of the certificate is hashed[26] and then encrypted with the private key of the owner, according to the algorithms specified in the Signature Algorithm field).

Thus we can check a digital signature of a certificate with its public key, and trust[27] the certificate if, and only if, the hashed signature matches with the hashed body of the certificate itself.

---

[24] Standard defined by *International Telecommunication Union (ITU).*

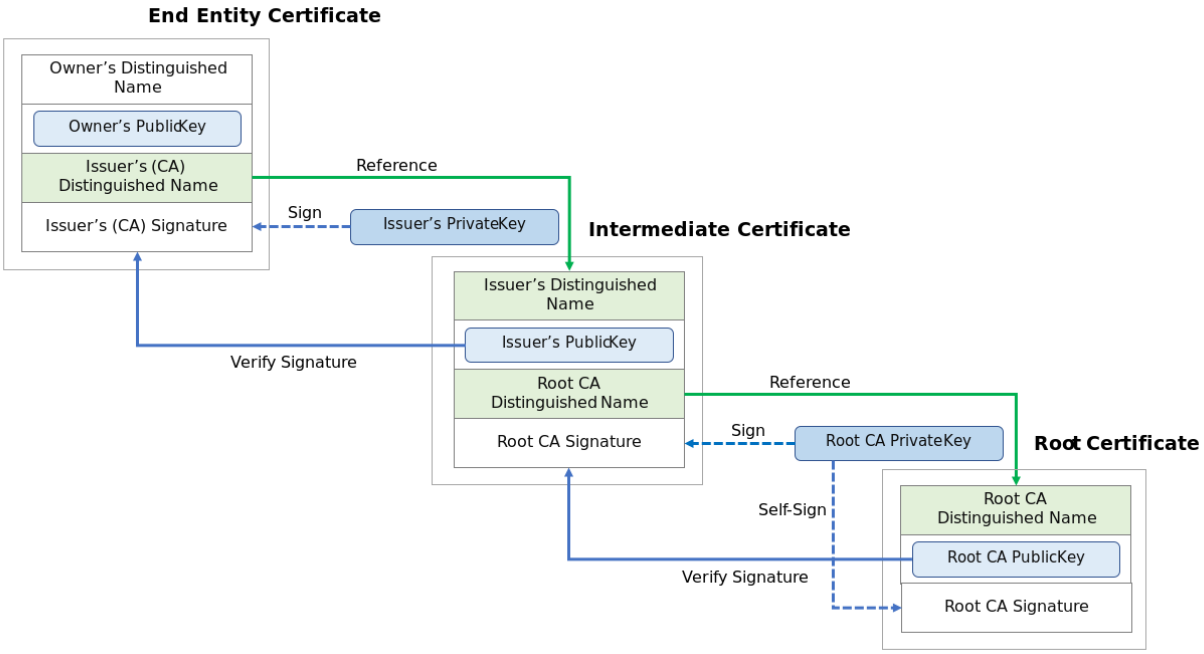
[25] A Cryptographic Hash Function is a deterministic algorithm that always maps original information into its same hash value, a.k.a. message digest. Hash Functions are strongly used because of their properties such as the *one-way* property, which makes hash functions *infeasible* to reverse their computation. The only way to break a hash function is to use a brute-force approach of possible inputs or using a *rainbow table*.

[26] It is more time-space efficient to sign the *hashed value* of a document than sign the entire document itself, due to its shorter length.

[27] Under assumption that the private key is not leaked/compromised.

In this research work, certificates are mostly *Self-Signed* certificates and *Certificate Authorities (CAs)*. A certificate can be released by the owner itself within its signature. A self-signed certificate can be trusted only if the *certificate chain* can be checked successfully, which is a list of certificates such that [8]:

> “
>
> 1. *The Issuer of each certificate (except the last one) matches the Subject of the next certificate in the list;*
> 2. *Each certificate (except the last one) can be verified using the public key contained in the next certificate in the list;*
> 3. *The last certificate is a trust anchor: a self-signed certificate that one trusts because it was issued by a trusted* ***certification authority****.*
>
> ”



Credits: Yuhkih, CC BY-SA 4.0, https://creativecommons.org/licenses/by-sa/4.0, via Wikimedia Commons

A Certificate Authority [16] is an authoritative entity that stores, signs, and issues digital certificates used in HTTPS. Typically, CAs sign 3rd parties certificates for web servers and emails. Self-signed CAs are called “Root CA”, and CAs signed by other CAs are called “intermediate CA”. Root CAs are provided with the OS and/or with internet browsers. Further information can be found in Section 6.3.

2.2.3. Keyrings

The last, useful, tool to introduce are *keyrings*.

Keyrings are digital archives to store cryptographic keys but also passwords. In our study case, we are dealing with *GPG*[28] *keyrings* used by *Debian* distros and a sub-type of keyrings, the *Java Keystore*.

2.2.3.1. Debian Authentication Keyrings

Debian operative system allows users to install Debian packages (*.dpkg*) through *APT*[29], a useful tool whose purpose is to automatically solve dependencies, configure modules of the packages, and manage the installed packages. APT is capable of providing *safe* and *trusted* packages through a secure mechanism: the "*Debian Project*" [18].

The goal of the *Debian Project* is to ensure the Debian distribution system has strong security against forgeries. It consists of distributing, collecting, and managing the digital signatures of developers within the *OpenPGP* standard keyring, containing keys of *Debian Developers*. This keyring (*Debian-keyring.gpg*) relies on *GnuPG* (*GPG*) implementation, whose realization is a concatenation of multiple *.gpg* files containing keys.

Some keyrings are stored in the following path: */usr/share/keyrings/* , and the main keyrings are provided with the Debian release installation. Keyrings can be updated through the official repository, the *Debian Public Key Server* (*keyring.debian.org*) a PKI[30] that deals only with keys for Debian Project members. These members can enroll in the *Debian Project* program and through a complex iter, which involves an identification check[31], the new members can become part of it and get the associated personal pair of keys[32]. Each new pair of keys are signed by two other members. New developers can sign their packages and put them online through Debian Archive[33].

---

[28] GNU Privacy Guard [17] is a free multi-platform implementation of the RFC4880 standard (OpenPGP) and its purpose is to offer a cryptographic suite (including most used Public Keys, Ciphers, Hashes, Compressions).
[29] Advanced Packaging Tool.
[30] Public Key Infrastructure, a theoretical infrastructure as a superset of policies, roles, and entities that create, manage, revoke, use, store certificates and public key encryption.
[31] GPG refers to this mechanism as "web of trust" in its official manual [17].
[32] A Private key and its associated Public key.
[33] Debian Archives is an official repository maintained by Debian official members, in these archives there are software and security updates, major releases and packages.

Each Debian release comes with a set of keyrings, the main ones located in */usr/share/keyrings/* are the following ones with an explanation of their purposes according to [19][20][21]:

> *debian-keyring.gpg*:
> Contains all OpenPGP keys of Debian Developers with upload privileges to *Debian Archives*.
>
> *debian-archive-keyring.gpg*:
> This includes all actively used *APT* keys to sign *Release* files in supported Debian releases.
>
> *debian-archive-removed-keys.gpg*:
> This includes all used keys in previous releases, that were used to sign Release files, and now are no longer supported.

Every time the user uses *APT*, with a Debian mirror, a **Release**[34] file is checked for **Packages.gz** integrity (MD5[35] sums) and it contains all the Debian packages available in that mirror [22]. *APT* uses all the keys used in these important keyrings in */etc/apt/trusted.gpg.d/,* for checking the signatures of the acquired *Release* file against the key database.

In the trusted GPG directory, several keyrings usually consist of three types of keyrings, each of them for every latest three Debian releases:

> *debian-archive-bullseye*[36]*-automatic.gpg*
> *debian-archive-bullseye-security-automatic.gpg*
> *debian-archive-bullseye-stable.gpg*
>
> *debian-archive-buster*[37]*-automatic.gpg*
> *debian-archive-buster-security-automatic.gpg*
> *debian-archive-buster-stable.gpg*
>
> *debian-archive-stretch*[38]*-automatic.gpg*
> *debian-archive-stretch-security-automatic.gpg*
> *debian-archive-stretch-stable.gpg*

[34] Starting from Debian 10, **Release** is deprecated and instead **inRelease** is used. The former uses a detached **Release.gpg** signature and the latter is cryptographically signed **in-line**.
[35] A MD5 checksum is a method of taking a file and condense it to a unique short number that identifies the content of the file.
[36] Bullseye is the actual and latest release - Debian 11 (2021).
[37] Debian 10 (2019).
[38] Debian 9 (2017).

Automatic public key is used for using the release of the next Debian release and upcoming expiring keys in 2027. *The Security-Automatic* key is used for security archives (i.e. security updates) and its key is signed by the FTP-Master key only. Lastly, the *Stable* public key is used for the current stable Debian release.

These keys usually are strictly verified and under control[39], which is why we will never see them through the report's analysis in Chapter 4.

2.2.3.2. Java KeyStore (JKS)

The JKS is a repository for storing public and private keys within signed certificates used in SSL[40]/TLS[41] encrypted connections. *TrustStore*, instead, is similar to the KeyStore concept, but it is meant to contain only the trusted certificates of the entities (i.e. CA certificates) that the user will use when dealing with third-party certificates.
Default Java KeyStore and TrustStore are installed as part of the *Java Development Kit (JDK)*. The main one is the *cacerts* file, which stores CA certificates, hence self-signed CA certificates embedding the CA public key. Java keytool program is provided in the JDK but it cannot extract private keys, there is a need to use third-party programs to do it, for instance, KeyStore Explorer or other programs that are discussed in Section 3.4.5.

There are other alternatives for the JKS (or Java TrustStore), like the PKCS format which is taking the lead with its latest versions of PKCS#11[42] and PKCS#12[43]. In macOS, the equivalently default Keystore used is the *KeychainStore* which also embeds KeyChains.

As stated in **[23]**, the pre-2017 versions of JKS are *weak and insecure*[44]. New JKS versions use, according to Oracle, an increased iteration count of the weak algorithm used, the SHA-1. This "fix" is not gonna mitigate the problem since it will just extend the brute-force attack time required for cracking such a format. It is recommended to use the PKCS#12 format since, at least for now, it does not present any security issues.

---

[39] Each key in *trusted.gpg.d* is signed by two Debian FTP-Masters, members of the FTP team **[24]**.
[40] Secure Socket Layer: It is a cryptographic protocol designed for providing security over communications in computer networks.
[41] Transport Layer Security: the successor of the SSL.
[42] PKCS#11 is an interface/hardware format for HSM or any cryptographic tokens hardware-based through its API, it provides the usual operations for dealing with certificates and keys.
[43] PKCS#12 is a software format for storing certificates and private keys, mainly used when converting or migrating such information to different platforms.
[44] CVE-2017-10356.

3. The Docker Images Analyzer Program

In this section will be illustrated the realization of the analyzer "script[45]" made by the author. Its purpose is to get all the *Officials* and *Verified-Publisher* Docker Images, execute the Cryptosense Host-Scanner executable, upload the generated trace and launch its analysis to get a report. Through the report, some checks and statistics are performed. Most of the final data obtained are analyzed manually.

There will be no code fragments in the following explanation, if the reader wants to see the source code, it can be seen through my GitHub repositories: here for the C++ program and here for the Python Web Scraping script.

3.1. Dependencies

In this project there were used some external libraries:

For the Python Script:

- Selenium 4, for scraping static and dynamic web pages.
- Web Driver Manager *[deprecated[46]]*, used for automatically managing the web drivers.

For the C++ Program:

- *Curl*, for making requests/responses to/from CAP APIs.
- *Boost*, for Thread Pool and other minor stuff.
- *Botan*, an X.509 parser.
- *Nlohmann-json*, is a beautiful json parser extremely versatile.
- *Python* [deprecated], official Python Library for C/C++, originally used for embedding the *WebScaping* script but unfortunately something is broken under macOS[47].

[45] Mixture of Script and Program actually.
[46] I used the Safari web driver embedded in Selenium 4, but deprecated code is still commented with Microsoft Edge web driver implementation.
[47] The entire project was developed under macOS and Xcode.

3.2. System Requirements

For completion:

- Processor with 4+ Thread/Core support
- At least 700MB of RAM (when dealing with ~500 Docker Images)
- At least 1GB of free disk space (and r/w permissions)
- Internet access
- Python (3.9+ suggested) installed
    - and its dependencies, see above
- C++ Compiler (Clang was used during development)
    - and its dependencies, see above
- Docker Installed and booted up
    - At least 200GB+ free space if you want to analyze ~500 Docker Images
    - Docker Account signed-in.[48]
- A Cryptosense Account and API Token
- A lot of time

3.3. File Structure

DIR	Files	Purpose
/Data	*/list_docker_official_images_RAW.json* */list_docker_store_images_RAW.json* */list_docker_official_images_API.json*	List containing all docker images from docker hub (through WebScrape script) in the form *<namespace> / <name>*. 'Official' are those images marked as official images, all of them have *Library* as namespace; 'Store' are those images released by Verified Publishers. Only official images can be retrieved through docker APIs directly in the C++ program in the *<namespace> / <name> : <tag>* format.
	/list_docker_store_images_FIXED.json	The above list is giving *no tags*, the Program will resolve the latest tags to keep track of the versioning and to resolve platform issues[49]. Only the official list (with API) comes with tags.

[48] See Chapter 6 - Docker for its limitations.
[49] See Chapter 6 - Docker.

/Data	*/list_docker_images_FAILED.json* */list_docker_images_NO_TRACES.json*	List of Docker Images in json format from Data Structure described in Section 3.5.1.1. The *FAILED* list contains all the docker images that have failed after the trace creation (Error occurred when using CAP) or they have created a *non-valid Trace* (size is 0 Bytes). The *NO_TRACES* list contains all docker images that have *no trace created* from the first phase (Error occurred in Docker Client Engine). Both lists can be used in the next runs to try to resolve the issues or just for logs.
	/Stats_871380@stud.unive.it i-th.json */JKS_871380@stud.unive.it i-th.json* */Reports_871380@stud.unive.it i-th.json*	*Reports,* is a local cache of the reports generated on the CAP. *Stats,* will contain some simple stats of the analyzed reports. *JKS* is the list of reports for JKS files where it describes the type of keys inside the Keystore.
/JKS	*Contains folders with names of the format: <namespace>_<name>-<tag>*	The C++ program provides a JKS check and creates a final JKS report (see above). To do this check, it was necessary to use a temporary directory to extract all the JKS files from the Image and then checked against Floyd's .jar tool for extracting private keys in /JKS. This .jar script creates a *hash.txt* output file, if it's empty, then the JKS contains only Public Keys, if it contains at least one hash, then it contains a Private Key.
/Script	*/start_container.sh* */terminate_container.sh*	Bash Script for Docker CLI management. *Start* and *Terminate* a container given the full image name [namespace/name:tag] and the sanitized name (no special symbols are supported in Docker CLI like '/' and ':').
	/process_image.sh	As above, but: it starts, copies in the container & runs the host-scanner executable, copies the generated trace

		into the host, and then terminates the container.
	/copy_extract_jks.sh	A script that prepares the directories necessary to run the .jar file. Copy from the container to the host the JKS file reported by CAP reports and then execute the .jar check for extracting private keys.
/Traces	*"/Tests" directory (just for personal purposes)*	It contains all compressed traces (with extension *.cst.gz*) obtained from the *process_image.sh* script, to be uploaded to the CAP.

3.4. CAP Integration

CAP allows organizations to create and manage users in different roles such as *Analysts*. Each Analyst-User is allowed to manage *Profiles*, that is a set of rules to determine the cryptographic critical issues, and also to create some *Projects* that act like folders, and in these folders, we can upload *Traces* (up to 100 per *Project*) and generate *Reports* from them.
To generate a trace, we need to scan the OS with some tools like the *Cryptosense Host Scanner*[50]. In this project, due to compatibility issues with Docker Images dependencies, JAR[51] files are not considered. Once the trace is uploaded to CAP, we can generate its *Report* by using a *Profile*. A *Report* is made of *Instances* and *Certificates*.

An *Instance* is the result of a found key whose key information determines the criticality according to the selected *Profile*. Each Instance has a type[52], which is defined by the author for extracting statistics. There is also provided an explanation on why such a level of criticality is assigned that includes the path of this key.

Certificates are just an abstraction of the real certificates into the scanned/analyzed host and they can be considered a specialization of *Instances*.

Note: Multiple *Instances* referred to the same file can occur since this file could be an archive of keys, and an *Instance* is created for each key.

[50] It allows scans of different formats and modalities.
[51] Also known as Static scan.
[52] Types of Instances classified by me: *CERT_SHA1, STALE_PK_RSA, STALE_PK_DSA, CERT_EXPIRED, SHORT_RSA, SHORT_DSA, CERT_3650, JKS_VALIDITY, JKS_SHA1, SOMETHING_ELSE.*

Cryptosense Analyzer Platform can be used and almost fully managed[53] through restful APIs via API Key. It is possible to

- Get User and Organization information
- Create a Project
- Upload a Trace
- Analyze the Trace and Generate a Report
- Delete a Report and a Trace
- Get Reports (Instances and Certificates)
- And more…

The requests are made in GraphQL format and the query construction can be made through the provided API Documentation or using the *Cryptosense Query Explorer* accessible in the personal area.

All the responses are in json format, except for one which is in XML format[54].

[53] In comparison to the Web UI, see *Section 6.1* for more backgrounds.
[54] For details see Section 3.5.4.2.

3.5. Implementation

Despite the efforts made to parallelize everything at its finest, Docker CLI operates the commands sequentially, meaning that some parallelism is lost in multiple commands present in *process_image.sh* bash script. For example, if we have three threads that have to execute this script, the script is running in parallel, but for each Docker command, Docker CLI puts in a Stack the commands and executes them, but only the execution of Docker commands is sequential, meaning that multiple containers can analyze in parallel after a sequential analysis launch, which is the part with the most gain.

Bash execution respects the ENV33-C rule [25] for executing trusted commands.

3.5.1. Data Structure

To keep track of the data managed and manipulated, and also to facilitate the debugging process, it was necessary to create several data structures. Buffers used in the program are lists.

3.5.1.1. Structs

Many structures were made in this project. Each of them provides a To/From jSon serialization to automatically provide json compatibility with these structures. The following table there are illustrated the purpose of each structure with additional information for some of them:

Struct Name	Purpose
Profile	Struct for the CAP *Profiles,* which includes the name, types, and ids.
Project	Struct for the user Projects with name and *ids*. It also counts the number of traces in a project, this is helpful to create a new project when the other ones are full.

DockerImage	Struct for the Docker Images information such as *namespace*, *image name,* and *tag*, which is resolved at object instantiation via Docker Hub API v2.
Image	The most important struct for tracking the Images dealt with the analysis throughout the entire program. It is used to save image information such as its *sanitized* name, *trace* file information, *trace id,* and *report id*.
Report	Struct that has a reference to the original *Image* object and saves the information as to its *id* and *list* of *Instances* and *Certificates*.
Instance	Struct is used to keep information such as *severity*, *criticality,* and the *type* of *Instance*.
Certificate	All the *certificates* are abstracted in this struct that keeps information such as the CAP *id* and its *PEM* encoding.

3.5.1.2. Lists

Plenty of the following lists were used as buffers among threads, thus some mutex is introduced[55]. To optimize memory and execution speed, all the lists are lists of references. For the entire execution of the program, almost all lists are used as FIFO structures.

The most important ones are:

- std::list<DockerImage::DockerImage*> listDockerImages
  - *List of Docker Images containing the information of Docker Hub.*
- std::list<Profile*> profiles *and* std::list<Project*> projects
  - *List for saving respectively CAP Profiles and Projects in CAP. Project list is the only list that needs to maintain its order.*
- std::list<Report::Report*> reports
  - *List of Reports downloaded from the CAP.*
- std::list<Image::Image*> images_failed_no_traces
  - *List of Images where they failed in providing their trace in the Scan phase which involves Docker Commands and Host Scanner.*
- std::list<Image::Image*> images_failed
  - *List of Images that in any point of the program, after the Scan phase, an error occurred (mostly about CAP).*
- std::list<Image::Image*> images_to_pull
  - *List of all Docker Images that we need to pull from Docker Hub and be scanned.*
- std::list<Image::Image*> images_scanned
  - *An Image after being Pulled, so after HostScanner created a trace file, the Image is now in the list of the Images Scanned. Thus the images are ready to be uploaded by the upload thread.*
- std::list<Image::Image*> images_uploaded
  - *List of Images already uploaded into CAP servers[56]. Thus, images are ready to be imported.*

[55] Details on next section 3.5.4.
[56] CAP lies on AWS servers, in particular S3 servers.

- std::list<Image::Image*> images_imported
    - *List of Images already imported into the project, now ready to launch its analysis/report generation.*
- std::list<Image::Image*> images_analyzed
    - *List of Images already analyzed/launched to be reported. Report is not immediately available thus it can also fail. Check Report thread works on this list for tracking the report's availability and success.*
- std::list<Image::Image*> images_done
    - *List of Images that have completed all the analysis processes and their report was correctly generated and ready to be downloaded.*

3.5.2. Web Scraping Docker Hub in Python

To analyze the docker images, a list of images must be provided. The first method that I found is to use the *Docker Hub API v2* which provides a list of all official images since they are published under user "*Library*". Other images list, unofficial and *Verified-Publisher*, can be only obtained from the Docker Hub webpage. This webpage is *dynamically* generated on the server side, so a basic web scraping will return only the default piece of HTML code (header and footer) but not the content that we need.

The final solution consisted of using Selenium (at the time of implementation and drafting of this thesis, the version is the v4) a Python package that can assist dynamic web scraping using the *Web Drivers*.

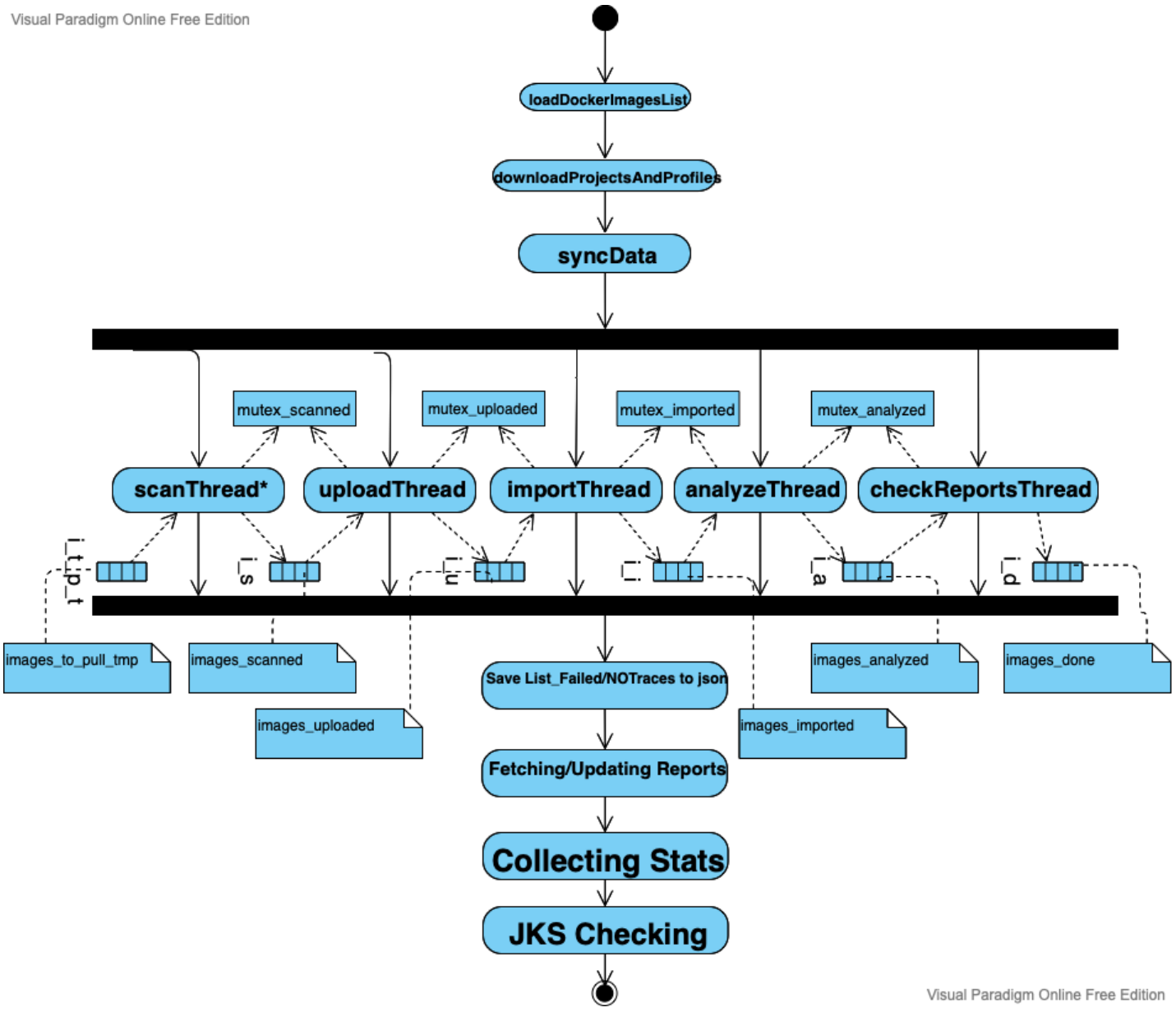
In short, the web driver gets the content at the generated link, that in this case is the Docker Hub base link with some URL filtering like the *type* of image (*official* for official images and *store* for published verified images), the *os* (host scanner is a UNIX executable, so Linux is our only choice) and the *CPU architecture* (In my case I am working on Apple Silicon/ARM64v8 platform). As I have been able to ascertain, the platform filter is almost irrelevant, because after discovering that the "*latest*" tag for each kind of platform is used improperly, the only way to fix this compatibility issue is to manually search for the correct associated tag for the right platform through *Docker Hub API v2*, which can be now used since we know the *namespace* and the *name* of the image that we need to pull and analyze.

Once the generated HTML code from the server is downloaded, we can pick up all the *hrefs* by ID in the DOM called "*searchResults*", which contains the *namespace* and *name* of the image.

The script automatically saves both *Official* and *Verified-Publisher* images and saves the lists into a json array file which will be used in the C++ program, where it will also perform the tag fixing. The total time elapsed for scraping all the web pages is around 15 minutes, due to the necessity of introducing 2 seconds of sleep from each request because the Docker Hub server is quite slow at generating the web pages.

3.5.3. Overview

The following UML activity diagram describes, in a very simplified and notation-abused way, how the C++ program proceeds to analyze the Docker Images.



In the "*loadDockerImagesList"* activity, the previously json list files, containing the docker images (in format *namespace/name*) are loaded into memory and for each of them, the appropriate tag is resolved as described in the previous section. After that, the "*downloadProjectsAndProfiles"* activity performs a GET request to the CAP API for downloading all the projects and profiles (name and id, where the latter one is the only type of information requested in each *GraphQL* query to use the CAP features).

Now we have the references to check the online traces and reports on the CAP platform and check if there are any suspended traces or reports from previously interrupted/failed runs of the program by checking and comparing the local files. Essentially, this phase prepares all the lists used from now on as described in the previous section (3.5.1.2) and it defines the counters used for managing the work-life of each thread used in this program.

As it can be seen from the diagram, 5 threads are created for different purposes, but actually, this diagram has an abuse of notation because there are [*#DockerImages + 4] threads* that are managed through a Thread Pool, by the Boost library. Specific information about this can be found in the next section (3.5.4).

After the join of all the active threads that successfully quit their activities through some race conditions, the thread pool will be empty and now we have all the reports on the *Cryptosense Analyzer Platform* ready to be downloaded. Before downloading them and saving them locally to improve debugging and performance times (i.e. "*Fetching/Updating Reports"* activity), some will be locally written some saving files to keep track of the images that failed the analyzing process in the Docker-Scan phase or during the CAP phases to generate a report. These files are made for debugging purposes. In the last activities, the program will perform some statistical analysis ("*Collecting Stats*") and JKS analysis will be launched to determine if the vulnerable JKS file also contains private keys or just the public ones ("*JKS Checking*").

3.5.4. Analyzing Concurrently

In this section will be covered, briefly, some descriptions and notes of the most important threads used in run time. This program is mainly written in C++ for adding concurrency to boost and optimize performances.

3.5.4.1. Scan Thread

There is not a single “Scan Thread” but multiple threads with the same code are created in the thread pool (i.e. a thread for each docker image), the purpose is to boost the time performances and avoid sequentiality or contention since this part takes a lot of time. All lists, mutex, and other variables are passed by *reference*. Instead, the (docker) Image object (like *namespace*, *name*, *tag*, *trace path,* and *trace ids*) are passed by *value/copy* since each new thread must have its copy of the struct, otherwise spawned threads will work on the same image reference.
The spawn of these scan threads must take place *after* creating the other 4 threads, this is because the thread pool is built using a FIFO queue, so we need to avoid that the following threads are executed at the end, making concurrency useless.
Image scan is launched through an *execve* call by using a bash script which performs these steps:

a) Pull the images from Docker Hub archives.

b) Run the docker images by creating a detached container in interactive-terminal mode with root privileges (required to save and run host scanner in some docker images).

c) Copy host-scanner from the host to the container.

d) Execute the host-scanner command line through the `docker exec` command in the background but wait for its completion.

   Note: the waiting completion is made by the host kernel and *not* by Docker.

e) Copy the generated trace file into the host.

f) Terminate and delete the container.

An elegant alternative – with no benefits – is to dynamically generate a *Docker Compose file* that automatically executes these commands.

Unfortunately, as already anticipated, the implemented solution suffers from a few sequentiality constraints imposed by the Docker Command Line Interface, since it uses a FIFO queue of the commands and executes them sequentially. One possible – but not implemented nor tested – the solution is to install multiple docker engine instances and spread these threads on these engines, but it will require much more CPU compute power and RAM space.

3.5.4.2. Upload Thread

In this thread, each docker image labeled as “scanned” (i.e. the trace file exists, it is valid thus the image is into the *images_scanned* queue) can be uploaded to the CAP. The upload consists of two phases: get the AWS S3 server authentication and then upload the file with that authentication information. The upload process ends with an XML response. This is the only part of the process that we need to deal with XML objects instead of json. Each file/trace uploaded is compressed so bandwidth usage and upload time is reduced.
This thread breaks its life cycle when the counter of all images to process/upload reaches 0.

3.5.4.3. Import Thread

After a trace is uploaded to the S3 server, it must be imported into the CAP application and in particular into the Organization/User/Project that we need to specify. Here new projects can be created on the fly if the current one is full. The json response can have three different outcomes: the trace import is *pending* or *failed* or *done*. When all the traces are done or failed, the import thread breaks its life cycle and it can be joined.

3.5.4.4. Report Generation Thread

If the trace is successfully imported, then an analysis and report generation can be executed. This part is trivial since we could get a generic error or get the given report ID. A clarification: having a report ID doesn't mean that the report is *ready*. A thread enters the join status after completing all the report requests present in the queue.

3.5.4.5. Check Report Thread

This last thread checks, by report ID, if the report is *pending*, *done* (ready), or *failed*. Generally, failed reports are associated with invalid traces, but this will not happen since both the upload and scan phases check the

trace validity and avoid this kind of failure. Once all the reports are *done* or *failed*, the thread breaks its cycle and it can be joined.

3.5.5. JKS Analysis

As already mentioned and anticipated in Section 2.2.3.2, we already know that JKS is weak and our only concern is restricted only to the private key presence in that Keystore/truststore. Since many instances are generated for different keys in a Keystore but refer only to the same file (Section 3.4), a collection of all JKS paths for each Docker Image is made, then:

- For each list, a new container is run;
- For each path, the script creates temporary directories in the host computer and copies the JKS into the temporary directory of the host;
- For each JKS copied, Floyd's .jar utility for extracting the private keys is executed;
- For each *hash.txt* file generated by the .jar utility, if it contains at least one row of text (actually private key hashes), then mark this JKS as critical (since it contains private keys) otherwise (i.e. no rows are present in the hash.txt file) then mark the JKS as non-critical since it contains only Public Keys.

4. Analysis' Results of Docker Images

Analyzing the results obtained from the previously discussed analysis it can be a little bit trivial since there is a lot of automatism that needed to be introduced after several manual research, especially on what type of results we could get from the analysis.
In the following section, it will be presented a cheatsheet table of the found *non-conformities* (the so-called "*instances"* in CAP), which describes the original purpose of such non-conformity, and how this can be considered a vulnerability, and its possible fix.

4.1. Non-conformities Cheatsheet

As already mentioned in previous sections, CAP provides for each instance a critical level according to the "*Cryptosense 2022*" profile that I have used to generate the reports. This profile includes many and most restrictive rules of other organizations such as FIPS and NIST.
The critical levels are encoded with a color that will be used for each non-conformity found and reported in this document:

> *PASSED*[57]:
> The instance conforms to all the profile rules and this is *not* a vulnerability.
>
> *LOW*:
> The instance is a non-conformity to some profile rules and its danger is classified as *low* due to the low probability *chances* to enabling attacks through this instance.
>
> *MEDIUM*:
> The instance can be classified as a mid-danger vulnerability, not necessarily an *actual* dangerous threat but if this vulnerability is exploited it could lead to serious security leakage.
>
> *HIGH*:
> The instance is *already* an issue and the exploit can be performed at any time, meaning that the system is already exploitable and weak.

The analysis of these instances aims also to partially "*mistrusts*" the critical level of such vulnerabilities that are not a problem due to their original and intended purpose, despite being classified even at a *high* level.

[57] Since the theme of the thesis is cryptographic vulnerabilities, all the "*PASSED*" instances are not discussed nor reported in this document.

The first type of instance regards the “*No encryption of private keys”*. Among all the scanned and reported Docker Images *only one image* has this type of instance: library/hitch:1.7.2-1.

> *Instance*: **Unencrypted** RSA/PKCS-8 **private key** at */etc/ssl/private/ssl-cert-snakeoil.key*
>
> *Purpose*:
> After installing Apache2, a post-install script of the ssl-cert package will install a *snakeoil*[58] certificate for default HTTPS configuration when *no SSL certs* are installed. It helps to ensure encryption but is insecure since it lacks a root authority signature.
>
> *Attack*:
> The key must be leaked internally. Once obtained the key, encryption, decryption, and impersonations are enabled.
>
> *Fix*:
> Usually this certificate and key must be insecure and it is checked by a snake oil user. It should not be removed unless it is used (no SSL certs are installed or configured), in that case, the Docker Image comes already with a cert+key configuration which is the same for all Hitch users, thus running the command:
>
> make-ssl-cert generate-default-snakeoil
>
> helps in ensuring an encrypted communication for the short time required to get/config the SSL certs.

[58] *Snakeoil* is a term intended as “fake” or “to be not trusted”.

Some Docker Images have *expired* certificates, most of them are CA certificates and the rest of them are non-CAs self-signed certificates whose purpose is to provide some functionalities to the applications inside the Docker Image, like all Node.js-based Docker Images. Many of the certificates that expired, in the latter scenario, expired in April 2022. See Section 4.2 for more information about Docker images differences in time.

> *Instance*: **Expired** X.509 certificate at /usr/share/ca-certificates/mozilla/Cybertrust_Global_Root.crt
>
> *Purpose*:
> CA certificates provided by Mozilla. CA certificates must be installed when the chain of trust is used for trusted third parties.
>
> *Attack*:
> Not directly a security risk, but invalid certificates lead to impersonation of the website, thus exposure for end-users to fake websites and annexed malware/viruses have more chance to be installed. Fake CA roots certs are dangerous and could partially break the chain of trust.
>
> *Fix*:
> Remove all (CA) certificates expired (or not valid), get them from CCADB[59], and install them manually (or via trusted scripts).

[59] See Paragraph 6.3 for more details.

Another issue regarding *valid* certificates is their long validity. CAP profiles have set the maximum validity field, up to 3650 days (10 years). Many root authorities set their expiration longer than 10 years (sometimes it is just 10 years and some days). This in reality was a common choice before 2020, when Apple and Google (which are the most influential browser-maker companies) announced that starting from September 2020, all certificates longer than 397 days (almost one year) are not *issued* anymore. This is due to some security concerns discussed below but also because of *forcing* the transition (rolling out) from SHA-1 to SHA-2 (or others).

> *Instance*: X.509 certificate at /usr/share/ca-certificates/mozilla/Amazon_Root_CA_4.crt with **more than 3650 days validity.**
>
> *Purpose*:
> CA certificates provided by Mozilla. CA certificates must be installed when the chain of trust is used for trusted third parties.
>
> *Attack*:
> The longer the validity, the higher are the chances to obtain the private key [31]. This could be related also to the algorithm used for encrypting the signature, if the algorithm is suddenly weak for a 0-day vulnerability this is not good, for instance, the SHA-1 is nowadays considered weak.
>
> *Fix*:
> Check if there are new certificates with shorter validity, if so, take action by removing and installing the shorter one. Usually, CAs have limited validity but old CA certs with longer validity are still used and they do not need to be replaced. In extreme cases, a certificate revocation is performed as a countermeasure, thus a revocation check must be done.

As we can expect from certificates with too long validity, there are public keys associated (thus private keys) that must be untouched for the same validity period. The stale public (and private) key is not a good thing and, as we can see below, the criticality level depends on the type of key (for what is used) and its “age”:

Instances (2):

- **Stale RSA (or DSA) public key** in PGP file at /usr/share/keyrings/debian-archive-*removed*-keys.gpg

- **Stale RSA (or DSA) public key** in PGP file at /etc/apt/*trusted.gpg.d*/debian-archive-buster-automatic.gpg

Purpose:
Debian Public Keys are used for managing Debian archives and for trusting packaging sources, see Section 2.2.3.1 for more details.
Usually, Debian removes old keys from previous releases and keeps them stored in a .gpg file. This is probably because for manual checking digital signatures of old files, no motivations are officially provided by Debian. Removed keys are extremely old (pre-2010), thus the red color.

The second instance is old stale keys, actually used, but created in the pre-2020s. This explains the orange color.

Attack:
Stale public keys means also stale private keys which is not good since it gets a higher chance to be compromised.
Keyrings are managed by automated processes like Debian apt-secure and maintained by Debian teams. Removed keys are intentionally left and probably can be removed without any undesired effects if the user knows that they will not be used.

The second instance is possibly a vulnerability because apt did not update any keyrings for a long time; not a critical security risk, if the docker image is used nowadays, but the Docker Image must be updated with new keyrings if it will be used in the future.

Fix:
Usually, a

```
sudo apt update
```

should update all the keyrings. Security updates also do this kind of keyring refresh/update.

> For updating the *Debian Archive Keyring* in a safe environment:
>
> `apt-get install debian-archive-keyring`
>
> If we are extremely paranoid we can get the desired keyring to update/install from the Debian FTP key page, and check each fingerprint.

In section 2.2.1 we already mentioned the importance of keeping longer key lengths to avoid brute force attacks in a reasonable time, and despite the lack of an international, also unique, standardization of key lengths (read more in Chapter 6.4) many organizations follow the recent results of brute force (and encryption algorithms) attacks to avoid any security issue regarding cryptographic keys, by imposing a common consensus like deprecating 1024 bit keys:

> *Instance*: **1024 bit RSA/DSA public key** in PGP file at /usr/share/keyrings/debian-archive-removed-keys.gpg
>
> *Purpose*:
> As in the previous instance, these kinds of .gpg files are old public keys used by the old Debian Release, left for unknown official reasons. The guess is that they are left for manual checking digital signatures for old files.
>
> *Attack*:
> By the age of these keys, many of them had shorter lengths. Nowadays 1024 bit RSA/DSA key lengths are not used because they can be easily broken by brute force attacks. Pay services allow to break in few hours 512-768 bit keys, it is possible that some entities with high resources can break 1024 bit keys.
>
> *Fix*:
> In general, dismiss and remove old keys less than 1024 bits. Nowadays recommendation is to use at least 2048 bits and consider the 4096-bit length as it will be the next standard.
> Most of the instances regarding short key lengths are referred to all the keys of these removed keys archives left by Debian.

As already explained, the digital signature on a certificate requires hashing *firstly* the body of the certificate and *then* encrypting the hash with an encryption algorithm. Above we saw how encryption could be harmful with less than 1024 bit key lengths, another concern is about the hash functions. SHA-1 (Secure Hash Algorithm 1) is (still) the most used, worldwide, and it is cryptographically broken since 2017. Already in 2004, there were some (theoretically) concerns about its collision strength, in 2011 NIST decided to deprecate SHA-1 for signing certificates starting from 2013. In 2020, chosen prefix attacks in SHA-1 are practical and such attacks allow "*colliding messages with two arbitrary prefixes, which is much more threatening for real protocols*" **[33]**.

> *Instance*: Use of **SHA-1** digest algorithm in self-signed X.509 certificate at /usr/share/ca-certificates/mozilla/EC-ACC.crt
>
> *Purpose*:
> CA certificates provided by Mozilla. CA certificates must be installed when the chain of trust is used for trusted third parties. As explained in Section 2.2.2, certificates use hash algorithms for the digital signature on them.
>
> *Attack*:
> In February 2017 there was found and proved the first public collision using SHA-1. For this reason, is now considered insecure and should be avoided even if the chances to create a public collision are low. Once the collision is done, the private key is leaked and as consequence, all previously discussed scenarios can be actuated.
>
> *Fix*:
> If it is a self-signed certificate of your possession, update the certificate by changing the digital signature encryption algorithm with other algorithms such as SHA-256. CA should update their certificates and then end-users must follow the updating process through CCADB, but for now, is not a real necessity (due to low probabilities to do a public collision) and many CA certs will end nearly (in 2026-2027).

One of the biggest flaws in the IT world is that almost every integrity check on files on the internet is based on MD5 hashes, and still now it is widely used. According to [34], MD5 was introduced in 1991, and already in 1996, there was a collision, suggesting to migrate to SHA-1. Between 2004 and 2006 various attacks and algorithms for forcing a collision were made and improved until the "tunneling" method-based attack [35] created the collision in just one minute. In 2019, ¼ of the most used CMS still use MD5. Fortunately, MD5 was not found in any CA certificate in my analysis, although it is still present on some 3rd parties certificates in some keystores:

> *Instance*: Use of **MD5** digest algorithm in self-signed X.509 certificate with label dukecert in JKS Keystore at /opt/java/openjdk/sample/jmx/jmx-scandir/truststore
>
> *Purpose*:
> Despite being a path of a sample provided by OpenJDK, the trustore, as mentioned in Section 2.2.3.2, is used to store trusted certificates such as CAs.
>
> *Attack*:
> MD5 hash functions are insecure and there are plenty of resources for breaking these hashes. Generally, collisions in the hash function could lead to the hash payload being accepted as legitimate when it has been altered, i.e. tampered artifacts can be created with the same legit hash.
>
> *Fix*:
> Keystores must be updated to use other hash algorithms such as SHA-256. If you want to still stick with MD5 (due to compatibility issues), consider adding salt[60] and hashing iteratively the produced hashing with MD5.

Oracle, one of the biggest companies in the US, develops and maintains *Java*, a high-level, multi-platform, class-based, programming language. *Java* is distributed through *JDK* (*Java Development Kit*) and provides a great amount of APIs. Among these APIs, Java supports internet communication with its *JSSE* (*Java Secure Socket Extension*) which implements *SSL*, *TLS*, and many useful tools such as encryption and decryption algorithms, server authentication, and message integrity. As can be intuitively known from Chapter 2, to achieve the CIA properties, certificates (and their related public and private keys) need to be used in these *JSSE APIs*. To fulfill the chain-of-trust mechanism, CA must be installed on the system.

[60] Random data added into the input of the hash function to safeguards secrets/passwords.

Oracle itself wanted to minimize dependencies when it comes to Java programs, thus in early 2005, they introduced the known Java Keystore for distributing CA certificates (see Section 2.2.3.2).

Unfortunately, *JKS* are weak:

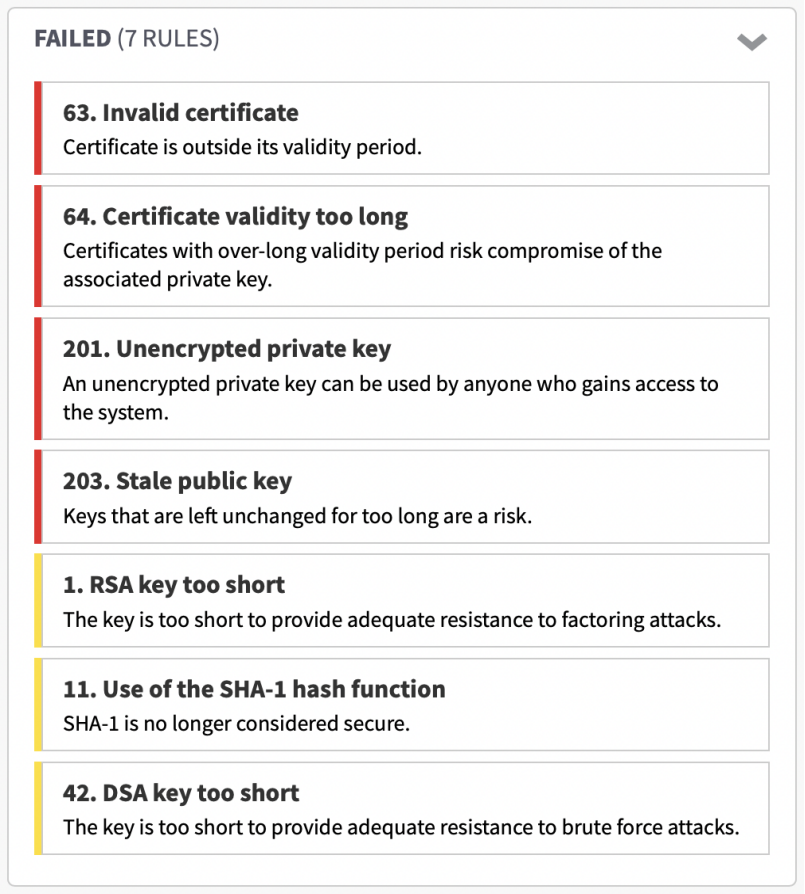
> *Instance*: **JKS** Keystore at /opt/java/openjdk/jre/lib/security/cacerts.
>
> *Purpose*:
> JKS are keystores that keep keys and their related certificates, in a single archive file. In this case, CA certificates are provided in a JKS truststore format for SSL/TLS connections for the Java Sockets. Such JKS are provided by Oracle in all OpenJDK distributions.
>
> *Attack*:
> As already mentioned in the same Section 2.2.3.2, Java KeyStore is a weak insecure format that relies on SHA-1 encryption. Moreover, Oracle itself provided a not-properly fix by enforcing the number of hash iterations but this will only increase the amount of time to break the JKS. This Floyd Github page provides the tools and a guide to breaking a JKS file and in particular decrypting private keys using HashCat. All the CIA paradigm is taken down.
>
> *Fix*:
> Convert the keystores into other formats known to be secure, like PKCS12. There are several ways to do that, through GUI programs like KeyStore Explorer or CLI using Java Keytool:
>
> `keytool -importkeystore -srckeystore old.jks -destkeystore new_ks.p12 -srcstoretype JKS -deststoretype PKCS12 -deststorepass [PASSWORD_PKCS12]`
>
> Java Keytool is provided in any JRE and JDK installations.

4.2. The Evolution in Time

Now we are going to see some reports on the evolution of a Docker Image after some time passed since the previous analysis. This part could have been automatized but since there are too many variables to consider with a short time for implementing it, this idea was not implemented at all.

So, in this section, there are reported only two cases that are selected and performed manually, after a long manual search of the best interesting candidate among the ones used in this research.

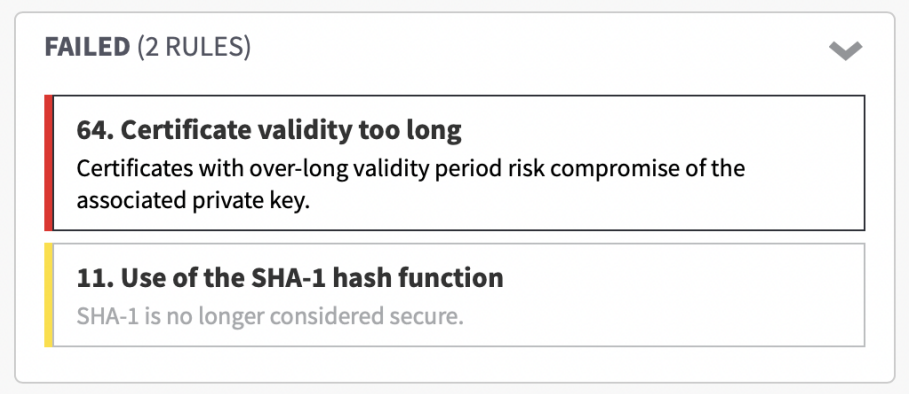
Our first candidate is **Hitch**, an *official* image. More specifically library/hitch:1.7.2-1. Hitch is a "*libev-based high performance SSL/TLS proxy by Varnish Software*"[61].
Hitch was analyzed firstly on April 14th, 2022[62]. The second analysis took place on May 9th, 2022. From April to May there are no differences in terms of vulnerabilities, maybe some differences in the instances, but in a quick look nothing changed at all:



[61] Took from the Hitch - Official Image | Docker Hub page description.
[62] In April, tag management was not yet introduced, so the first analysis taken in April was using the *latest* tag.

So Hitch, for almost 1 month, had several issues (7 rules) that failed to pass the vulnerability check.

The third analysis took place on May 31st, 2022. Almost all Hitch version tags were updated 3 days before, so May 28th, 2022. As we can see from the following image:



The number of rules that failed to pass the CAP check is reduced – significantly – by 5 rules. Only SHA-1 and the long certificate validity rules are left. Since we are considering only the rules, the number of instances for each rule could be a great amount.

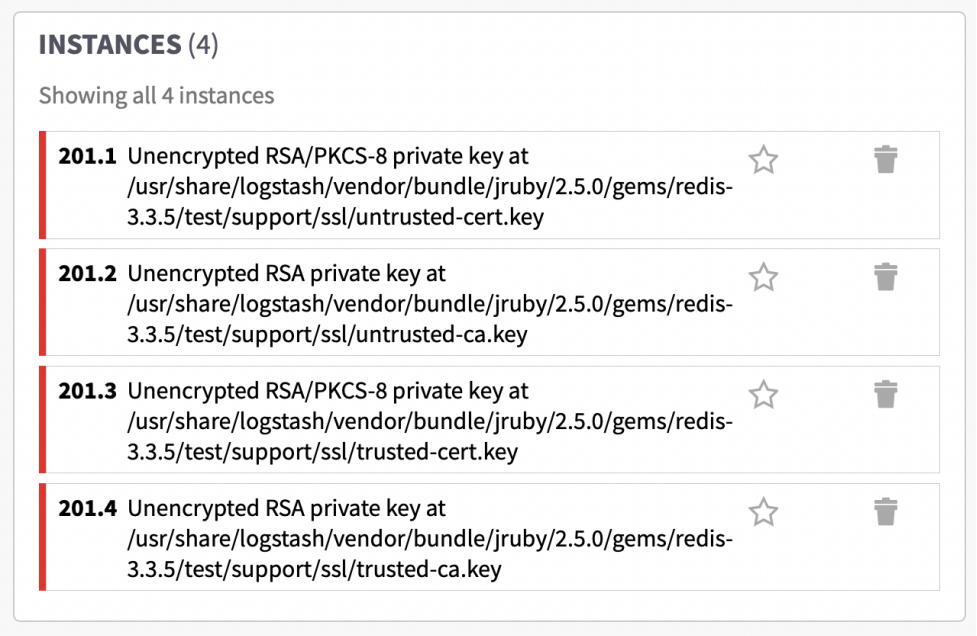
So the Hitch Docker Community, which is responsible for maintaining this Docker Image, worked hard to provide the best and safest environment for Docker users. In particular:

- Expired (CA) Certificates were removed.
    - Thus some stale public keys were removed.
- Unencrypted Private Keys were removed (one was a temporary private key, the snakeoil seen above, the other one was a private key for test).
- Other stale public keys were removed, hence Debian keyrings were updated
    - By checking the keyring folder at /usr/share/keyrings/ all keyrings are updated to February 25th, 2021[63].
- RSA/DSA keys too short were removed
    - This is because all short keys were part of the debian-removed-keys keyring that was probably updated.

[63] Unfortunately, I do not have access anymore to the previous version of the image for checking the DateTime of the keyrings.

SHA-1 encrypted signatures on CA certificates are still present, thus the same CAs with long validity are still present. Unfortunately, there are no changelogs on GitHub or docker hub pages.

The second study case is **logstash**. The logstash version with a serious vulnerability was library/logstash:6.8.19, dated October 16th, 2021. Starting from logstash 8.0+[64] the big issue was solved. The instance was an *unencrypted private key* more specifically private keys used by jRuby[65]:



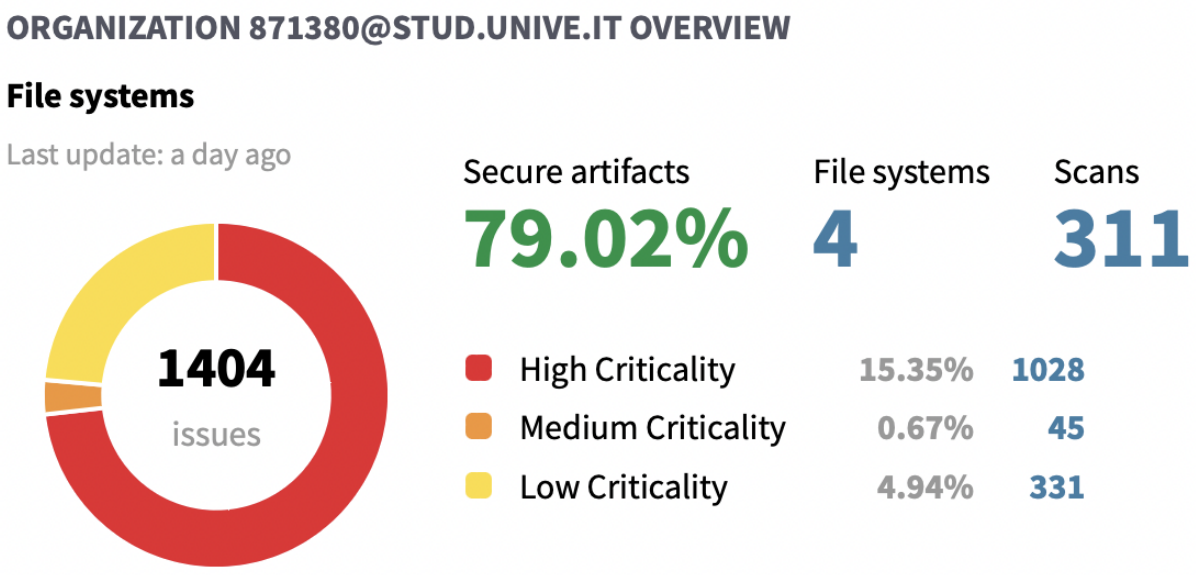
As we can see from the path of these unencrypted keys, these keys were used as test keys, so probably not a very big issue. As said before, after v8.0 and later, this instance disappeared. There is also something else to say: Starting from version 6.8.14, the *Log4Shell CVE scan* was introduced as a Docker safety check scan and a *Log4Shell CVE* was detected in every logstash version until version 6.8.21 and 7.16.1. This CVE seems to be not entirely correlated with the unencrypted private key issue, but *Log4Shell* seems to be related to the Apache Log4j that is based on Java [32].

[64] Repted tests made in versions: 8.0 (May 31st 2022), 8.1.3 (May 9th 2022) and 8.2.2 (May 31st 2022).
[65] jRuby is a programming language (originally Ruby) adapted to the Java Virtual Machine (JVM).

4.3. Statistics

Statistical data are useful to understand how such vulnerabilities are diffused and how such vulnerabilities are *real* threats.

Due to time constraints (see paragraph 6.2), research has been conducted only[66] on Official Images (~120) and *Verified-Publisher* Images (~180). Other constraints bound to the build of some Docker Images and the Host-Scanner compatibility, the number of Docker Images analyzed is reduced to ~300 Images (some of them are excluded for redundant purposes regarding the evolution in time discussed in the previous paragraph). For other ideas discharged from such constraints, see the "*Future Developments"* paragraph in Chapter 5.
The following image is dynamically built by the CAP after these months of research:



As we can see from it, each Docker Image provides, on average, at least ~80% "secure" artifacts, thus certificates, keys, and keystores are *safe* and do *not* suffer from any cryptographic non-conformity case discussed in the cheatsheet section (4.1). Interesting how, always on average, the *High* criticality is the first criticality class in terms of the percentage of issues present (~15%), following the *Low* one with ~5% and *Medium* with a percentage under the 1%. This could also mean that all the reports generated

[66] In the moment of writing the thesis (May-June 2022), Docker also introduced another category called "*Open Source Program*" which could be more useful for such docker communities.

with the "*Cryptosense 2022*" profile are prone to "*emphasize*" such vulnerabilities, which is good from a security point of view.

In reality, we already saw in the non-conformities cheatsheet (**4.1**) that many of the instances are about false positives, such as the *Debian removed keys* keyring (labeled as *high criticality*), or *CA certificates* using long time validity (whose criticality label is set to *low*), thus we can almost deduct that the overall security artifacts percentage can be increased at least by a 10%.

Even if we consider the raw results, another interesting statistic is that the same criticality proportions:

PASSED	LOW	MEDIUM	HIGH
79.02%	*4.94%*	*0.67%*	*15.35%*

That can be rewritten as "*non-passed only*" criticality proportions:

LOW	MEDIUM	HIGH
23.57%	*3.20%*	*73.23%*

They are almost the same for both *Official* and *Verified-Publisher* Images.

Note that for the issues that arose before, only *~120/145 Official* images and only ~180/400 *Verified-Publisher* Images were analyzed. It could be possible that *Verified-Publisher* images could be worse (in terms of proportions) since ~220 images were not analyzed (see Limitations in 5.2).

Here follow the statistic results for each type of instance-class:

- ***In a total of 523 Certificates:***

Self-Signed	CA	Hash Vulnerable[67]	Validity Length (>10 yrs)	Expired	Insufficient[68] Key Length
94.26%	77.25%	39%	84.32%	13.96%	3.63%

Interesting results for Certificates, where most of them are self-signed certificates thus a big piece of that percentage is related to CA certificates.

[67] Supported hashes in CAP are MD2/5, SHA-1/256/224/384/512, and SHAKE-128/256. Hash vulnerable functions considered are the following: MD2, MD5 and SHA-1.
[68] Multiple supported key lengths by CAP, the Insufficient ones are RSA < 2048 bit and EC < 256 bit.

Another interesting and *worrying* data is the *hash vulnerability* share (39%) which is still greater than expected. The majority of the certificates with more than 10 years (84.32%) are CA certificates and root authorities certificates of the company that built the Docker Image. Even if the expired certificates are around 14%, this is probably not relevant information since some Docker Images were updated in the meantime and almost one (that I have verified) updated an expired CA certificate. Still, expired certificates provide security issues and need to be replaced as soon as possible. As already mentioned, a few of the expired certificates are CA certificates and it's up to the Docker community team to update such certificates. The last data, "*Insufficient Key Lengths*" (share of 3.63%) should not be confused with key lengths of Debian keys (in GPG files). Such low share can be related to extremely old certificates, already expired (for example the *brutus.neuronio.pt* certificate expired in *1996*, present in *Bitnami*[69] and *Amazon*[70] images where it comes from a test directory[71] of Python 3.8).

- ***In a total of 952 Keys (certificates + keyrings):***

Private	Type			Insufficient Key Lengths
	RSA	***DSA***	***EC***	
1.58%	67.33%	26.89%	4.83%	29.83%

A small percentage of these keys are *private* keys, all of them are coming from test directories[72], especially from *pygpgme* [36]*:*

> "
>
> *PyGPGME is a Python module that lets you sign, verify, encrypt and decrypt messages using the OpenPGP format. It is built on top of the GNU Privacy Guard and the GPGME library.*
>
> "

[69] From the following Docker Images: `bitnami/node-snapshot:12.22.8-debian-10-r20` and `bitnami/python-snapshot:3.6.15-debian-10-r129`.
[70] From the Docker Image: `amazon/aws-sam-cli-emulation-image-python3.8:2022-04-27`.
[71] The directories are:
`/var/lang/lib/python3.8/test/capath/*`
[72] The most frequents were the following:
`/var/www/html/addon/securemail/vendor/singpolyma/openpgp-php/tests/data/*`
`/usr/share/doc/pygpgme-0.3/tests/keys/*`

Another interesting fact is the short diffusion of Elliptic Curves keys, which requires shorter lengths to have the same strength as a high number of bit lengths required for RSA/DSA keys. This time, the considered keys are not just the ones present in certificates but also keyrings, especially the *debian-removed-keys.gpg* keyring that has the biggest share of old keys with shorter lengths. Just for completeness: *all* the insufficient keys are RSA/DSA keys with 1024 bits.

- ***In a total of 33 JKS files (into ~311 images):***

Although JKS are especially diffused as keyrings in general, it's interesting how the ~99% of them:

- Contain *only public keys*
- They are provided by *JDK* installations.

Just one JKS found contained a private key in a *Groovy*[73] Image, and it was a key for sample/test purposes.

~98% of them were cacerts files, the ~1% were truststore files and the remaining ~1% were generic keystores.

By keeping in mind all these statistics, the overall situation is not bad at all, especially considering that I have only analyzed Docker Images supposedly to be in a good state since its official status. The only real issue of this overview is that the end user should not use those left keys and certificates as production tools, which is extremely impossible to do by "mistake" in a scripted way.

[73] In particular this image: *library/groovy:jdk8*.

4.4. Still, Open Questions

Many questions and doubts were dissipated during the previous sections regarding docker images, but there are still a few opened questions related to the parties involved in this analysis process:

- Why does Oracle *not* provide a PKCS12 Keystore format for cacerts in its JDK installations?
- Why do some 3rd parties software provide non-conformities tools such as *unprotected private keys* or *short keys* for *test* purposes?
- How are the *real*[74] proportions of all *Verified-Publisher* Images?
- How much are different the results compared from community[75] images?
- Why is there still a large diffusion of SHA-1/MD5 hash functions, especially when it is used for security mechanisms?
- Why is there no organization that standardizes whose CA/roots authoritative certificates are “safe[76]”?
- What is the real purpose of keeping the removed Debian archive keys in the latest releases?

These are only a few questions that the companies and organizations themselves cannot provide, nor directly and not transparently, especially when they are security-oriented questions that must be a concern for everyone.

[74] Intended as proportions of all 400+ scanned images available on Docker hub.
[75] All the images uploaded by the Docker community, not marked as official nor uploaded as verified publisher.
[76] See Paragraph 6.3.

5. Conclusions

This chapter concludes the thesis and internal stage work for Ca' Foscari University of Venice, part of the conclusion were already given in the previous chapter due to evidence of the given results. Following this conclusive chapter, a background section is given for explaining more in-depth some aspects that were previously simplified for readability and for adding some interesting elements into the context of some entities related to this thesis work.

5.1. What Has Been Learned

The whole project was biased on CAP capabilities to provide a fast and flexible analysis that consisted basically of certificates, keys, and keystores.

In the *Cryptographic and Cyber Security* sections many notions were introduced to give also a motivational explanation of why these three objects were relevant and important to consider, and much useful information was discovered because of understanding the results of analysis, such as the JKS vulnerability (see 2.2.3.2); the Debian organization into teams and their internal processes to ensure that such teams could provide the best security effort for their operative system and how this one works in terms of security and cryptography.

The implementation phase of the automated script program for analyzing such images revealed a lot about how the *Cryptosense Analyzer Platform* works and what are their strengths points, whose points can be found also in Section 6.1 ("Some Additional Information … on CAP").

The analysis part revealed a lot about how these services (i.e. CAP) provide different levels of analysis (i.e. profiles) and how much these results can easily trick the end-user into believing that the scanned Docker Image is potentially a security problem… When in reality most of the instances detected are just left for unknown *official* reasons and probably (but no guarantees) could be easily deleted, especially test directories (see cheatsheet) that are full of non-conformities and provide only noise.

In the statistical analysis part there is a full overview of many problems related to standard transitions (i.e. broken hashes functions), which is a common problem in the IT industry, and also minor issues regarding the

distribution of CA certificates in a *simple, easily accessible,* way (see also the Backgrounds on CA Certificates).

The indirect results learned from the analysis revealed also how the Docker Community works on their products in a certain amount of time (see The Evolution in Time); there is also a way for Docker itself, as an organization, to improve the security of its hub.

Though the results were not explicitly shocking, it was reassuring to see that – at least for the ones analyzed – the results were good and there could be a slight chance to improve cryptographic safety/cleanliness as the community members of the Hitch image proved in these months (see the first study case of section 4.2).

5.2. Assumptions and Limitations

Some assumptions and limitations were already written in many parts of the thesis.

The first assumption is that our target images were the most used and downloaded, thus the original ones. Despite this motivation, the assumption is enforced due to technical limitations explained below.

Another assumption is that the entire analysis is biased on CAP and Docker, so their limitations would affect the thesis work: the statistics and instances types are retrieved (manually) from the ones discovered by the actual analysis results explicitly written in the CAP reports.

Implicit CAP limitations were imposed by the required dependencies of the Host-Scanner executable for running which could not be used in many Docker Images. As already mentioned for the number of images scanned out of its total, the statistics and final results could not be exactly compared between the Official images and Verified-Published images though they were still interesting.

At the API level of the CAP, it is not possible (yet) to change the project size to its maximum number of traces (which is 100 for every kind of license) right after a new project is created, since its default value is 20. This default behavior will fill up the project slots when using automated scripts/programs like the ones created for this project. Right now the only way is to use the Web UI and manually edit the project limit.

Technical limitations were applied due to Docker constraints such as the number of pull requests, tag versioning, and platform compatibility with Arm64/v8 processors. Other limiting factors to the C++ implementation were applied due to the huge amount of data available and the (almost) infinitely many ways of how this data could be formatted, thus some analyses like certificate revocation, JKS cracking (with HashCat), and Docker Image sanitizer (for cleaning/solving all detected issues) were not implemented.

Due to time constraints and the aforementioned limitations, it was not possible to implement the difference checks between one image and its updated (or previous) version, that is why few checks were made manually, although it would have been easier and probably more interesting to analyze a trend in this scenario.

Informative limitations are bound to companies, organizations, and web transparency: most of the information found, was a little bit difficult to find, mostly because of the highly fragmented documentation (i.e. Debian).

5.3. What Impact Should We Expect

From an optimistic point of view, we should expect that the Docker Community of each Docker Image should *act more quickly* and *use these platforms for detecting security vulnerabilities*. *Awareness* of security concerns must be spread not only to local hosts and not be confined to non-cryptographic scenarios.

Docker already provides a Docker Scan feature[77] **[37]** but it's an *optional* plugin, not so advertised, moreover, the provided results are more or less approximated to only a few CVE defined by the Snyk engine itself and lack a *dedicated* platform to view, manage and report (automatically) such vulnerabilities detected, especially with UI for basic end-users that do not have the cryptographic/cyber security knowledge. Thus, Docker should give users an alternative way (possibly automated and user-friendly) to verify the Images (and Dockerfiles) in its Hub, as already suggested in 5.1.

5.4. Future Developments

The C++ project can be optimized and extended into researching more images (the community ones), providing fixes to existing images, and then

[77] Docker Scan runs on Snyk engine.

being pushed into Docker Hub. There could also be a way to improve the number of scans of the docker images by reducing the missing dependencies between the image and the host scanner program.

The Venafi integration (See in Section 6.1) in CAP is tempting because it could enable the certificate revocation check for an extended analysis. Other certificate improvements can be made by providing also the new CA to update/install through the CCADB list in an automatic way.

A possible way to speed up the process is to deploy the project into a cluster of servers by creating a cooperative computing layer for splitting the workload among the servers, especially with different proxies since the main bottleneck (also in terms of policies) is Docker Hub.

Docker itself still has some work to be done: starting from the API infrastructure that exists and works only if the user knows what images it wants, and ending with the possibility to introduce other vulnerabilities-scan engines by integrating them and providing less spartan[78] solutions.

Last but not least important, is to build a global database (maybe run by Docker) of the instances/issues detected, collect them and categorize them in the correct criticality, for example: if it is an unused key or something important left by mistake. The purpose of such a database is to give correct information about how images can be sanitized or corrected.

[78] json reports only through shells.

6. Some Additional Information …

In this section there are a little information and fun but not-so-fun facts about some entities involved in the project.

6.1. … on CAP

- During the first implementations of the projects, some mechanisms were found difficult to deal with the CAP APIs. Still, fortunately, the Software Engineers in Cryptosense listened to my feedback and they also provided new features to CAP, like the deletion of the report via API and the new Certificates Navigation page which helped a lot in the manual research of suspicious certificates.
- An interesting feature, that was barely mentioned in the whole project, is the Venafi Integration. Venafi is a company that manages and provides services regarding machine identities. Among the offered services, there is integration into CAP for analyzing certificates found in reports and checking their revocation. This integration is useful but unfortunately requires a Venafi subscription for obtaining an API token to be put into CAP.
- In Section 2.2.3.2 we already saw that JKS vulnerability/weakness was discussed in **[23]**, among the links provided by Floyd, there is also this link **[26]** for Cryptosense's CEO that explains JKS weakness and how this vulnerability is now a fixed check on its *Cryptosense Analyzer Platform*.

6.2. … on Docker

- There exists a *pull* limit of up to 100 pull requests every 6 hours; for signed-in users into the Docker Dashboard application, this limit is extended up to 200 pull requests/6hrs.

- If we pull a docker image without any tag, the Docker Hub system checks the platform compatibility with the default "*latest*" tag, but if this *does not* succeed, it will run another pull request by itself for checking the appropriate platform, but this will probably fail again because tag associated to the platform were used improperly by their publishers, often creating one tag for each version and each platform.

  ➢ The counter of the pull requests (one request made by the user + more than one made by the server) will be increased unjustifiably, for this reason, limiting more pull requests.

- An issue about certificates is the lack of automatism to update CA certificates in the Docker images buildings, especially the deprecated ones. Deprecated images can be divided into two categories: the ones that cannot be updated due to no active members in charge of maintaining that image, or because the owner itself deprecated the image because of product evolution. It would be nice if Docker provided this option for deprecated images.

- There are still problems with the Apple Silicon platforms, more specifically ARM64/v8 platforms: many tags are associated with the variants of the ARM64 processors, but the problem is that many times the manifest file associated with that docker image is not properly built, Docker should establish some standard *rules, and not simple guidelines,* on how the publisher should use properly tags and create manifest files in the right way.

  Note: Among the ARM64/v8 processors, the Apple Silicon can still run AMD64 containers but, due to the malformation of the manifest file, some of these will not run even if the platform supports that.

6.3. … on the CA Certificates

- Roots Certificates, or CAs, are provided by some organizations, like Mozilla in their *CA Certificate Program*. Mozilla provides this root store (CAs database) for embedding the trust anchors into its product like Firefox and Thunderbird. Mozilla is maintaining its root store with a lot of effort. When in Section 4 we were asking how to update/replace CA root certificates, Mozilla allowed us through the *CA Certificate Program* to see the adopted policies and get a list of PEM certificates of CAs, Intermediate Certificates, and Removed CAs. There is a problem: Mozilla provides all the certificates but some of them are not trusted by Mozilla itself **[28]**, and these "*distrusted*" certificates are managed via their products, not by the root store. For example in **[29]** we can find a conversation among Mozilla developers taking a distrusting action after a proposal consensus (among browser makers) on distrusting *Symantec Roots* certificates. Thus, always Mozilla in **[28]**, suggests *not* using *all* of the provided root certificates in the CA Certificate Program but using the root certificates in the "*Common CA Database*".

- The *Common CA Database* **[30]** is a repository for maintaining a list of trusted root certificates among the CCADB members. This organization is run by Mozilla but active members are also Google and Microsoft. The fact that: Mozilla itself runs its root store; it will not trust all of the certificates in it; and only three members maintain the CCADB; emphasizes the lack of standardization for trusting the root certificates, despite the effort made by browser makers trying to reach a consensus for distrusting these CAs.

6.4. … on Key Lengths

- Key lengths are *almost* standardized due to mathematical agreements among researchers but their status as recommended or deprecated is *not*. In [27] we can see that many different organizations have different minimum/suggested key length requirements that keep evolving.
- Theoretically, actual computing models cannot break anything in a reasonable time, but quantum computing aims to solve such computing problems (like factorization for breaking RSA encryption) in a short time. This is truly a problem for cryptography but since quantum computing is *far away* from reality and commercialization, this problem “is not a problem”.

7. Bibliography and Sitography

7.1. Basic Information and Definitions

[1] Cryptosense. “About Cryptosense.” *Cryptosense*, 2022, https://cryptosense.com/about.

[5] Wikipedia. “Cryptography.” *Wikipedia*, 2022, https://en.wikipedia.org/wiki/Cryptography.

[6] Wikipedia. “Key (cryptography).” *Wikipedia*, 2022, https://en.wikipedia.org/wiki/Key_(cryptography).

[7] Arampatzis, Anastasios, and Sandeep Singh. “Diffie-Hellman Key Exchange vs. RSA Encryption.” *Venafi*, 14 July 2020, https://www.venafi.com/blog/how-diffie-hellman-key-exchange-different-rsa.

[10] Smirnoff, Peter, and Dawn M. Turner. “Symmetric Key Encryption - why, where and how it's used in banking.” *Cryptomathic*, 18 January 2019, https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking.

[14] Wikipedia. “Elliptic-curve cryptography.” *Wikipedia*, 2022, https://en.wikipedia.org/wiki/Elliptic-curve_cryptography.

[15] Wikipedia. “Public key certificate.” *Wikipedia*, 2022, https://en.wikipedia.org/wiki/Public_key_certificate.

[16] Mozilla. “CA/FAQ.” *MozillaWiki*, 30 December 2022, https://wiki.mozilla.org/CA/FAQ#What_are_certificates.3F.

[25] Tucker, Ben, and David Svoboda. “ENV33-C. Do not call system() - SEI CERT C Coding Standard - Confluence.” *Confluence*, 30 November 2021, https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177.

[27] Giry, Damien, and Jean Quisquater. “NIST Report on Cryptographic Key Length and Cryptoperiod (2020).” *Keylength*, https://www.keylength.com/en/4/.

[28] Mozilla.org. “Beware of Applications Misusing Root Stores - Mozilla Security Blog.” *The Mozilla Blog*, 10 May 2021, https://blog.mozilla.org/security/2021/05/10/beware-of-applications-misusing-root-stores/.

[29] Mozilla Developers. *Wikipedia, the free encyclopedia*, https://groups.google.com/g/mozilla.dev.security.policy/c/FLHRT79e3XE.

[30] *Common CA Database by Mozilla*, https://www.ccadb.org/.

[31] Bisson, David. “4 Reasons Shorter Certificate Validity Periods Are a Good Thing.” *Venafi*, https://www.venafi.com/blog/4-reasons-why-shorter-certificate-validity-periods-are-good-thing.

[32] Okamoto, Hideki. “Log4j – Apache Log4j Security Vulnerabilities.” *Apache Logging Services*, 23 February 2022, https://logging.apache.org/log4j/2.x/security.html.

[34] Tao, Xie, et al. “MD5.” *Wikipedia*, https://en.wikipedia.org/wiki/MD5.

[36] Henstridge, James. “pygpgme · PyPI.” *PyPI*, 8 March 2012, https://pypi.org/project/pygpgme/.

[37] Docker Inc. “Vulnerability scanning for Docker local images.” *Docker Documentation*, https://docs.docker.com/engine/scan/.

7.2. Teaching Materials

[3] Focardi, Riccardo. *[CM0475-1] SECURITY 1 (CM9) - a.a. 2020-21* | *"Basic Concepts"*. 2020.

[4] Luccio, Flaminia. *[CM0480] CRYPTOGRAPHY (CM9) - a.a. 2020-21* | *"Lecture 1", “Lecture 11”*. 2021.

[8] Calzavara, Stefano. *[CM0475-2] SECURITY 2 (CM9) - a.a. 2020-21* | *"HTTPS"*. 2021.

[13] Salibra, Antonino. *[CM0525] CRYPTOGRAPHY FOUNDATION (CM9) - a.a. 2020-21* | *"Finite Fields and Elliptic Curve" & "Factorisation I & II"*. 2020-2021.

7.3. Debian References

[17] Code GmbH. “Using the GNU Privacy Guard.” *GnuPG*, 2017, https://gnupg.org/documentation/manuals/gnupg/.

[18] Wolf, Gunnar. “debian-keyring / keyring · GitLab.” *Debian Salsa*, 26 April 2022, https://salsa.debian.org/debian-keyring/keyring.

[19] Debian.org. “Distribution Archives.” *Debian*, 10 September 2021, https://www.debian.org/distrib/archive.en.html.

[20] Debian.org. *README for the debian-archive-keyring package*. 2021, /usr/share/doc/debian-archive-keyring.

[21] Debian.org. “apt-secure(8) — apt — Debian bullseye.” *Debian Manpages*, 10 June 2021, https://manpages.debian.org/bullseye/apt/apt-secure.8.en.html.

[22] Debian.org. “7.5. Package signing in Debian.” *Debian*, https://www.debian.org/doc/manuals/securing-debian-manual/deb-pack-sign.en.html.

[24] Debian | FTP-Master Team. “ftp-master.debian.org Archive Signing Keys.” *FTP Master*, https://ftp-master.debian.org/keys.html.

7.4. Publications

[2] National Institute of Standards and Technology (NIST). “FIPS 199, Standards for Security Categorization of Federal Information and Information Systems.” *NIST Technical Series Publications*, February 2004, https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.199.pdf.

[9] Calzavara, Stefano, et al. *Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem*. 2019. *Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem | IEEE Conference Publication | IEEE Xplore*, IEEE Symposium on Security and Privacy (SP), https://ieeexplore.ieee.org/document/8835223. doi: 10.1109/SP.2019.00053.

[11] Hellman, Martin E. “AN OVERVIEW OF PUBLIC KEY CRYPTOGRAPHY.” *IEEE Communications Magazine*, vol. 16, no. 6, 1978, pp. 42-47, https://netlab.ulusofona.pt/im/teoricas/OverviewPublicKeyCryptography.pdf.

[12] Barker, Elaine. “Recommendation for Key Management: Part 1 - General.” *NIST Technical Series Publications*, NIST, 5 May 2020, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf.

[23] Floyd. “Java Key Store (JKS) format is weak and insecure (CVE-2017-10356).” *floyd's*, 19 September 2017, https://www.floyd.ch/?p=1006.

[26] Steel, Graham. “Blog - Mighty Aphrodite - Dark Secrets of the Java Keystore.” *Cryptosense*, 21 April 2016, https://cryptosense.com/blog/mighty-aphrodite-dark-secrets-of-the-java-keystore.

[33] Leurent, Gaëtan, et al. “SHA-1 is a Shambles∗.” *Cryptology ePrint Archive*, 2020, https://eprint.iacr.org/2020/014.pdf.

[35] Klima, Vlastimil. “Tunnels in Hash Functions: MD5 Collisions Within a Minute.” *Cryptology ePrint Archive*, 2 April 2006, https://eprint.iacr.org/2006/105.