h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

University of applied sciences

– Department computer science –

Detection of embedded secrets in Docker images

Final thesis for the attainment of the academic degree

Bachelor of Science (B.Sc.)

Presented by

Christoph Linse

Student number: 753086

Advisor	:	Prof. Dr. Christoph Krauß
Co-Advisor	:	Prof. Dr. Alois Schütte

DECLARATION

I declare that the thesis has been composed by myself and that the work has not be submitted for any other degree or professional qualification.

I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included.

I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others.

My contribution and those of the other authors to this work have been explicitly indicated below.

Darmstadt, 06.03.2020

Christoph Linse

ABSTRACT

Nowadays cloud native computing is becoming more and more popular. Docker's application containers in particular are an essential component of cloud native computing. Every container is based on static data structures, so-called images. These images form an important but also critical element in this eco-system. Images are stored in dedicated cloud environments or temporarily in other public or private container registries. This leads to problems if the developer has integrated symmetric or asymmetric secrets into the image like SSH keys, clear text passwords, certificates and others.

The idea behind this work is to develop an approach to detect RSA private keys and an Amazon access tokens which are both commonly used by developers in an application container image. Related work is available for discovering secrets in source code in general but not in container images directly. This thesis develops and follows a theoretical approach with prototypical implementation to prove that it is possible to apply known key-leak techniques to detect embedded secrets in docker images.

ZUSAMMENFASSUNG

Heutzutage wird cloud native computing immer beliebter. Insbesondere die Anwendungscontainer von Docker sind ein wesentlicher Bestandteil des nativen cloud computing. Jeder Container basiert auf statischen Datenstrukturen - sogenannten Abbildern. Diese Abbilder formen ein wichtiges, aber auch kritisches Element in diesem Ökosystem. Abbilder werden in cloud Umgebungen oder temporär in anderen öffentlichen oder privaten Container-Plattformen gespeichert. Dies führt zu Problemen sobald der Entwickler symmetrische oder asymmetrische Geheimnisse wie SSH-Schlüssel, Klartext-Passwörter, Zertifikate und andere in das Abbild integriert hat.

Die Idee hinter dieser Arbeit ist die Entwicklung eines Ansatzes zur Erkennung privater RSA-Schlüssel und Amazon-Zugriffstokens. Beide Token sind allgegenwärtig und werden von Entwicklern in Container images verwendet. Verwandte Arbeiten existieren, um Geheimnisse im Quellcode im Allgemeinen, aber nicht direkt in Container-Bildern zu entdecken. Diese Arbeit entwickelt und verfolgt einen theoretischen Ansatz mit prototypischer Umsetzung um zu beweisen, dass eine Anwendung bekannter Key-Leak-Techniken zur Erkennung eingebetteter Geheimnisse in Docker Images möglich ist.

CONTENTS

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

Part I

THESIS

1 INTRODUCTION

Nowadays the use of application containers has become indispensable. For developers and system engineers the advantages of those application containers are obvious. Container technology provides isolation and portability for any built application. The applications generally involve business software instead of consumer software. This includes web-applications, databases and further large systems.

In enterprise environments several containers often form one application. Therefore the given isolation has to be partially removed in order to enable intercommunication between these containers. Credentials have to be integrated to guarantee a certain degree of security. Credentials such as passwords or other tokens can establish a more or less secure connection to a specific remote endpoint. This finally results in a runnable application stack with secure intercommunication.

This presupposes developers to adhere to security standards when creating a container-stack. A container is usually an instance of an image. In other words the image builds the base of every container. A fully developed image is usually found on online platforms, known as container-registries. These registries are normally available for the public. Anyone who has access to the image can interact with the integrated file system directly. Integrated secrets are available for attackers, if a developer has incorporated secrets statically into the image.

These secrets can be tapped and exploited. After all the embedded secrets are obsolete. That is a fatal problem in container images in general.

The trust of images builds an additional problem [1]. The image integrity is generally not guaranteed and can be leveraged by the popular man in the middle attack(MITM). There is a signing process necessary with e.g. Docker content trust or a non-central blockchain approach [10]. This leads to a guaranteed integrity assurance.

Outdated software packages can also be a problem for container images. Vulnerabilities of outdated packages are common and public available on Common Vulnerabilities and Exposures(CVE) systems [8]. In theory this problem can be solved by hard work through a consequent patching mechanism.

However this paper focuses only on the problem of embedded secrets which are totally necessary for containers to communicate in a secure manner.

Nowadays Docker is a famous container technology. Docker allows deployments in cloud native environments and on bare metal machines. Possible supported operating systems are for example Windows, MacOS and Linux. The latest IT news reports that Docker is currently undergoing a

radical change. There might be alternative solutions in the future [11]. This work provides examples with Docker, as containers are still promising. Since Linux fundamentals form the basis, this work can be adapted to other container technologies.

Currently the computer science has developed some approaches to discover secrets in general [6]. Related work to detect secrets in container images is not given yet. On this basis the thesis starts its work.

1.1 MOTIVATION

Secrets like passwords and other authentication tokens are used by almost every software that needs a secure communication in an unfaithful environment. The main goal of secrets are to tackle challenges like confidentiality, integrity, authenticity and accountability. A key loss would lead to a collapse of the protection objectives in general. A key-leak is inevitable as soon as a key is integrated into an image. The attacker only needs to launch the container from the image to gain access to the file. Afterwards a scan of the file system for tokens can be made easily.

A developer is probably trained in security standards and knows that tokens should not be integrated statically. But also developers are error-prone people and there is no technical prevention for the static integration of secrets. The development of a concept for the detection of embedded secrets in container images allows a better control of the secrets used and shows possible key integration of the developer. This concept does not work preventively but can be used as a further control instance.

1.2 GOAL OF THIS WORK

This work develops a concept to detect embedded secrets in container images. This concept includes the access method to the container image and an analysis workflow for the detection of defined secrets. Two important types of secrets are used to define a certain scope. A further goal is the prototypical implementation of the concept. The prototype is evaluated by applying realistic scenarios. Finally clear results should emerge from the prototype. The prototype is available on GitHub and on the attached CD.

1.3 STRUCTURE

It is worth to have a structure of this paper in mind before falling into details. This helps to understand the following work in the best possible way. This thesis is basically aimed at everyone who is interested in computer science and has basic knowledge of Linux architecture and container virtualization.

The background chapter starts with container technology introduction in general. That includes a description to containers in general with a brief comparison to the classical virtualization. Furthermore core concepts of containerization is explained in detail. Then the topic UnionFS have to be intro-

duced because this knowledge is necessary for understanding the following container images.

The third chapter gives an introduction to key-leak techniques. Scientifically elaborated documents are reviewed to get an overview of possible key-leak techniques. Which existing methods can be used and adapted for the theoretical concept is considered .

The fourth chapter contains the theoretical concept to detect embedded secrets in images. The concept starts with setting a scope of secrets to detect. Then the image access is defined by a proper comparison. Finally the analysing workflow is defined based on the defined image access method. The analysing workflow represents the core work that represents the scientific delta.

The fifth chapter introduces the practical realization of the theoretical concept. The prototype is derived from the theoretical concept and built accordingly.

Lastly the prototype is evaluated with the help of valid use cases. Self-made images and public images are applied to the prototype. The results are discussed briefly and concisely in the same chapter.

The thesis is completed with a conclusion and future-work chapter. The next chapter will introduce as mentioned with containerization insights.

2 BACKGROUND CONTAINER TECHNOLOGY

This chapter provides the necessary knowledge about containers in a structured form. First containerization is classified and described superficially. Then the containerization core concepts can be discussed in more detail. Practical examples are given to demonstrate these concepts. Afterwards basic knowledge of union file systems is provided in a dedicated section, as these are necessary to understand the last section of Docker images.

2.1 CLASSIFICATION AND PLACEMENT

This section classifies full virtualization and containerization concept shortly by demonstrating the architectural difference.

Containerization is widely used in cloud environments. Containers are almost the foundation of every cloud environment. However classical virtualization exists long before container concepts. Both concepts have advantages and disadvantages. That is the reason why they are also combined as hybrid concepts. Figure 2.1 shows the architectural difference between container technology and full virtualization. Full virtualization allows it to run an en-

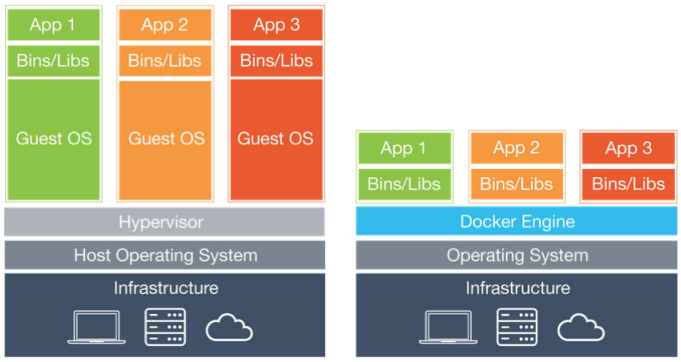

Figure 2.1: Difference between container and full virtualization

tire guest operating system in a virtual machine (VM) on a host operating system. This is possible through an installed piece of software called hypervisor which is build classically on top of the originally operating system. This virtualization model provides solid security through this additional isolation layer. One obvious drawback is the heavyweight and high resource usage characteristic. In contrast to classical virtualization a container does not need an additional operating system layer as seen in Figure 2.1. The container is just using and sharing the underlying kernel from the host.

Containerization technology is closer to the underlying host operating system than the classical virtualization strategy. That makes containers more

lightweight and flexible. Comparisons between container concepts and classical virtualization with regard to the application purposes are described in [] The next section gives a closer look at containerization paradigm itself.

2.2 CONTAINERIZATION PARADIGM

The paradigm of containers is very simple in theory. Containers are basically operated as stateless and separate units. These units can fulfill certain tasks by acting autonomously as isolated software components. The paradigm is integrated by certain kernel functionalities. These functionalities have been started introducing with the jails concept in BSD in the early 2000s [7]. In 2008 more functionalities have been introduced in a user-friendly way in the Linux kernel. The framework is called Linux Containers(LXC's) [7]. However the usage of LXC's can become quite complex. The using of LXC has become much easier since Docker was introduced as a wrapper. Docker offers a user-friendly interaction with the LXC framework and has established itself as a state-of-the-art construct in terms of containerization. Developers are nowadays able to provide containers due to Docker at a much faster pace than with pure LXC's.

Native Linux features are responsible for the encapsulation between host and deployed containers. Since the introduction of container technology under BSD, native functions form the basic framework of containers. These necessary isolation and permissions concepts are described in the following.

2.3 LINUX NAMESPACES

A Linux namespace encloses a global system resource in an abstraction that makes the processes within the namespace appear to have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but invisible to other processes. Namespaces are the basic building block of containers under Linux. The following namespaces exist under Linux:

- Cgroup
- Interprocess communication (IPC)
- Network
- Mount
- PID
- User
- UTS

Unfortunately it is utopian to describe every namespace type in depth in this work. The IPC and mount namespace are described now briefly. Then the more interesting namespaces are explained in a dedicated subsection.

Mount namespaces provide an isolation of the list of mount points seen by the processes in each namespace instance. Processes in each of the mount namespace instances see different single views of directory hierarchies. This view can range from physical or network drives, mount paths or advanced features such as union file systems which are discussed in section 2.5.

IPC namespaces isolate certain IPC resources like System V IPC objects and POSIX message queues which both are data structures they allow via e.g. shared memory to transfer information between processes.

2.3.1 *PID*

Traditionally the Linux kernel has always maintained a single process tree. The tree contains a reference to each process currently running in a parent-child hierarchy. A Linux system starts with process PID1. This is the root of the process tree and the root process initiates the rest of the system by starting different handlers and services. All other processes start below this process PID1 in the tree.

The basic idea behind PID namespaces is to create and append a new root tree to the already existing tree with its own PID1. This makes the child process to a root process. Processes in the subordinate namespace have no way to detect the existence of the parent process. This ensures that processes that belong to a process tree do not inspect or kill processes in other process trees. However processes in the higher-level namespace have a full view in the lower-level namespace of processes.

2.3.2 *Network*

Due to the global instance of the network interface on a single host it is possible with granted permissions to alter routing and ARP tables. With network namespaces totally different instances of network interfaces can be provided. Routing and ARP tables then operate independent of each other. This prevents communication between network namespaces.

Network namespaces are complex but important to know. These are responsible to establish communication between containers and hosts and between containers themselves. The following enumeration shows the standardized CNI (Container Network Interface) workflow. CNI describes how network namespaces are created and how a desired communication between these namespaces can be established.

1. Create network namespace
2. Create bridge network/interface
3. Create virtual-ethernet pairs

4. Attach virtual-ethernet to namespace and bridge
5. Assign IP addresses
6. Bring interfaces up

For a better understanding an example with 2 network namespaces is shown in Figure 2.2. Each color represents a network namespace with its associated virtual network interface pairs(veth-x and veth-x-bridge). For simplicity IP addresses are not shown. In this picture C1 is just a prefix for the underlying hostname which is arbitrary in this case. Basically communication between any networks from a view of a network namespace is not possible as already mentioned. Network communication between namespaces is allowed after the CNI procedure. Finally the two interface endpoints (purple and orange) have a valid IP address which leads to a working network communication. This is the beginning of network communication through namespaces and

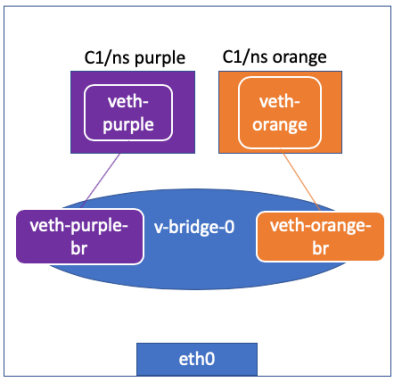

Figure 2.2: Example of basic network within a container cluster

CNI. However only inter communication between different namespaces on a single host is not sufficient. Several steps are required to communicate externally to other host and across the Internet.

Figure 2.3 shows a more comprehensive setup. It shows that the local host is also the gateway because it has one network connection through the interface etho and it has access to the bridge network created on the host. A routing table entry in the blue network namespace like the following necessary, if the blue namespace network needs to access the endpoint with the IP address 192.168.1.3

```
ip netns exec blue ip route add 192.168.1.0/24 via 192.168.15.5
```

This allows the direction from the namespace to the outside endpoint. To enable access from outside to this network namespace it is necessary to create a NAT rule via iptables.

```
iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -j MASQUERADE
```

In order to provide access to the internet from a namespace it is necessary to add a default route as seen below.

```
ip netns exec blue ip rout add default via 192.168.15.5
```

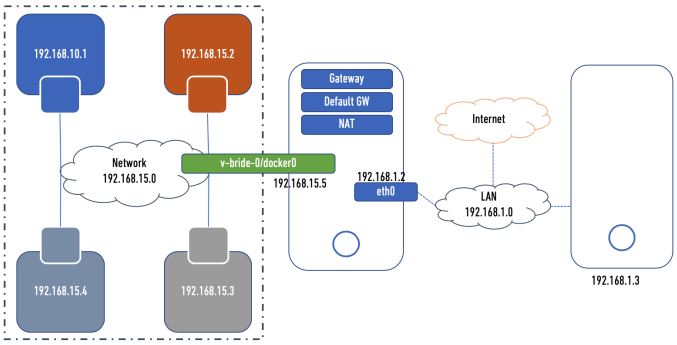

Figure 2.3: Example of comprehensive networking within and outside of container

This example of network namespaces demonstrates the power and flexibility of network namespaces. Normally these steps are automatically done through a container daemon like Docker. This in turn can also be managed by an orchestrator like Kubernetes or Mesos.

Lastly the user namespace is left and described in the following.

2.3.3 *User*

User namespaces isolate user and group IDs so that they appear differently in and outside the user namespace. User namespaces give processes the ability to believe that they are working as root when they are inside the namespace. User namespaces can also operate on capabilities.

What are capabilities? In contrast to privileged processes that bypass all kernel permission checks, unprivileged processes have to pass full permission checking based on the process's credentials such as effective UID, GID and supplementary group list. Linux has privileged process rights divided into different units called capabilities. These distinct units/privileges can be independently assigned and enabled for unprivileged processes.

The next section describes in short another Linux kernel feature called cgroups. They provide another level of security in order to provide a smooth working container environment.

2.4 CGROUPS

Control groups (Cgroups) are a way of enforcing hardware resource limitations and access controls on a process or set of processes. The cgroup scheme provide a hierarchical, inheritable and optionally nested resource control mechanism. In the world of containers Cgroups manifest themselves as instruments to prevent runaway containers and denial of service attacks. The following enumeration contains the resources that are controlled by cgroups.

- CPU usage
- Memory usage

- Disk usages
- Device whitelist
- Network traffic based on tags(class id value) and iptables for filtering
- Application freeze and unfreeze by sending special signals
- PID limitation to limit a specific amount of processes

After this section awareness is raised of the fact that containers work in isolation via native Linux functions. The next important section gives an architectural introduction to the base of every container - a container image.

2.5 UNIONFS

This paper uses Docker as container technology as already mentioned in chapter 1 . To have knowledge of union file systems is necessary before delving into container and especially Docker image insights. Union file systems build the basis for container images in general. This section explains one important file system - the Overlay2 file system.

A union file system represents a file system by grouping directories and files. There are several union file system available like BTRF, AUFS, ZFS and Overlay2 which are compared in detail in [9]. Only Overlay2 will be considered in this work because Overlay2 is directly implemented in the Linux kernel [9] and is meanwhile the standard in Docker related to Docker Inc. [4]. Basically Overlay2 needs at least 4 directory types to work correctly:

- Lower directory - usually read-only and can be an overlay itself
- Upper directory - is normally writable
- Merged directory - represents the union view of the lower and upper directory
- Work directory - used to prepare files before they are switched to the overlay/merged destination in an atomic action

The elements of an Overlay2 file system are now described with the help of an example. First the following folder structure in Listing 2.1 is assumed.

```
somedir/
|-- lower1
|-- lower2
|-- merged
|-- upper
‘-- work
```

Listing 2.1: Tree orig

The folder structure contains all the mandatory Overlay2 elements. The mount point can be created without additional software packages because Overlay2 is natively supported under Linux. This is illustrated by the following mount command.

```
mount -t overlay -o lowerdir=./lower1:./lower2,upperdir=./upper,
    workdir=./work overlay ./merged
```

First the command must know what type of file system to mount. This information is provided by the -t switch. In this case it is an overlay type. The next flag -o allows to add options to mount the specific filesystem. Each option with associated values is separated by a comma. The option lowerdir is set to a chain of folders separated by a colon. The lowerdir argument takes only the lower1 and lower2 directory. Then upperdir is set to the upper folder of the provided hierarchy. The worker option represents a single folder and is set accordingly to the worker folder. Lastly overlay option needs an argument to provide the union mount on the file system. This union view is presented through the merged directory.

The behavior of the Overlay2 file system is demonstrated in the following. The file system as a whole is populated with files as seen in Figure 2.2.

```
somedir/
|-- lower1
|   ‘-- lower1_file
|-- lower2
|   ‘-- lower2_file
|-- merged
|   |-- lower1_file
|   |-- lower2_file
|   ‘-- upper_file
|-- upper
|   ‘-- upper_file
‘-- work
    ‘-- work
```

Listing 2.2: Tree filled

A file creation in one of the lower- and upper directories is visible as expected in the merged directory. A file or directory object in the upper directory tree appears in the overlay. The same object is not visible in the lower directory. Each directory content is merged to create a combined directory object in the overlay. A file or directory that originates from the overlay is removed from the overlay when it has been removed from the upper directory. A file or directory that originates from the lower directory remains in the lower directory when it was removed from the overlay-directory, but a 'whiteout' mark is created in the upper directory. A whiteout takes the form of a character device with device number 0/0 and a name identical to the removed object. A whiteout ensures that the object in the lower directory is simply ignored. Also a whiteout is not not visible in the overlay directory.

Another important fact about union file system is the use of copy on write strategy. A storage driver manager takes care of copying files to the upper

layer when a file from an underlying layer has been modified. A duplicate is created and the modification takes place. This is an efficient resource management technique because operations may just need a copy instead of creating a new file.

The general knowledge about the Overlay2 file system is provided. This is important to understand the main component - the Docker image. This is described in the following.

2.6 DOCKER IMAGE

This section explains the Docker image in more detail. First the architecture of a Docker image with the most necessary information for subsequent work is presented. Then the construct for building a Docker image is explained. Lastly a description of the metadata information of an image takes place. These basics are very important because they are part of the theoretical concept.

2.6.1 *Image architecture*

A Docker image is ultimately a stack of selected file system layers to provide a starting point for a container. Figure 2.4 shows how a Docker image is stacked. At the bottom is always the Linux kernel. A Debian and a Busybox layer are placed on the kernel. Both form now already a complete and runnable Docker image. On top of these layers can be more layers stacked as shown on the Debian layer with an additional Emacs and an Apache layer. The Busybox layer does not have another further layer placed on the top. Docker finally attaches a read/write file system across all underlying layers when a container is launched from an image.

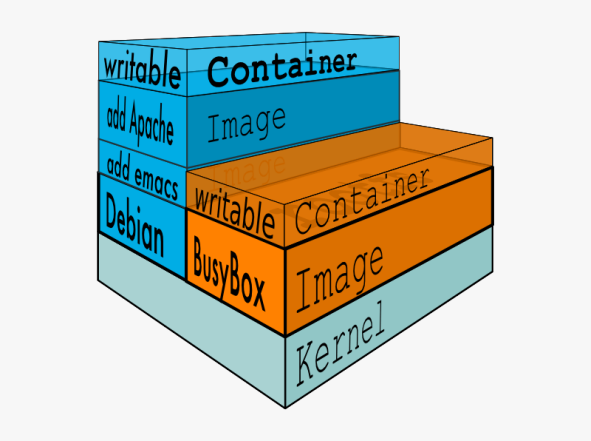

Figure 2.4: Docker filesystems stacked to represent an image

Docker uses storage drivers in order to manage images and corresponding file system layers. A storage driver handles mainly the details about the way these layers interact with each other. There are several storage drivers

available like ZFS, BTRFS and many more which can be configured by the responsible system-engineer or developer. These storage drivers have advantages and disadvantages which should be carefully considered. Docker uses in the latest version Overlay2 as storage driver per default. The Overlay2 informations about an image can be viewed by the Docker inspect command.

```
docker inspect ubuntu
```

The inspection only shows the part that is interesting for the docker image architecture.

```
"GraphDriver": {
    "Data": {
        "LowerDir": "/var/lib/docker/overlay2/500c09474020bb3abd
            19255dcc7664589e47f6aeda1e73f157976c036eda7481/diff
            :/var/lib/docker/overlay2/d59720c859dcab34d196b7bb6c
            7ee7546db87eeb95b2795365db5b103257cb89/diff:/var/lib
            /docker/overlay2/97f106f45bbc27ecd4439cb3cede3f725e0
            c0fb0f642463d4bc656aec76d5b28/diff",
        "MergedDir": "/var/lib/docker/overlay2/9c055fc060f8f992
            dcfa46d8653f6f2c51a8fbde44b788207fb47addc3de2fd3/
            merged",
        "UpperDir": "/var/lib/docker/overlay2/9c055fc060f8f992
            dcfa46d8653f6f2c51a8fbde44b788207fb47addc3de2fd3/
            diff",
        "WorkDir": "/var/lib/docker/overlay2/9c055fc060f8f992
            dcfa46d8653f6f2c51a8fbde44b788207fb47addc3de2fd3/
            work"
    },
    "Name": "overlay2"
},
```

Listing 2.3: Docker inspection results

As seen in Listing 2.3 the introduced Overlay2 elements and mechanisms are completely used. This shows once again that Docker only uses the well-known Linux core functions instead of building own mechanisms.

One interesting and important fact exists about the mount process. The Overlay2 storage driver has a symbolic link for each image layer implemented. During the mount process these symbolic links are used instead of the real folder names. The reason is the total length of 65 characters for each folder name. These symbolic links help avoid the Linux 'mount' command from exceeding page size limitation. It is also important to note that the **diff** directory in each layer builds the chain of the overlay. In each layer are additional helper files available like the lowerfile which creates a relation to associated parent. This lowerfile exists only if a parent folder exist. The lower files are responsible for creating the order from the most upper layer to the lower layers.

Every Docker image has a name and a tag. An image name is made up of slash-separated name components, optionally prefixed by a registry hostname. The hostname must comply with standard DNS rules, but may not

contain underscores. A tag name must be valid ASCII and may contain lowercase and uppercase letters, digits, underscores, periods and dashes. A tag name may not start with a period or a dash and may contain a maximum of 128 characters. It is important to know that the name and the tag are separated by a colon. This knowledge about the naming and tagging connection is helpful, since it finds application in the following chapters. In the following the construct of building a Docker image described.

2.6.2 *Dockerfile*

Docker Inc. introduced a construct called Dockerfile in order to build a Docker image. Each entry in this Dockerfile starts with a keyword. These keywords can be used by a developer to assemble an image. Each entry in a Dockerfile creates a different file system layer. In other words each file system layer represents an instruction with help of a keyword in a Dockerfile. The Dockerfile construct provides around 20 keywords [5]. Listing 2.4 shows a typical Dockerfile.

```
FROM ubuntu:18.04
COPY app.sh /opt/
RUN chmod +x /opt/app.sh
CMD sh /opt/app.sh
```

Listing 2.4: Dockerfile

The FROM statement starts out by creating a layer from the ubuntu 18.04 image. COPY adds an example bash script from the Docker client's current directory. RUN makes the program executable. Finally CMD specifies which command has to be executed inside the container.

An image can be created with the corresponding Dockerfile and the command Docker build. The responsible Docker command is shown below.

```
docker build <my_new_image> -f <dockerfile>
```

The Dockerfile construct is valuable to know because it is responsible for integrating secrets into an image. Logically there are two categories that are responsible for integrating static files. First *direct integration* which is related the actions COPY and ADD. The difference between the two commands is the range of functions. ADD is instead of COPY able to unzip an archive directly to the endpoint. ADD can also request files, folders and archives from an URL and save the content directly to the endpoint.

The second category *indirect integration* is a bit more comprehensive. This category is formed solely by the action RUN. Docker itself uses RUN to trigger a shell command and commit it to a new image layer. The executed shell commands for RUN are inline defined. That allows cases, loop-constructs and external program execution. A flexible bunch of code is allowed since it is just standard bash. This allows a developer to use available tools like ssh-keygen, openssl and manual file and folder creation. It is totally up to the developer what do to with that inline command.

At this time it is known what a Docker image is and how it can be built. It is also known that static files can be integrated into an image in direct and indirect ways. In the following the metadata of an image are introduced.

2.6.3 *Image metadata*

The metadata is another important part of an image. Every Docker image saves informations of the image build process on a system. These informations are locally stored and accessible for the root user. In this work it is assumed that no manipulation of the metadata is performed locally.

The metadata is programmatically accessible through the Docker history command. The output of an Ubuntu 18.04 is shown below in Listing 2.5.

```
[\{'Comment': '', 'Created': 1579137634, 'CreatedBy': '/bin/sh -c #(nop)
      CMD ["/bin/bash"]', 'Id': 'sha256:ccc6e87d482b79dd1645affd
    958479139486e47191dfe7a997c862d89cd8b4c0', 'Size': 0, 'Tags': ['
    ubuntu:latest']\}, \{'Comment': '', 'Created': 1579137634, '
    CreatedBy': "/bin/sh -c mkdir -p /run/systemd && echo 'docker' > /
    run/systemd/container", 'Id': '<missing>', 'Size': 7, 'Tags': None
    \}, \{'Comment': '', 'Created': 1579137633, 'CreatedBy': '/bin/sh -c
     set -xe \\t\\t&& echo \\'#!/bin/sh\\' > /usr/sbin/policy-rc.d \\t&&
     echo \\'exit 101\\' >> /usr/sbin/policy-rc.d \\t&& chmod +x /usr/
    sbin/policy-rc.d \\t\\t&& dpkg-divert --local --rename --add /sbin/
    initctl \\t&& cp -a /usr/sbin/policy-rc.d /sbin/initctl \\t&& sed -i
     \\'s/^exit.*/exit 0/\\' /sbin/initctl \\t\\t&& echo \\'force-unsafe
    -io\\' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \\t\\t&& echo \\'
    DPkg::Post-Invoke \{ "rm -f /var/cache/apt/archives/*.deb /var/cache
    /apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; \};\\' >
    /etc/apt/apt.conf.d/docker-clean \\t&& echo \\'APT::Update::Post-
    Invoke \{ "rm -f /var/cache/apt/archives/*.deb /var/cache/apt/
    archives/partial/*.deb /var/cache/apt/*.bin || true"; \};\\' >> /etc
    /apt/apt.conf.d/docker-clean \\t&& echo \\'Dir::Cache::pkgcache "";
    Dir::Cache::srcpkgcache "";\\' >> /etc/apt/apt.conf.d/docker-clean
    \\t\\t&& echo \\'Acquire::Languages "none";\\' > /etc/apt/apt.conf.d
    /docker-no-languages \\t\\t&& echo \\'Acquire::GzipIndexes "true";
    Acquire::CompressionTypes::Order:: "gz";\\' > /etc/apt/apt.conf.d/
    docker-gzip-indexes \\t\\t&& echo \\'Apt::AutoRemove::
    SuggestsImportant "false";\\' > /etc/apt/apt.conf.d/docker-
    autoremove-suggests', 'Id': '<missing>', 'Size': 745, 'Tags': None
    \}, \{'Comment': '', 'Created': 1579137632, 'CreatedBy': '/bin/sh -c
     [ -z "$(apt-get indextargets)" ]', 'Id': '<missing>', 'Size':
    987485, 'Tags': None\}, \{'Comment': '', 'Created': 1579137631, '
    CreatedBy': '/bin/sh -c #(nop) ADD file:08e718ed0796013f5957a1be7da3
    bef6225f3d82d8be0a86a7114e5caad50cbc in / ', 'Id': '<missing>', '
    Size': 63206511, 'Tags': None\}]
```

Listing 2.5: Prototype structure in Python

The metadata of Docker images provides a lot of information about a Docker image even if the Dockerfile is not available. On one hand this Listing 2.5 might be confusing. On the other hand it is helpful to get an overview how

the meta data is structured. Through the help of the history command the data is representated similar to JSON. The only difference is that numeral values are not marked in quotations.

Many attributes with their corresponding values can be find in the above example. Attributes like "Created", "CreatyBy" are first followed by "Id" and many more. Dockerfile keywords and corresponding parameters can be extracted as well. Most of all Dockerfile instructions like COPY, ADD an RUN are useful since they are responsible for integrating static files into the image. In this example only an ADD command as Dockerfile keyword is used. A special fact applies to the RUN command. RUN commands are not directly listed in the meta informations. Instead only the executable command that follows a RUN command is listed. A COPY instruction is not used in this example.

This chapter provided a basic knowledge about the architecture and the interaction between Docker images and Overlay2. This basic knowledge of these elements and mechanisms under the hood is of crucial importance. This information provides important concepts that are necessary for the development of a theoretical concept. Before a theoretical concept is developed related work is shown about key leak techniques. Important leakage techniques are absolutely essential to detect secrets.

3

KNOWN KEY LEAK TECHNIQUES

Scientific work about key leak techniques on a source code base has been done in [6]. The main work consists in the application of different mechanisms to uncover different types of secrets. These are keyword search, pattern-based search and heuristics-driven filtering. This work mainly dealed with the detection of API tokens from Amazon and Facebook.

The first approach called keyword search focuses on fixed strings in files such as a RSA private key. RSA private keys contains always a prefix like the following:

```
-----BEGIN OPENSSH PRIVATE KEY-----

-----BEGIN RSA PRIVATE KEY-----

-----BEGIN PRIVATE KEY-----
```

This keyword search approach works well if the secrets contain fixed string prefixes. On the other hand, it is not sufficient for private keys without fixed prefixes. Due to this fact, the referenced paper [6] is more focusing on pattern-based search.

Pattern-based search works on the basis of regular expressions and is therefore suitable for strings with a fixed schema. Tokens provided by cloud providers and other dedicated vendors follow a fixed scheme. Therefore these tokens are recognized by regular language classes. However this pattern-based search method also has disadvantages in the form of false-positives. These false-positive issue is handled in [6] by heuristics and source slicing.

Heuristics have been tested by looking at cases where a matching secret key appears within 5 lines of each other. This is usually precise but there is a risk of missing the key where credentials are not close together. Another tested heuristically approach to reduce false-positives is trying and guessing whether they are auto-generated or hand-written. An additional framework was used to follow this approach. The false-positive rate was successful decreased as noticed in [6].

The following concept will use the keyword search and pattern-based search to detect secrets. The reason is the following: It is only intended to prove that the recognition of secrets in container images is generally possible. In theory an extension of heuristics-driven filtering improves the results if the detection of secrets in container images is possible.

The following chapter only adapts these key mechanism keyword search and pattern-based search in the theoretical concept to detect secrets in container images with docker as example.

4

A THEORETICAL CONCEPT TO ANALYSE A DOCKER IMAGE

In this chapter the theoretical concept of detecting secrets in Docker images is evaluated in developed. Thus it begins with setting a scope of secrets to detect and then introduce which method has to be used to gain file access to images in general. Conclusively the process of image analysis is explained with the help of a suitable visualisation.

4.1 SCOPE OF SECRETS

In order to steer the work in a specific direction, it is necessary to set a scope of secrets which has to be discovered. A brief definition of this topic will be evaluated in this segment.

Software developers and system engineers are the main stakeholders who create container images regularly. Nowadays the technical environments are often cloud services. Many of the cloud service players are Amazon, Microsoft and Google. There are many more service providers available. The following list points out the central common features of these global players.

- Compute Engine - Includes virtual machines or clusters
- Kubernetes Engine - Includes Kubernetes components
- Storage - Includes several types of databases and hardware provisioning
- Diverse - Networking, Monitoring, Artificial Intelligence, BigData, etc ...

This is a large bundle of functions provided by each individual cloud provider that can be used within a container and thus in an image. There are many other APIs accessible, apart from the obvious features. For example Google offers more than 100 API's for developers. Fortunately each cloud provider has Identity and Access Management (IAM) integrated for providing the principle of least privilege. But this is at the discretion of the operator.

In order to perform actions via these API's and services, authentication is required beforehand. Google, Microsoft and Amazon have established several kinds of authentication. The developer has to choose application credentials based on what the application needs and where it runs. The ranges of credentials is big and contains amongst others credentials API keys, OAuth 2.0 client authentication, environment-provided service accounts and other types of tokens which are derived from an associated technical user. Since

access keys are nowadays often used by developers directly in the code, the API token gets a special consideration in this paper. It only plays a secondary role, whether the token comes from Google, Microsoft or Amazon. When it comes to the analysis it is important is to recognize a corresponding architecture or scheme of the token. In this thesis an API token from Amazon is investigated. An adaptation to other tokens is also possible when the schema is determined and adapted. Amazon itself uses a combination of an access key and secret token which is normally directly used in the code.

To be independent of a global player, most of all solutions can also be used by subscribing to these services directly from the associated vendor. As a last resort most of all available solutions can be maintained bare-metal. In every case the authentication depends on possible options of the software itself. Simple authentication via user name and password is still common, as well as authentication via certificates. Certificates themselves are flexible, versatile to use and therefore popular. The asymmetric mechanism behind this is usually RSA. RSA is widely used and still the state of art when confidentiality or authenticity is needed. In a RSA key pair, the private key is finally the sensible part which is responsible for the protection goals. Due to the popularity and important use cases of the RSA private key turns into a second popular candidate in this work.

RSA keys are usually created with e.g. client tools like openssl or ssh-keygen. The folder and filename can be changed with passing correct command line arguments to the programs which makes the place and name of the private key arbitrary. The key file can be placed and named whereever the developer sees the necessity. The program which requires the key needs only a correct configuration to find the keyfile. Only the content of the keys counts and has to be untouched. Tools like openssl and ssh-keygen have in common that generated keys are stored on the filesystem. It might be theoretically possible to extract these keys as a plain string and integrate it in a source code, but it is serious design mistake which needs a great deal of effort. Therefore the scope of this work for private keys is only on file system level with consideration of arbitrary locations and names. It has to be considered that RSA keys normally should contain a passphrase to provide additional security in case someone steals the private key file. The passphrase is just a key being used to encrypt the file that contains the RSA key, using a symmetric cipher (usually DES or 3DES). The used symmetric cipher can be examined by reviewing the header of the private key. To use the private key for public key encryption, the file must first be decrypted with the decryption key. If the private key itself is password protected (additional passphrase), then this passphrase must also be made available to the container so that the private key can be used. Otherwise a password request exists and the container does not work automatically. The passphrase in general can be accessed from different sources like a file, an environment variable or another stream which can be piped into the tool like ssh which uses the private key file. In container/image context the passphrase can be integrated during runtime to the container or as a static file itself into the image.

A runtime integration offers the same hurdle as a dictionary attack when it comes to crack the password since the password is provided by the orchestrator and therefore not directly accessible. This top-down method does not usually lead to efficient success. If instead a static integration into the image is done it would leads to the possibility to compromise the passphrase of the private key part. Passphrases are arbitrarily chosen and does not follow a syntax. Therefore no key based search or pattern-based search is possible to find these passwords. This is another hurdle to find the password but is still possible because it is statically integrated. The scope of this theoretical concept is still to find secrets which use a fix schema. The discussion part will provide more informations about this case of setting a passphrase on a private key.

To summarize these two types of secrets, the API-access token from Amazon is a set of sensitive key pair while the RSA private key alone is very sensible. Due to the nature of inserting API tokens in plain text into source code, this is a different level compared to RSA private keys that exists on a file system. These two types of keys define the target to be detected in the images.

4.2 EVALUATION OF ACCESS METHODS

This segment examines the first building block of the core concept of image analysis. It will evaluate and decide which methods has to be used to get file access to an image in general.

Basically an image is distributed arbitrarily and it cannot be assumed that the Dockerfile can be accessed. Therefore only the pure availability of the image is realistic. The first key question is which is an universal access to an image in order to perform a depth analysis. In conclusion the architecture of Docker images leads to three obvious possibilities that are described in the following subsections. These different approaches have advantages and disadvantages. On one hand the key feature of the access method is a well working interaction with the necessary analysing module. When on the other hand the file access method needs big effort or it modifies an image in order to embed necessary actions to such extend that it can have a bad impact for the whole scanning workflow in general. This has to be considered deciding on which access method has to be finally used.

4.2.1 *Additional image layer*

The access is made through a new additional layer which contains and also adds a program to the obtained image. When the image is being initiated as a container, the program can work on the entire file system. Before the new layer is being added, preprocessing on the original image is mandatory. The first result has to be temporary saved. That includes all the metadata informations which are necessary for the scan. These metadata are explained more deeply in section 2.6.3 and will furthermore be used by the processing

module in section 5.3.2. A change of the base image would lead to a lost of these important informations.

After this processing is done a new additional layer can be added with the analysing program. Finally the program would need an endpoint, where the result is saved. The result has to be saved on a permanent storage due to the nature of containers, because the results are removed after stopping the container.

4.2.2 *Tarball approach*

The idea behind this approach is to pipe a running container into a tarball. The container must be started and remain online until all informations are extracted and stored in an archive. After exporting the container can be deleted immediately because the processing only takes place on the tarball. This archive contains the complete file system including the writable container layer. The archive can be analysed afterwards by a program. This program can save the results locally or deliver it to an endpoint.

4.2.3 *Direct access*

Theoretically, direct access to the image is also possible, since an image is present on the local system before it is started as a container. The background chapter showed, that a Docker image is just a stack of several image layers. The direct access to the image as a whole needs a manual overlay-mount on the host system itself. Necessary informations to mount the overlay correctly is proved by the section 2.6. For the overlay the mandatory lower directories have to be examined. The sequence of the directories in the chain is also important. These can be analysed through the provided informations which are available locally. That includes especially the lowerfile of each Docker layer. These informations has to be used in order to finally get the overlay-mount. Finally the program that performs the analysis can access the mount point and do its analysis.

4.2.4 *Decision of access method*

These mentioned approaches have advantages and disadvantages. In subsection 4.2.1 the access approach has the drawback of an additional layer and requires a bi-directional interaction between host and container, or container and a specific endpoint in order to save the results. It also needs a somewhat of a copy of the image in order to save the meta-information. The modifying of the base image leads to a higher complexity and effort. The second approach from 4.2.2 only has the image as a base and does not modify anything from it. It only needs a start of a container temporarily. Unlike in 4.2.1 the container does not have to run during the analysing process. However the fact of starting a container is still a drawback because it can lead to an initially undefined consumption of resources. With the direct access method

from 4.2.3, the analysis is performed without starting a container and therefore no additional container load exists on the host.

Lastly the third approach has the big advantage to access to the image directly through the filesystem. Since direct access to the image via the file system requires the least effort from a logical perspective, this access method is used in this theoretical concept.

4.3 ANALYSING PROCESS

At this point the secrets to be discovered are defined and the file access method to be used is determined. The next step in this concept is to define an analysing process in order to detect these mentioned secrets. The analysing module contains 4 sub-modules, while each sub-module works independent to provide most of flexibility:

- Image-obtaining
- Meta-extraction
- Image-mount
- File-scan

The structure of this analysing section goes as followed:

First the abstractive flowchart in Figure 4.1 gives basic informations about the analysing process in general. It contains the analysis workflow shown with every module in action. Afterwards each module is described in detail in a separate subsection. Finally, there is a pseudocode 4.1 that shows the analysing workflow in one comprehensive flow to catch the last detail questions.

Figure 4.1 starts with obtaining the target image. This obtaining module needs an input argument, which is the name of the Docker image. The obtaining module manages old Docker images and downloads the target image from a container registry. The preprocessing module extracts then the metadata from the image. If the processor recognises the keywords COPY, ADD or RUN a further analysis of the image is necessary, as these commands are responsible for adding possible secrets statically into the image. No further image investigation will take place, when none of these keywords exists. It is important to note that a RUN keyword does not appear directly in the meta informations. Instead the RUN command is represented by a list of program names that have to be compared with the meta-data. The metadata-informations are explained already in section 2.6.3. If the analysis had been taken place the image would have been mounted by the image-mount module which would have been able to alter it. The necessary mounting informations are already available as mentioned in section 2.6.1. Finally the scan of RSA keys and Amazon access token will take place with the mentioned methods from the known key leak techniques 3, namely keyword-based search and pattern-based search. A result is then returned in a flexible way.

The architecture itself has to be built in a way of reusability and should work in a heterogeneous environment. This is achieved through development with proven properties from the field of distributed systems. This includes working with standardized structures and scalable microservices.

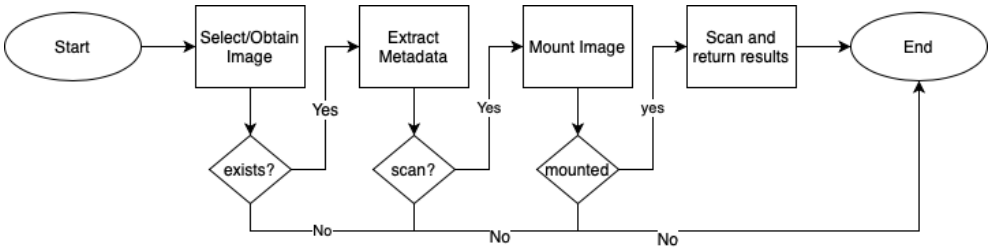

Figure 4.1: Abstract view of analysing process

4.3.1 *Image-obtaining*

The architecture of this module belongs to a classical input output system. This module takes the image name as an argument and will download the image as a file output on the host system. As already known, the output is saved as several folders and files depending on the used union file system which represents the image. Before the download starts the module has to take care about garbage of old images in order to save hardware resources and prevent errors during the future proceeding. This will be done through the Docker API since Docker already provides functions to clean up images from a host system.

The target image can be downloaded from an arbitrary container registry. In this work the module will only pull images from public registries without authentication. In theory it is also possible to pull images from registries where authentication is needed. These actions would also take place in this module. If the image does not exist the analysing process is terminated. However if this module has found and downloaded an image it would continue with the preprocessing of the image. The responsible module is explained in the next subsection of this analysis section.

4.3.2 *Image preprocessing*

This following module is v ery important as it forms the basis for the file scan module. It primarily decides whether the specified image is in need of a scan. The module can also decide which part is in need of a scan for embedded secrets if necessary. The basis for this decision builds the meta-information check. As written in the background chapter the history of an image provides several meta informations like the executed Dockerfile commands. If keywords such as ADD and COPY are found or any RUN methods are used further investigation or analysis is required, as these are responsible for key integration. If none of these have been performed, there is no suspicion of

secrets being integrated and the termination process can be initiated. The reason to check only COPY, ADD and RUN keywords is described in 2.6.3.

The image pre-processing module has to handle the Dockerfile actions ADD/COPY and RUN differently in case of further investigation. This exemplification is already known from the background chapter. In the matter of the ADD and COPY command there is a resulting pattern in the meta information that always seems to be the same. The pattern is the following:

```
ADD file:08e718ed0796013f5957a1be7da3bef6225f3d82d8be0a86a7114e5caad50
    cbc in /
```

It can be seen that an ADD follows a "file",separated by colon and followed by hex code with 65 signs. Therefor it can be seen where the file is copied to. Especially the destination path is appealing to watch. The preprocessing module has to find this destination. This has to be done by introducing a correct pattern like the following:

```
"(dir|file):[a-f0-9]{64}\sin\s"
```

ADD can be treated analogous to COPY in the Dockerfile as well as in the metadata information.

Recognized destination paths finally set the folders that the scan-module has to analyze. These folders are the potential target paths where secrets may exist. Due to the flexibility of these COPY and ADD commands there are a few cases to consider in order to examine the correct targets. It is important to note that many different folders can be examined folders with the same name, and folders with a parent/child relations. Different folders without a file system relation are not a complication and can be therefore treated as a normal scanning target. If duplicate folders are detected the preprocessor must detect and remove this duplicate so that the folder is considered only once.

Subsequently the RUN command implies the simple bash command which already exists in each metadata entry in the history. This is the reason why RUN is not explicitly listed as a keyword in the metadata informations. One approach is to detect actions that follow a RUN action as they are listed in the metadata information. That means there is a data structure with valid keywords necessary which will be compared to the whole metadata. The following enumeration shows commands which must be inserted into a suitable data structure for valid comparison with the metadata.

- ssh-keygen
- openssl
- git clone
- wget

This list is not complete but it provides a first step to extract useful informations from originally RUN commands. All of these programs are ubiquitous. Ssh-keygen and openssl are able to create many types of keys and most

of all RSA key pairs. In this context the private part is the interesting part which can be looked further into. Wget and git clone are common utils to request and download a bunch of files, folder and archives from an endpoint. Especially git clone is ubiquitous, since git is a very common for developers. Furthermore it is important to only recognize these particular programs when they are used and not installed! A developer might install a openssl via a package manager like apt when openssl is not available. This installation doesn't lead to a integration of secrets and has to be omitted. Furthermore a concatenation of tools is possible and has to be considered by this module. The concatenation looks like the following:

```
git clone https://github.com/blackbird71SR/Hello-World && wget
    https://lorempixel/secret.jpg
```

These requirements finally lead to a potentially valid pattern derived from the metadata information.

```
"(/bin/sh\s-c|&&)\s(openssl genrsa|wget|git clone|ssh-keygen)"
```

For each detected program the syntax must be taken into account. Each program specifies an optional target path that must be extracted. This forms the next potential targets which must be considered by the scanning module. When the targets are examined additionally the special interrelation between folders like duplicated folders have also to be considered as mentioned previously.

When no optional endpoint is specified for a program, the current WORKDIR is used. This belongs to the COPY or ADD and to the RUN command. The WORKDIR variable is set to the root(/) folder on the target system if no other specification has been made by the developer. It must also be ensured when using the keyword WORKDIR that the correct relative base path is considered. The following snippet shows the effect of using the WORKDIR variable:

The WORKDIR instruction sets the working directory for any RUN, COPY and ADD command in the Dockerfile

```
ADD test relativeDir/          # adds "test" to ‘WORKDIR‘/
    relativeDir/
ADD test /absoluteDir/         # adds "test" to /absoluteDir/
```

It can be seen that The WORKDIR variable is simply used as prefix when a relative Unix path is used. If the target path of an ADD, COPY and RUN command is recognized, the WORKDIR variable must finally be taken into account unless an absolute path was used. However it is a special use case when the WORKDIR variable is set as root directory or the developer made an instruction to copy files to the root folder. In this case everything should be scanned except for the default root files and folders. This behavior might be confusing and is demonstrated with the following listing: Before the final targets can be defined by the preprocessing module, the root files and folders of a plain Linux system has to be set. This can be done static and scalable since there are standards of root hierchachies established. Every folder that is not listed from the static root folder deviates from the original file system and sets a certain difference. This difference finally sets the target path(s) to scan.

In general this scanning of certain areas enables a higher analysis speed and reduces the false-positive rate significantly instead of scanning a whole bulk of data. Unfortunately this advantage is also a disadvantage at the same time. When secrets are already in upper images they can't be discovered, because associated meta informations are only available starting from the latest base image. That requires an already trusted base image, as they are marked on container registries as official or trusted. It is a must do to only use trustworthy base images since the key problems are not solely existing. Also other types of security vulnerabilities may be included such as manipulated application packages as an example. It is also important to note that no manipulation on the history can be recognised with this module. This would be an important function extend otherwise the manipulation would allow to bypass this pre-processing module.

However this work belongs to normal development processes without manipulations of a build pipeline and this pre-processing module starts at the point where a developer or system engineer creates its own layer from a trusted base image. When the pre-processing is successfully done the targets to be scanned must be made available. The image-mount module is responsible for that and will be explained in the following.

4.3.3 *Image-mount*

In the current state the file system contains only folders of the layers that belong to the selected image. Due to the nature of images it is not possible to map the determined folders to the layers on file system level. In other words there is no assignment of the identified folders to be scanned to the really existing folders on the file system. This is an essential element since this module mainly takes care of providing the real data. This would have in fact reduced the number of folders in the folder-chain to mount.

Due to this reason the ultimate goal of this module is to provide good access to all folders for the scan module. Since Docker creates an overlay over the folders this module will achieve it as well. This overlay leads to an access point for the scanning module. The important information about the required lower folders, and how they are chained originally, will be extracted through the local provided informations of the image. The basis builds the lowerfile in each overlay layer. This image mount module must examine and compare the files. The lower file with the highest amount of lower-directories is the most upper and the highest parent layer. The lowerfile of the highest level of the image contains the most lower directories in a correct order and has to be used for the chain of the overlay mount-point. This chain is the base to create an overlay mount point as well as the 'writeable' folder is a prerequisite and has to be created accordingly. Finally the merged and work directory are created for a fully working overlay. The according elements of an overlay2 file system are described in section 2.5. If the mount process was successful the scan module can be triggered in order to detect potentially secrets. This module will be explained in the following.

4.3.4 *File-scan*

The file scan module is responsible for analyzing specific files for secrets. It's known that this module must detect RSA private keys and Amazon AWS tokens since it is the goal of this paper.

In case of the AWS tokens the amount of existing programming languages makes it difficult to include all technologies in a prototype. The idea is to rather start with specific files to get exactly comprehensible results. This means that not every type of source-code can be examined immediately and this module must be built scalable to support more environments in future. To be more specific the module will not strive for archives (like jars) but will instead refer to pure source code files. In order to find the Amazon AWS token the module will check the files for the following pattern. According to Amazon an access token is built upon a fixed schema which has to be used by this scanning module.

```
AKIA[0-9A-Z]{16}
```

For the recognition of RSA private keys it is not clear which is the name and the file extension of the token. This is the reason why every file in the given directory hierarchy will be checked in except of defined source code file types. The file scan module will search for this fixed prefix at context level in the respective folder hierarchy.

```
-----BEGIN OPENSSH PRIVATE KEY-----

-----BEGIN RSA PRIVATE KEY-----

-----BEGIN PRIVATE KEY-----
```

In both cases the analysis must be done recursively because there can be another folder within a folder which therefore can be nested other folders again. The analysis is performed programmatically with Linux tools like grep and with file operations to browse through files. When the module determines a hit the indicated result is added to a data structure. At the end of the scan the result is returned to the requested client.

To finish off the analysing process Figure 4.3.5 shows a more detailed sequence of analysis. It is a pure logical sequence without going into technical details and implementations.

4.3.5 *Pseudo code analysing*

The following pseudocode is aligned to the programming language Python because it omits all the distracting brackets and makes it more readable to English speakers/readers. For completeness the pseudo code contains a short description of what the program does for a task. Afterwards the abstract algorithm is started.

```
This analysing program will scan a Docker image for embedded secrets.
If a secret is detected then a data-structure containing the secret will
     be returned; if not, a empty data-structure will be returned.
```

```
Enter an image name
if image exist
        pull image
        check keywords COPY, ADD and RUN
        if keywords used
                paths_list = empty
                for keyword in keywords:
                        switch(keyword)
                                COPY:
                                ADD: paths_list.append(check dst PATH
                                    method1)
                                break;
                                RUN: paths_list.append(check dst PATH
                                    method2)
                                break;
                scan paths
                if secrets found
                        return secrets in data-structure
                else
                        return empty data-structure
        else
                return empty data-structure
else
        go to Enter an image name
```

Listing 4.1: Pseudocode of analysing workflow

This chapter provided the theoretical concept to detect embedded secrets in Docker images. The next chapter concentrates on the practical realization.

5 PRACTICAL REALIZATION

This chapter demonstrates a practical realization of the theoretical concept. This is structured in the following subsections.

SUBSECTION 5.1 system environment

SUBSECTION 5.2 prototype structure

SUBSECTION 5.3 implementation core modules

5.1 INTRODUCTION SYSTEM-ENVIRONMENT

Containerization has been successfully established in Linux environments. However it is also available in other environments like Windows and MacOS. Containerization via Docker in Windows and MacOS is implemented through the use of an emulated Linux underneath. The prototyp is developed for a pure Linux environment due to the recommendation to use Docker with Linux. The well-known and stable system Debian GNU/Linux is used as a derivative. Other Linux major distributions like the SUSE or RedHat family are not considered directly. The reason for the Debian based system is the already gained knowledge about Debian systems in the past. The compatibility to other Linux families can be provided by customising the path definitions in the responsible module in subsection 5.3.3.

The landscape of the system environment including working environment is shown in Figure 5.1. The MacOS working machine provides locally a head-

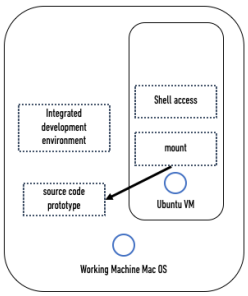

Figure 5.1: System- and working environment

less Ubuntu as a virtual machine. The hypervisor is the well known VirtualBox. The code itself is written on the working machine and mounted on the virtual machine. This allows code accessing and execution on the virtual machine through a ssh tunnel. This corresponds to the idea that the prototyp is operated and tested on a simple Linux system.

Python is used as programming language for this prototyp. The decision of using Python can be made very uncomplicated in this context. It exists a very useful Python library for the Docker engine API. This API allows everything the Docker command does. This includes starting, stopping and altering of containers, images amongst other resources. All available actions can be viewed in the official docs [3].

Python is an interpreted programming language. That allows to run the same code on multiple platforms without recompilation. Hence it is not required to recompile the code after making any alteration. This interpreting mechanism makes programming and testing easier and faster during the development process.

Furthermore Python provides an easy syntax which is readable by any english speaker. For non-Python developers it is like reading pseudo code. This makes it easier to adapt this prototyp in a larger project with a different technology stack.

Lastly Python provides a helpful module called virtual environment (venv). Virtual environments create a dedicated Python environment that allows packages to be installed, modified and used without disturbing the global Python binary. This feature is used to manage the necessary packages like for instance the Docker SDK. This feature is especially useful when the tool is delivered to a remote system. The tool works in existing Python instances independent. In this context it is worth to know that Python in version 2.x has not been supported anymore since January 2020 [2]. Accordingly the newest supported long-term Python version is used. The same applies to the package manager Pip. The package manager should correspond to Python in a compatible version. The exact versions of this prototyp is Python 3.7 and Pip3.

Now the system environment and the programming language to be used is well defined. With this in mind the next section can show the general project structure of the prototyp.

5.2 PROTOTYP STRUCTURE

The following Listing 5.1 gives an overview of the project structure and how the prototyp is implemented in this paper.

```
.
|-- __pycache__
|-- analysing_manager.py
|-- modules
|   |-- __pycache__
|   |-- mount.py
|   |-- obtain.py
|   |-- preprocessing.py
|   `-- scan.py
`-- venv
    |-- bin
    |-- include
```

```
|-- lib
‘-- pyvenv.cfg
```

Listing 5.1: Prototyp structure in Python

The root folder contains the main program analyzing_manager, a modules folder and the venv directory. The venv directory builds the virtual environment feature of Python. The modules folder contains the essential modules of the prototyp, namely obtaining, preprocess, mount and scan. These represent the modules from the theoretical concept in section **??**. They are managed and used by the analyzing_manager module which lies in parent (root) folder. The core modules are deliberately created as distinct units. That makes the maintenance of the prototyp easier. Changing a single core module promises more flexibility and counteracts problems instead of maintaining a large monolith. A replacement of core modules is also possible since it hasn't got any relations to the other modules. Only the interface to the to the analyzing_manager module has to be valid.

The analyzing_manager is the main program and interacts with the core modules. To request the prototype, the analyzing_manager is available as a central endpoint. A single endpoint to interact with provides a good starting point for the analysis of the process. Common technologies like RPC's can be used to trigger this single analyzing endpoint. Also microservice implementation with REST is possible.

The procedure shown below starts automatically when the analyzing manager is triggered with an image name as input.

```
analyzing = analyzingManager(img_name)
analyzing.prepare_environment()
analyzing.preprocess()
if analyzing.necessary:
     analyzing.mount()
     analyzing.examine()
```

After the analyzing object has been instantiated, the method prepare_environment is started. The obtaining module 5.3.1 is addressed and executing its task. The Docker image has to be downloaded and garbage collection has to be done by this module as well. After this step the preprocess method is called and the corresponding preprocessing module is triggered. Through this preprocessing a decision can be made if a further image investigation is necessary. The mount and scan module has to be instantiated If this is necessary. Both the synchronous mount call and the examination call are executed sequentially when an analysis is required. The secret will be printed to the stdout stream when secrets are found. Otherwise a standard info message will be piped into the stdout buffer.

Note: Stdout, also known as standard output, is the default file descriptor where a process can write output.

This section has given an overview about the implementation and the workflow of the prototyp in general. In the following section the implementation of the core modules is presented in more detail.

5.3 IMPLEMENTATION CORE MODULES

This section demonstrates the prototypical realization of the main modules obtain, preprocess, mount, scan. Each of the core modules modules is dedicated to one section. The prototypical realization of each module is shown with an overview of the methods used in combination with a practical implementation meaning code snippets in order to archive the desired goals. The order of the modules to be described is based on the sequence of the analysis process from the theory chapter 4.3. First it starts with the obain module.

5.3.1 *obtain.py*

The obtain module has to take care of the waste of the system environment. This concerns locally stored images that should not be examined. Then the module should download the image to be examined. This module is built very compact and contains only a few methods as seen in the enumeration.

- stop_all_containers(self)
- remove_old_containers(self)
- pull_image(self)

The procedure for the garbage collection consists in general of the methods stop_all_containers and remove_old_containers. Possible running containers needs to be stopped. Then all locally existing images and therefore all overlay directories on the filesystem has to be removed. Both is programmatically done through the equivalent Docker stop and Docker remove command as shown in the code-listing 5.2. These commands are accessible through the available Docker SDK. This dependency has to be installed and integrated into the virtual environment before. The code itself is explanation enough. But it is important to note that the container.reload method is used to get all valid attributes of each container in order to stop it correctly afterwards.

Second the Docker pull command is simply used to download the target image. The module expects a string as an argument to download the image with the given name. The latest tag of the image is used if no tag has been specified. This is achieved by simple string manipulation since it is known that the image name and tag are separated by colon as described in section 2.6.1. The corresponding code snippet to this function can be seen in the code-listing 5.2 as well. It is necessary to specify an image tag for the download. If no tag is specified every image will be downloaded with all available tags. This would lead to a big amount of image layers and conclusively to chaos.

```
#   stop potential running containers
def stop_all_containers(self):
    for container in self.client.containers.list():
        container.reload()
```

```
        container.stop()

#   remove old images
def remove_old_images(self):
    for img in self.client.images.list():
        self.client.images.remove(str(img.id), force=True)

#   pull image
def pull_image(self):
    if ':' in self.img_name:
        self.client.images.pull(self.img_name)
    else:
        self.client.images.pull(self.img_name + ':latest')
```

Listing 5.2: Python snippet - obtaining image

This obtaining module is not very complex. This has advantages because simplicity offers little potential for error in contrast to the following preprocessing module.

5.3.2 *preprocessing.py*

The preprocessing module decides whether an image needs to be scanned. Which areas of the image need to be scanned is also decided by this module. This preprocessing reduces the amount of false positives and increases a much faster pace when it comes to scanning. This corresponding python module needs a bit more logic to work properly. The following function list helps to get an overview of the logic. Only core functions are listed, helper functions are not listed as they are not necessary at this point.

- collect_metadata(self)
- contains_key_actions(self)
- fetch_direct_copy_targets(self)
- fetch_indirect_copy_targets(self)
- clearup_targets(self)
- examine_workdir(self)

The function collect_metadata is the first important method. The function initilizes the Docker environment in order to fetch the locally provided target image. Afterwards the extraction of this fetched image is done through the equivalent Docker history command. The history command provides metadata informations which are explained in section 2.6.3. This metadata is stored in an instance attribute to provide a global access to these metadata.

The decision whether the target image has to be scanned or not is made in the method contains_key_actions. An image has to be scanned when keywords such as ADD, COPY or any RUN commands have been used during

the building process. This was developed in the theoretical concept in section 4.3.2, The determination is made with help of a proper data-structure compared against the metadata information of the image. The Python dictionary is a data-structure to provide one or more key:value pairs. The key:value pair represents the keyword with an associated status whether one of the keys was used. This associated status value is a simple boolean. The key part of the data-structure is derived from the concept. The value of each key is set by default to false. The final data-structure is shown below.

```
action_dict = {
    "ADD": False,
    "COPY": False,
    "openssl": False,
    "wget": False,
    "git clone": False,
    "ssh-keygen": False,
}
```

The comparison implemented in Python can be seen in code-listing 5.3. The corresponding value will be updated to true when an entry from the dictionary is detected. The whole data_structure will be checked and a dedicated boolean is set and returned. The method contains_key_actions returns true if any key values are set to true. That means in general that a further investigation of the image is mandatory. No further investigation is necessary if each value is still set to false.

```
def contains_key_actions(self):
    for key in self.action_dict.keys():
        if key in self.img_meta:
            self.action_dict.update({key: True})

    for value in self.action_dict.values():
        if value is True:
            return True

    return False
```

Listing 5.3: Python snippet - scanning decision

Furthermore the methods fetch_direct_copy_targets and fetch_indirect_copy_targets have in common to extract and to return detected path(s) used by COPY, ADD or RUN. The method fetch_direct_copy_targets takes care about fetching targets which has been integrated via ADD and COPY. The method fetch_indirect_copy_targets takes care about the indirect integration via openssl, wget, git clone and ssh-keygen. The implementation of both methods looks different.

The method fetch_direct_copy_targets searches in the meta information for this following regex pattern which was determined in the theoretical concept 4.3.2.

```
"(dir|file):[a-fo-9]{64}\sin\s"
```

A string slicing takes place in order to extract the target path if this pattern matches in the meta information. Simple string slicing with determination of the position of special delimiter signs is possible. This is due to the fixed schema or the fixed syntax of the meta information. The examination of the target path realised in Python can be seen in the code-listing below. This developed algorithm deserves a tiny explanation.

```
def fetch_direct_copy_targets(self):
    temp_list = list()
    for match in re.finditer("(dir|file):[a-fo-9]{64}\sin\s", img_meta):
        target_path = examine_with_string_slicing()
        temp_meta = slice_orig_meta
        examine_workdir(temp_meta)
        if target_path is '/':
            continue
        if target_path[0] is '/':
            temp_list.append(target_path)
        if target_path[0] is '.':
            temp_list.append(workdir)
        elif target_path[0] is not '/' and target_path[0] is not '.':
            path = trim(path)
            temp_list.append(path)

     return temp_list
```

Listing 5.4: Python pseudo snippet - fetch COPY/ADD targets

Each match from the Regex pattern is processed further. The determined destination folders are appended to a list. The determination starts with examining the target_path of the match. A temporary metadata is extracted in order to set the current WORKDIR for the corresponding target_path when the loop is continuing normally. The WORKDIR variable must be determined exactly since the WORKDIR variable can change several times in the build-process. This is why a string slicing is necessary. The slicing takes places from the start of the original metadata until the position of the already determined corresponding target directory. The last occurring WORKDIR variable of this sliced metadata sets the final WORKDIR variable for the corresponding target_path. The WORKDIR variable is important when the target_path is relativ instead of absolute as known from section **??**.

Furthermore it is important to distinguish between the root path, relativ paths and absolute paths of the examined target_path. This is examined by analyzing the first character of the already known target_path. Loop rounds are omitted when the target_path is the root directory. This is due to the last entry since this is always an ADD command for the base image layer. This has no effect on a copy action from a developer who explicitly chose the root directory as target. The original target_path is added to the target_list when an absolute path is recognized. A string concatenation will take place in order to set the correct target folder if a relative path is recognized. The result is added to the target_list. Finally the full examined target_path (array) is returned after this processing.

As decided the method fetch_indirect_copy_targets searches in the meta information for special programs. These programs are recognized in a regex pattern seen below 4.3.2. This pattern was defined in the theoretical concept as well.

```
"(/bin/sh\s–c|&&)\s(openssl genrsa|wget|git clone|ssh–keygen)"
```

A helper function to this associated command is called when any of these commands have been recognized. The helper function is mainly responsible for the string slicing. The helper method expects the output option of the command and the corresponding determined position of the option in the metadata. The output option may differ depending on the command. Thats is why a helper function comes in place to omit duplicated code. The helper function will determine and return the output path finally. Depending of the amount of matches in the meta-data the helper function may be called multiple times. That is why an examined target from the helper function is appended to an array.

A special case takes place when no output argument is given. In that case the current WORKDIR has to be checked and returned. Absolute and relative paths play a role as with COPY and ADD. The treatment of relatives and absolute paths in fetch_indirect_copy_targets are identically as in fetch_direct_copy_targets. A full in depth insight of the corresponding work between the fetch_indirect_copy_targets and the helper functions can be seen in source code of the attached CD.

Finally this preprocessing module holds the method clearup_targets. This method takes care about the data structure which holds the targets globally. The global data structure is transformed into a Python set. This set automatically removes duplicates.

At this point the target directories are examined and the necessary mount module can be triggered. The realization of this mount module is described in the next subsection.

5.3.3 *mount.py*

The structure of the mount module is built straight forward. Only core functions are listed below:

- create_overlay_dirs(self)
- remove_overlay_dirs(self)
- mount_overlay(self)
- unmount_overlay(self)

The functions create_overlay_dirs and remove_overlay_dirs are responsible for preparing the union mount environment. There are working directories necessary to mount an Overlay2 file system as written in section 2.5. Every folder in except of the lower directories are created or deleted by these functions. The creation and deletion is programmatically done via the available

os package and is due to the simplicity not worth to show at this point. It is important to note twice that the lower directories are not created or deleted by these methods. The lower directories are automatically provided on the file system when the obtaining module starts cleaning up the garbage or downloading the target image. Therefore the lowerdirs must be examined separately. The examination is done by an additional helper method. The examination is shown in the code-listing 5.5. The helper method tries to return a directory chain as a string concatenation of the existing image layers. To do this the directory must be traversed. The Python package *os* helps to walk through the local file system. The helper function iterates over the directories where the image layers are located. At each iteration the folder name is concatenated accordingly to the necessary syntax which can be seen in Listing 2.5 in the background chapter. Finally a valid lowerdir chain as a string is created and returned.

```
def get_lower_directories(self):
    dirs = os.listdir(self.overlay_path + '/l')
    lower_chain = ""
    for dir in dirs:
        lower_chain += f"l/{dir}:"

    lower_chain = lower_chain[:len(lower_chain) - 1]
    return lower_chain
```

Listing 5.5: Python snippet - get lowerdirs

After this examination the information of the lower chain is used as a parameter for the mounting process. The mount command is performed by the method mount_overlay. The following code-listing 5.6 shows the implementation.

```
def mount_overlay(self):
    self.remove_overlay_dirs()
    self.create_overlay_dirs()
    directory_diffs = self.get_lower_directories()
    mount_cmd = f"mount -t overlay -o lowerdir={directory_diffs},
        upperdir=./{self.merged_dir_name},workdir=./{self.work_dir_name}
         {self.overlay_name} {self.merged_dir_name}/"
    os.chdir(self.overlay_path)
    os.system(mount_cmd)
```

Listing 5.6: Python snippet - mount

It can be seen that remove_overlay_dirs and create_overlay_dirs are triggered first. Afterwards a helper function to obtain the lower directories is requested. With that gained information the actual mount process is executed afterwards. The name of that overlay_name variable is important since it is used to unmount the process as the following snippet 5.7 shows.

```
def unmount_overlay(self):
    umount_cmd = f"umount {self.overlay_name}"
```

```
os.system(umount_cmd)
```

Listing 5.7: Python snippet - umount

Finally the overlay is created and a direct access to the image is achieved. This access will be used by the file scan module. The implementation of that module is shown in the next subsection.

5.3.4 *scan.py*

The scan module contains three important functions as seen in the following enumeration.

- scan_for_rsa_pk(self)
- scan_for_aws_key(self)
- get_root_diff(self)

The function scan_for_rsa_pk includes a data_structure to hold prefixes of RSA private keys. These static prefixes are known from the theory chapter and is stored in a Python array seen in the following.

```
prefix_list = [
    "-----BEGIN OPENSSH PRIVATE KEY-----",
    "-----BEGIN RSA PRIVATE KEY-----",
    "-----BEGIN PRIVATE KEY-----"
    ]
```

The idea is to combine each prefix with the already determined target path list. This information can then be combined with standard linux tools to perform a scan for RSA private keys. The following code snippet shows the core method to scan these directories with the linux standard utils.

```
def scan_for_rsa_pk(self):
    mount = Mount()
    for prefix in fix_strings:
        for dir in self.target_list:
            if dir is '/':
                for target_root in self.get_root_diff():
                    os.system(f"find {mount.overlay_mount_path +
                        target_root} -type f -iname '*' -exec grep -
                        HIr -- '{prefix}' "'{}'" \;")
            else:
                os.system(f"find {mount.overlay_mount_path + dir} -
                    type f -iname '*' -exec grep -HIr -- '{prefix}'
                    "'{}'" \;")
```

For preparing the final target dir the prefix builds with the target_list the cartesian product. Two cases has to be considered depending on the resulting *dir* variable. A new target path examination is done by the function get_root_diff when the root directory has to be scanned. This function returns a list of all folders that are different from a standard Linux root file

system. This deviation forms the targets to scan on root level. This leads finally to a third nested for loop in the cartesian product. The full path is created through the overlay mount path and the examined target dir in combination. Finally the find grep utils are responsible to uncover the secrets.

The original destination remains the same if no files/folders are placed at root level. However the scan method is in both cases the same. It does not matter if the scan takes place at root level or in subfolders. The Linux find standard util searches for any kind of file (including hidden files) in the provided target path per iteration. Each discovered file will be piped into the grep command. The grep command finally checks the prefix against the file at context level. Because of the grep options the result is finally printed to the stdout buffer.

The scan_for_aws_key function requires a work pattern instead of a data structure with prefixes. This regex pattern from the theoretical concept is simply applied. The function is slightly leaner than scan_for_rsa_pk. This is because only one pattern is used for instead of several prefixes. This leads conclusively to the elimination of one for loop. This is advantageous for the speed of the process. Also by reducing complexity. The implementation is almost identical to the previous function of RSA detection. The only difference is the parameter usage of the tool grep. An option is enabled so that patterns can be recognized. Because of the small difference, the code is left out at this point and referred to the CD.

The prototype is exemplary fully implemented according to the schema and can be evaluated. This is done in the following chapter.

6 EVALUATION

In this chapter the prototypical implementation of image analysis is evaluated. The input for the prototype consists of self-made images as well as images from public container registers. The self-made images will purposely contain RSA and AWS secrets while it is not clear if the public ones will contain these types of secrets. The goal is to show that the prototype works in general and also for public images. Therefore a number of public images has to be scanned to find secrets. The results of the locally images and public images are briefly and concisely presented. The next section of this chapter starts with the development of self-created Docker images.

6.1 SELF DEVELOPED IMAGES

The creation of Docker images containing secrets to detect needs a directive. An arbitrary development of Dockerfiles and thus Docker images leads to chaos and misleading scan-results. Regardless of how the results turn out, it is desirable to be able to clearly assign the results.

The section structures the creation of image as follows. Basically there are two categories of Docker images. The first contains RSA secrets and the second AWS tokens. Each category requires the creation of two images. In other words there will be 2 Dockerfiles for the RSA and the AWS category. In total there are 4 Docker images which have to be created by developing a corresponding Dockerfile. The reason for the development of 4 screens results from the possibility to use different integration variants. It is known that the direct and the indirect method exist. Therefore 4 images are needed to cover both categories logically completely.

It is important to consider different cases when developing a Dockerfile. WORKDIR changes has to take place since they are commonly used by developers. Finally it is important to use absolute and relative destination paths.

RSA and AWS images are developed in dedicated subsections. The first subsegment develops RSA images.

6.1.1 *Docker image RSA private key*

This subsection develops Docker images with RSA keys integrated in a direct and indirect way. The corresponding images are created and demonstrated by the following two Dockerfiles.

```
FROM ubuntu:18.04
COPY files/id_rsa .
WORKDIR /opt
ADD files/rsa.private .
```

```
WORKDIR /mnt
COPY files/id_rsa ./my_pka
```

Listing 6.1: Image with RSA secret using COPY and ADD

```
FROM ubuntu:18.04
RUN apt update && apt install -y wget openssl git openssh-client
RUN ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
RUN openssl genrsa -out ./rsa.private 1024
WORKDIR /root
RUN openssl genrsa -out ./linse.private 2048
```

Listing 6.2: Image with RSA secret using RUN

Two RSA private keys are integrated directly using COPY and ADD instructions as seen in the first Dockerfile in Listing 6.1. Another three RSA private keys are integrated in an indirect way using RUN commands. The tool ssh-keygen and openssl generate RSA keys as shown in the second Dockerfile in Listing 6.2. Both Dockerfiles contain WORKDIR changes and the use of relative and absolute paths.

The images are created and handed over to the analyzis manager one after the other. The entire analyzis workflow is triggered each time the analyzis manager is started. That includes garbage collection, fetching the image, mounting and lastly scanning the image. The analyzing manager is finally started by passing the image name as a command line argument. That demonstrates the following command.

```
python analysing_manager.py <img_name>
```

After starting this program the status is written to the standard output buffer at important points. A complete analysis with console output of the Dockerfile from Listing 6.2 is shown in Listing 6.3.

```
#####   Preparing environment   #####

#####   Preprocessing Image rsa_second   #####

#####   rsa_second   Has to be analysed #####

#####   Mounting   #####

#####   examination RSA keys starts   #####
/var/lib/docker/overlay2/merged/root/.ssh/id_rsa:-----BEGIN RSA PRIVATE
    KEY-----
/var/lib/docker/overlay2/merged/root/linse.private:-----BEGIN RSA
    PRIVATE KEY-----
/var/lib/docker/overlay2/merged/rsa.private:-----BEGIN RSA PRIVATE KEY
    -----

#####   examination AWS tokens starts   #####
```

Listing 6.3: Result RSA keys analyse

Each step is printed in the console. The scan result is finally visible at the bottom of the listing. Every generated RSA key has been found and printed out with the corresponding location. The analyzis program recognizes the RSA keys completely from the first image shown in Listing 6.1 as well.

Amazon web tokens form the target to be analyzed at next. This is done in the following subsection.

6.1.2 *Docker image AWS token*

This subsection develops Docker images with AWS tokens integrated in a direct and indirect way. The corresponding images are represented by the following two Dockerfiles.

```
FROM ubuntu:18.04
COPY files/aws .
WORKDIR /opt
ADD files/aws /tmp/
WORKDIR /mnt
ADD files/aws .
```

Listing 6.4: Image with AWS token using COPY and ADD

```
FROM ubuntu:18.04
RUN apt update && apt install -y wget openssl git openssh-client
RUN wget https://raw.githubusercontent.com/c-linse/rsa_aws_fake_keys/
    master/aws/java/src/aws_client.java -O /tmp/secrete.txt
RUN git clone https://github.com/c-linse/rsa_aws_fake_keys.git
```

Listing 6.5: Image with AWS token using RUN

The analyzing program is applied to the second Docker image Listing 6.5 because wget and git clones has to be considered as well. Now all 4 example RUN commands are considered by the analysing tool (wget, git clone, openssl and ssh-keygen). The result below in Listing 6.6 shows that every AWS token has been found and printed out with the corresponding location. The analyzis program recognizes the AWS tokens completely from the first image shown in Listing 6.4 as well.

```
#####   aws_second   Has to be analysed #####

#####   Mounting   #####

#####   examination RSA keys starts   #####

#####   examination AWS tokens starts   #####
/var/lib/docker/overlay2/merged/rsa_aws_fake_keys/aws/java/src/
    aws_client.java:    String ACCESS_KEY_ID = "AKIA2E0A8F3B244C9986";
/var/lib/docker/overlay2/merged/tmp/secrete.txt:    String ACCESS_KEY_ID
     = "AKIA2E0A8F3B244C9986";
```

Listing 6.6: Result AWS token analyse

6.1.3 *Intermediate results*

At this point it is important to emphasize that locally self-generated images with RSA and AWS secrets integrated can be recognized by the prototype. Also the speed of the analyzing manager is remarkably fast. Only the download is the bottleneck in this program. The download depends on the size of the image and the bandwidth of the internet connection. An additional timer module was integrated in order to get an overview of the time required by the analyzing manager. This measurement includes every step apart from the downloading phase. This means preprocessing, mounting and finally scanning. The timer is yet another python module and is started before the analysing object is initiated. The timer is stopped when result is printed out and the object is destroyed. The following overview shows the required analysis time of the previously developed images from subsection 6.1.1 and 6.1.2.

IMAGE 6.1 ~0.1449288309995609 seconds

IMAGE 6.2 ~0.10929401899920776 seconds

IMAGE 6.4 ~0.1398549079985969 seconds

IMAGE 6.5 ~0.4584746649998124 seconds

The high performance of the scan is caused by the previously determined destination folders by the preprocessing module. A complete bulk scan is thereby excluded and only specific folders scanned.

The calculated time of the scans does not provide general times since it depends finally of the image size. It would probably make sense to scan several Docker images of different size to get valid relations. The calculated time by this prototype gives a rough hint how big is the impact of using the preprocessing module. In this case the times are impressive compared to a bulkscan that takes several minutes.

It is known that the prototype works for locally self-developed images. But it is worth to know if the prototype can detect secrets in public images as well. This question is examined in the next section of this evaluation chapter.

6.2 PUBLIC AVAILABLE IMAGES

Public images are stored and available in container registries. Container registries are provided by several cloud providers. Examples are the following providers and their solutions.

- Docker Inc. - DockerHub
- Google - Google Container Registry(GCR)
- Amazon - Elastic Container Registry(ECR)

There are many other providers and solutions available. Each cloud provider offers normally a private area to make images only accessible to sepcial users/groups. A valid authentication is necessary to get access to these locked images. However the provided prototype only supports the query of public container registries without authentication.

DockerHub provides a large bundle of images provided by the community and is the standard container registry for public images. This platform is therefore a very good candidate for the designed prototype.

A simple strategy is used to find potentially suspicious images from DockerHub. The search in DockerHub includes mainly backend technologies that expects a communication to remote endpoints. This in turn requires secure communication using mechanisms such as RSA. An additional filter is set to fetch only non official DockerHub images. These images are not proved officially by Docker Inc. Images from non-verified third party vendors theoretically have a higher potential of vulnerabilities than official ones. The search is performed manually without programmatically API queries.

However the scan is autonomous because the prototype only needs the name of the suspicious image. The scan is performed one after the other without any parallel process execution. One of the scans was performed on a frequently used image with more than 10 million downloads. This image was updated in Oktober 2019. The image is called *nodered/node-red-docker* and has the following SHA *sha256:0bd9a1d2200474e7471bada2eb633f7193320ee47cb3b8aa34326d19f7f485c6*. The console output of the scan can be seen in Listing 6.7.

Note: The SHA is calculated over every image-layer by a special algorithm. This defines an image explicitly and makes clear which image was used exactly.

```
#####   Preparing environment   #####

#####   Preprocessing Image nodered/node-red-docker   #####

#####   nodered/node-red-docker   Has to be analysed #####

#####   Mounting   #####

#####   examination RSA keys starts   #####
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/mqtt/test/
    helpers/private-key.pem:-----BEGIN RSA PRIVATE KEY-----
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/mqtt/test/
    helpers/wrong-key.pem:-----BEGIN RSA PRIVATE KEY-----
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/mqtt/test/
    helpers/tls-key.pem:-----BEGIN RSA PRIVATE KEY-----
/var/lib/docker/overlay2/merged//usr/src/node-red/node_modules/mqtt/
    examples/tls client/tls-key.pem:-----BEGIN RSA PRIVATE KEY-----
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/oauth/
    tests/oauthtests.js:var RsaPrivateKey = "-----BEGIN RSA PRIVATE KEY
    -----\n" +
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/http-
    signature/http_signing.md:    -----BEGIN RSA PRIVATE KEY-----
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/agent-base
    /test/ssl-cert-snakeoil.key:-----BEGIN RSA PRIVATE KEY-----
```

```
/var/lib/docker/overlay2/merged/usr/src/node-red/node_modules/https-
    proxy-agent/test/ssl-cert-snakeoil.key:-----BEGIN RSA PRIVATE KEY
    -----

#####   examination AWS tokens starts   #####
```

Listing 6.7: Result RSA keys analyse

The result shows that 8 private keys have been found in the Docker image. For each RSA key found the corresponding folder is displayed. The folder structure indicates that these keys are mostly used by the MQTT protocol. It can be assumed that the keys found are only for test and demo purposes. This can be deduced from the folder structure. Further investigation may help to assign this private key. It cannot be excluded that these are potentially secret keys. This also applies to the other private keys All these private keys needs a further investigation in order to obtain possible sensitive informations.

This section showed that the prototype can also be applied to public images. This confirms a possible universal application if the prototype shall be developed as a product.

7 CONCLUSION

The goal of this bachelor thesis was to develop an approach to detect embedded secrets in Docker images. A theoretical concept was created to achieve this goal. In this concept the decision was made to detect only RSA private keys and Amazon access tokens. These two secrets set the scope of this paper. These two keys have different characteristics. RSA contains a fixed prefix, while an AWS token follows a specific pattern at the content level. Then a decision was made that direct access to the image would be a better approach than the alternative tarball and additional container layer approach. This was followed by developing an analysis procedure consisting of 4 core modules. These core modules are the following

- Image-obtaining
- Meta-extraction
- Image-mount
- File-scan

The Image-obtaining module deletes old Docker images from the local system and downloads the target image from a container registry.

The preprocessing module extracts the metadata from the image. The metadata is a bulk of data of each image and is created by the image build process itself. Most important informations in the metadata are most of all Dockerfile instructions. A further analysis of the image is necessary if the processor recognises the keywords COPY, ADD or RUN. These commands are responsible for adding possible secrets statically into the image. No further image investigation will take place when no keywords like that are existing.

The image will be mounted by the image-mount module if the scan is necessary. The mount type is the union-file-system Overlay2. This module provides a central point from which the host system can access and finally scan all files of the image.

Lastly the file-scan module is responsible for detecting RSA keys and Amazon access tokens with the use of key-based search and pattern-based search. The scanning module uses the path of the overlay mount point and can use it as a classic Unix path.

This concept was prototypically implemented in a Linux environment using the programming language Python. Each core-module of the analyzing procedure was implemented as a different python module to provide independence and most of flexibility. An additional module called analyzing manager was used to manage the whole analyzing workflow by calling and

managing each of the mentioned python modules. So there is a central program to which an image can be passed.

Lastly the developed prototype was evaluated. Self-defined images as well as public images were used to prove the prototyp is working.

The self-created images contained intentionally RSA keys and Amazon tokens. The direct and indirect methods were used to integrate the secrets. These two methods are responsible for the integration of static content. The direct method includes only the use of the COPY/ADD instructions in the Dockerfile. The indirect method includes all commands that can be executed via RUN. The direct method is deterministic and completely covered in the prototype. With the indirect method, initially only a certain degree of coverage is possible. This is due to the variety of potential tools that can be used for static integration. It was started in the prototype with 4 tools. Attention was paid to possible scalability. Then the self-made images were handed over to the prototype to perform the first tests.

Furthermore public images were also analysed by the prototyp. Public images are inevitable to achieve valid and more trustworthy results. A simple strategy was used to manually search for images on Dockerhub. Non-official Docker images that use backend technologies were searched.

The tests of local and public images showed positive results. For the self-developed images, the direct as well as the indirect method was covered in the desired context. Each secret was uncovered in every self-defined Docker image within a very short time. The detection of secrets in public images has also worked. Several RSA keys in a common Docker image were found. It is not clearly definable whether these keys are used for production or test systems. However the result is that the prototype has found the desired keys.

8

FUTURE WORK

First, the prototype is currently build only for Docker images. The images are based on standards of the OCI (Open Container Initiative). A possible improvement is to replace the dependent Docker SDK's with a more universal approach. An approach with Linux standard utils would result in even more flexibility. There would be no need to install any dependent docker tools on the scan host

Another enhancement is the functional extension of the scalable scan module. The category of indirect copies consists of much more than the 4 utilities that are currently provided by this prototype. Although it is possible to handle the tools openssl, ssh-keygen wget and git in a reasonable way, there are enough other tools to consider. A productive use of such an analysis tool is pointless without considering many other tools. The following list gives an idea which tools should be considered in future.

- scp
- curl
- ftp
- sftp
- rsync
- gcp
- ...

This is just a new bundle of tools to extend and by far not the end. There might be many more tools available which can be responsible to integrate files statically. As with antivirus programs, it can lead to similar effects of signature maintenance. If tools are used by the developer that are not included in the program, this leads to false-negatives. The integration of new tools does not need big effort, since a helper function is already existing. Only the output parameter has to bet set for the corresponding tool. But it is necessary to pay attention to updates of the integrated tools. Theoretically parameters of the tools can change. These must then be adapted in the software.

Not only the extension of more tools is useful, also the extension of more secret types. Currently the scan module detects only RSA keys and Amazon access tokens. Depending on the use cases, other types of token from other cloud service providers may be worth to scan. Since the analysis engine has a modular structure, further secret types can easily be extended by adding functions to the scan module.

Furthermore an enhancement of the homogeneity can be reached. The first valid step would be to reach other Linux families like RedHat and SUSE systems. This can be adapted easily by only setting correct parameters which are related to the Linux file system. As an example the docker image layers may vary depending on the used Linux family. An early identification of the system could set the path which has to be used. This can be done in one of the preprocessing modules. As a result the software can be used by the whole Linux family. MacOS and Windows systems are a bit more difficult to support since they are using an emulated Linux underneath. The prototype can be surely applied to the underlying Linux machine. The interface to this Linux system can be a problem. However it is generally not recommended to use containerization on Windows and Mac. Therefore this should only be an optional feature in the future and not a necessary one.

BIBLIOGRAPHY

[1] Theo Combe, Antony Martin, and Roberto Pietro. "To Docker or Not to Docker: A Security Perspective." In: *IEEE Cloud Computing* 3 (Sept. 2016), pp. 54–62. DOI: 10.1109/MCC.2016.100.

[2] Python Software Foundation. *PEP 373 – Python 2.7 Release Schedule*.

[3] Docker Inc. *Docker SDK for Python*. Docker Inc.

[4] Docker Inc. *Docker storage drivers*.

[5] Docker Inc. *Dockerfile reference*.

[6] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. "Detecting and Mitigating Secret-Key Leaks in Source Code Repositories." In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 396–400. DOI: 10.1109/MSR.2015.48.

[7] Murugiah Souppaya, John Morello, and Karen Scarfone. *NIST Special Publication 800-190, Application Container Security Guide*. Sept. 2017. DOI: 10.6028/NIST.SP.800-190.

[8] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. "Security Analysis of Container Images Using Cloud Analytics Framework." In: *Web Services – ICWS 2018*. Ed. by Hai Jin, Qingyang Wang, and Liang-Jie Zhang. Cham: Springer International Publishing, 2018, pp. 116–133. ISBN: 978-3-319-94289-6.

[9] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. "Evaluating Docker storage performance: from workloads to graph drivers." In: *Cluster Computing* 22.4 (2019), pp. 1159–1172. ISSN: 1573-7543. DOI: 10.1007/s10586-018-02893-y. URL: https://doi.org/10.1007/s10586-018-02893-y.

[10] Quanqing Xu, Chao Jin, Mohamed Faruq Bin Mohamed Rasid, Bharadwaj Veeravalli, and Khin Mi Mi Aung. "Blockchain-based decentralized content trust for docker images." In: *Multimedia Tools and Applications* 77.14 (2018), pp. 18223–18248. ISSN: 1573-7721. DOI: 10.1007/s11042-017-5224-6. URL: https://doi.org/10.1007/s11042-017-5224-6.

[11] Heise online. *Container: Docker verkauft Enterprise-Geschäft und bekommt neuen CEO*.