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. Especially Docker's application containers are an essential part in cloud native computing and a critical member in this eco-system. Docker images are distributed arbitrarily in cloud environments or temporarily in other public or private container registries. This leads to problems if the developer has integrated some symmetric or asymmetric embedded secrets such as SSH keys, plaintext passwords, certificates, and many more.

The idea behind this work is to develop an approach to detect a secret RSA private key and an Amazon access token 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 shows an approach using Docker as an example to prove that an application of known key-leak techniques to detect embedded secrets is possible.

ZUSAMMENFASSUNG

Heutzutage wird Cloud Native Computing immer beliebter. Insbesondere Docker's Applikations Container sind im Cloud native Computing ein wesentlicher Bestandteil und ein kritisches Mitglied des Ökosystems. Docker-Images werden manchmal willkürlich in Cloud Umgebungen oder temporär in anderen öffentlichen oder privaten Container-Registern verteilt. Dies führt zu Problemen, wenn der Entwickler einige symmetrische oder asymmetrische eingebettete Geheimnisse wie SSH-Schlüssel, Klartext-Passwörter, Zertifikate und viele weitere integriert hat.

Die Idee hinter dieser Arbeit ist die Entwicklung eines Ansatzes zur Erkennung eines geheimen privaten RSA-Schlüssels und eines Amazon-Zugriffstokens, die beide von Entwicklern in einem Anwendungscontainer-Image verwendet werden. Verwandte Arbeiten stehen zur Verfügung, um Geheimnisse im Quellcode im Allgemeinen, aber nicht direkt in Container-Bildern zu entdecken. Diese Arbeit zeigt am Beispiel von Docker, dass eine Anwendung bekannter Key-Leak-Techniken zur Erkennung eingebetteter Geheimnisse möglich ist.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABKÜRZUNGSVERZEICHNIS

Part I

THESIS

1 INTRODUCTION

Nowadays the use of application containers has become indispensable. The advantages of those application containers are obvious to developers and system engineers. Container technology provides isolation and portability for any built application. Applications are the most common form of consumer software, but most of all software that is needed in the business environment, such as large databases and web applications.

In enterprise environments, several containers also often form one application. This means that the given isolation has to be partially removed and intercommunication between these containers has to be enabled in order to provide a running application stack. Therefore 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 now presupposes that developers are security-conscious and adhere to security standards. A container is usually an instance of an image and 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 direct access to the image also automatically gets access to the file system within the image. If a developer has integrated secrets statically in order to create a secure communication across the internet into the image, these are easily visible to curious people or attackers. These secrets can be tapped by attackers, exploited and thus become obsolete. That is a fatal problem in container images in general.

In addition to this problem, there are others such as the trust of images regarding integrity [1]. A popular attack is a man in the middle attack in this case. There is a signing process necessary with e.g. Docker content trust or a non-central blockchain approach [10].

Even obsolete packages are a problem for container images. Obsolete packages often contain vulnerabilities that are published and visible 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 will only focus on the problem of the embedded secrets in container images, which are totally necessary for containers to communicate in a secure manner with other containers or services.

Nowadays Docker is a famous container technology. Docker allows deployments in cloud native environments, on bare metal machines on Windows Systems, Mac OS and Linux operating systems. Latest IT-news proves that Docker is currently undergoing a radical change and that there will be

alternative solutions in the future [12]. Since containers are still promising for the future, this work will provide exemplary samples with Docker.

However, this work can be adapted to other container technologies, since Linux core features still form the basis.

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. That is the basis on which the thesis starts its work with Docker as an example.

1.1 MOTIVATION

Secrets are used by almost every software that needs a secure communication in an unfaithful environment. For example, software that transfers sensitive data through the internet to a service-endpoint. The main goal of secrets are to tackle challenges like confidentiality, integrity, authenticity and accountability. A key leak would lead to a collapse of the above protection objectives in general. If a key has been integrated into a container image, a key leak is inevitable. It is not difficult to steal keys from an image, since the attacker only needs to launch the container from the image to gain access to the file. The attacker can afterwards scan the file system for tokens, or check the source code or encapsulated archives and binaries for secrets.

Even if it is unknown and not allowed to the developer, nothing and no one can prevent technically the static integration of secrets. The static integration is a pure failure of the developer.

The development of an image analysis for embedded secrets allows a better control of the secrets used. This does not work preventively but can be used as a control instance before further processing or delivery.

1.2 GOAL OF THIS WORK

This work tries to develop an approach to detect embedded secrets in container images. This is approached structurally by a theoretically developed concept, which contains various elementary building blocks. These include access to the container image as well as analysis and other points. The concept limits the goals to be aimed at to two important secrets in order to set a certain scope. In addition, new evaluated theoretical concepts are presented in this article using prototype applications. The prototypes are available on GitHub and on the attached CD. Finally, this prototype is applied in practice to obtain the result of the theoretical concept.

1.3 STRUCTURE

Before falling into details its good to have a structure of this paper in mind. This helps for the following work in the best possible way. This work is basically targeted for everyone who is interested in computer science with basic knowledge about Linux architecture and container virtualization.

In this second chapter right after this section, it starts with container technology introduction in general related to Linux. The difference to classical virtualization is described briefly. Linux namespaces are also explained, because they represent an important functionality in the container area followed by a brief introduction to cgroups. There follows a deep dive into container images which form the basis for the work. This is an essential section that contains the functionality + usage of the union mount file system OVERLAY2.

The third chapter introduces related work regarding key leak techniques in general with source codes and file systems. These techniques will be adapted in the following chapters.

The fourth chapter will contain a theoretical approaches to analyse the image. That includes the secret types to detect, the image access which has to be used and the processing of the image itself. This chapter afterwards focuses in particular on the practical development of evaluated methods to verify and compare the elaborated approaches.

Finally, there will be a summary of the collected results and a solid basis for future scientific work.

The next chapter will introduce as mentioned with containerization insights.

2

BACKGROUND CONTAINER TECHNOLOGY

2.1 CLASSIFICATION AND PLACEMENT

Containerization technology has been introduced in BSD in the early 2000s and finally established in the Linux kernel in 2008 [7]. The resulting Linux containers(LXC's) are much more flexible and closer to the underlying host operating system than the classical full virtualization strategy from the 1960s. Figure 2.1 shows the architectural difference between container and full virtualization.

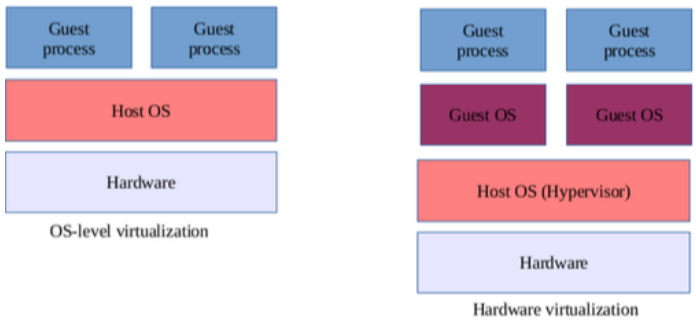

Figure 2.1: Difference between container and full virtualization

The fact about the architectural difference leads to important facts. A container is a lightweight alternative to full machine virtualization. In fact a container does not require as seen in 2.1 a virtualized hardware. The container is just using and sharing the underlying kernel from the host.

Containers themselves are basically operated as stateless and distinct units, which are usually provided and destroyed by an orchestrator or by hand. If updates are available, the containers are simply replaced. This enables developers and system engineers to make and push changes to apps at a much faster pace.

The immutability of containers also affects data persistence. Instead of mixing the app with the data used, containers emphasize the concept of isolation. Data persistence should not be achieved by simply writing to the container root file system, but by using external, persistent data stores such as databases or volumes. Data should be used outside the containers themselves, so that when a container is replaced by a new version, all data is still available for the new container.

From BSD's first rudimentary container approach with jails to Linux LXC's to Docker, native Linux features provide the functional foundation for encapsulation between host and deployed containers. These necessary isolation

and permissions concepts of a recent Linux system(Kernel version 5.3.11) are described in the next sections.

2.2 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. There are the following namespaces available under Linux:

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

It doesn't make any sense to describe every namespace in detail. In the following only essential namespaces are explained.

2.2.1 *Mount*

Mount namespaces provide an isolation of the list of mount points seen by the processes in each namespace instance. Thus, the 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 2.4.

2.2.2 *IPC*

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

2.2.3 *PID*

Traditional 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. Each time a Linux system starts, it starts with only process

PID1. This process is the root of the process tree and initiates the rest of the system by starting different handlers and services. All other processes start below this process in the tree. The basic idea behind PID namespaces are 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 itself. With PID namespace isolation, the processes in the lower-level namespace have no possibility of detecting the existence of the higher-level process. This ensures that processes that belong to a process tree do not inspect or kill processes in other process trees of siblings or higher-level processes. However, processes in the higher-level namespace have a full view in the lower-level namespace of processes, as if they were another process in the higher-level namespace.

2.2.4 *Network*

Due to the global instance of that network interface on a single host it is possible across the whole operating system to create or edit routing + arp table entries with correct granted permissions. With network namespaces, it is possible to have totally different instances of network interfaces and routing + arp tables that operate independent of each other. This prevents communication between network namespaces.

Network namespaces are complex, but important to know in order to remove partially the isolation and establish communication between containers and hosts, and between containers themselves. The following list shows the standardized CNI (Container Network Interface) workflow, 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 veth pairs
4. Attach veth 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 below. 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 image. C1 in this picture is just a prefix for the underlying host 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 across 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.

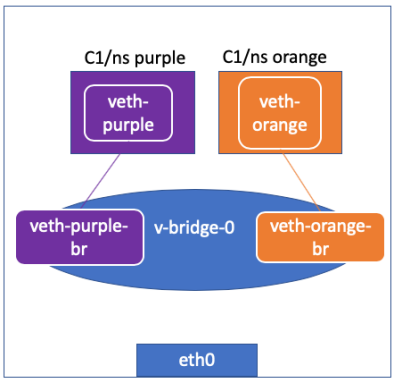

Figure 2.2: Example of basic network within a container cluster

This is just the beginning of network communication through namespaces and CNI. But just inter communication between different namespaces on a single host isn't sufficient. Several steps are required to communicate with another host externally and over the Internet.

Figure **??** shows a more comprehensive setup. There is shown 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. If the blue namespace network wants to access the endpoint with the IP address 192.168.1.3, there is a routing table entry in the blue network namespace like the following necessary

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

This allows only one 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 its just necessary to add a default route

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

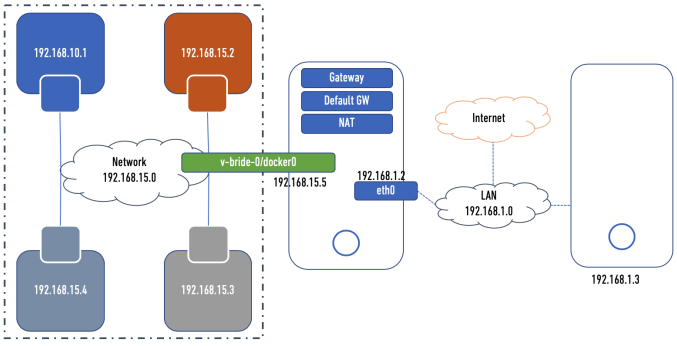

Figure 2.3: Example of comprehensive networking within and outside of container

This example of network namespaces, which are associated to different processes, shows the power and flexibility of network namespaces. Normally these manual steps are done through CLI via a daemon like Docker. This in turn can be managed by an orchestrator like Kubernetes or Mesos.

At next there is just one Linux namespace left, the user namespace.

2.2.5 *User*

User Namespaces insulates user and group IDs so they appear different both inside 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. In addition it appears as if another process is being executed by a user with low permissions. User namespaces can also operate on capabilities and privileges. 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 today 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, which provide another level of security in order to provide a smooth working container environment.

2.3 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 obviously manifest themselves as instruments to prevent runaway containers and denial of service attacks. The following lists contain the resources are controlled via 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 gained that containers can work isolated just via Linux native functions. The next important section gives an architectural introduction to the root of every container - a container image.

2.4 UNIONFS

As already mentioned in 1 this paper will use Docker as container technology. Before diving into container- and especially Docker image insights, it is required to have knowledge about union file systems. Union file systems build the basis for container images in general. The first subsection will explain one important file system, the Overlay2 file system. The second subsection gives an architectural overview of docker images in general. Finally the last subsection will introduce the interaction between a Docker image and Overlay2

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 - can be read-only, and could 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 properties of an Overlay2 file system are now explained using an example. First, the following folder structure is assumed 2.1.

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

Listing 2.1: Tree orig

The folder structure contains all the mandatory Overlay2 elements to work properly. Because of the Overlay2 is directly integrated in the Linux kernel, the mount point can be created without additional software packages. This is illustrated by the following mount command.

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

First, the mount command must know what type of file system to mount. This information is provided by the -t, which stands for type. In this case it is an overlay type. The next flag -o allows to add options to mount the specific

filesystem with the associated necessary components. Each options is separated by a comma. In this example, the option lowerdir is set to a chain of folders separated by a colon. The lowerdir arguments takes only the lower1 and lower2 directory. The upperdir is the next option and is set to the upper folder of the provided hierarchy. The worker options represents a single folder and is in this to set work. Finally, overlay is the last keyword that needs an argument for providing the union mount. The mount command uses the merged directory for the overlay.

In the next step, the directories get to demonstrate the behavior of the Overlay2 file system.

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

Listing 2.2: Tree filled

As expected, a file creation in one of the lower- and upper-directories is visible in the merged-directory.

Whereever objects with the same name exist in upper- and lower-directories, then their treatment depends on the object type:

A **file** object in the upper directory tree appears in the overlay, whilst the object in the lower directory tree is hidden.

The contents of each **directory** object are merged to create a combined directory object in the overlay.

When a file or directory that originates in the upper directory is removed from the overlay, it's also removed from the upper directory. If a file or directory that originates in the lower directory is removed from the overlay-directory, it remains in the lower directory, but a 'whiteout' 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. The result of the whiteout creation means that the object in the lower-directory is ignored, whilst the whiteout itself is not visible in the overlay-directory.

Another important fact is the copy on write strategy. When a file is modified, which already exists in an underlying layer and the storage driver manager will take care of copying files to the upper layer and the differences will be applied. This is an efficient resource-management technique because operations may just need a copy instead of creating a new file.

Now the general knowledge about the Overlay2 file system is known. This is important to understand a main component of this paper, the Docker image. This will be viewed next.

2.5 DOCKER IMAGE

In this section, the Docker image is explained in more detail, as it is an important part of the work. First, the architecture of a Docker image is presented with the necessary information for further work, since the overall architecture with all information would make the basic chapter too large. Then the construct for building a Docker image is explained. This is called Dockerfile and in the introduction the key factors and usage of Dockerfile is explained. Finally, the last subsection describes the metadata information, as it is important to follow the theoretical concept.

2.5.1 *Docker Image architecture*

A Docker image is ultimately a stack of selected file system layers to provide a starting point for a container. Fig 2.4 shows how a Docker image is stacked. At the bottom there is the Linux kernel. On top of that takes two different image layers place. In this case it is a Debian and a Busybox. Both of them are runnable base images. 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 itself doesn't have another further layer stacked on the top. When a container is launched from an image, Docker finally attaches a read/write file system across all underlying layers, as it is seen on both images in this example.

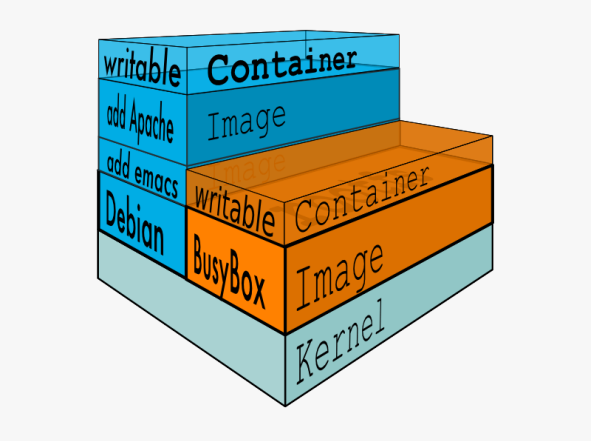

Figure 2.4: Docker filesystems stacked to represent an image

In order to manage images and especially corresponding file system layers, Docker uses storage drivers. Each storage driver handles the implementation differently. There are several storage drivers available like, ZFS, BTRFS and many more which can be configured by the responsible system-engineer or

developer. 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 provides a result tailored to show only filesystem level information, as this is the interesting part for the docker 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 mechanisms of the Overlay2 union file system 2.4 are used in order to provide a starting point for a container. Each component of Overlay2 is found and is used by Docker. This represents once more, that Docker is using only Linux well known core functions instead of building own mechanisms.

One interesting and important fact is the mount process. The lower-directories which represents a readonly structure are not mounted directly by their folder name. Docker uses for each image layer a symbol link, which redirects to the original folder. The reason belongs only to the length of the folder name, which is in total a length of 65. 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. That can be seen in listing 2.3 as well. In each layer are additional helper files available like the lowerfile, which creates a relation to associated parent, if a parent exist. The lower files are responsible for creating the correct order from the most upper layer to the lower layers. Due to this fact will be a correct order provided, since it is necessary for a correct behavior of a union file system.

Furthermore, every 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. The next section describes the construct of building a Docker image, called Dockerfile.

2.5.2 *Dockerfile*

In order to build a Docker image, Docker Inc. introduced a construct called Dockerfile. 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]. The following listing in 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. The COPY command adds an example bash script from the Docker client's current directory. The RUN command makes the program executable. Finally, the last layer specifies which command will be executed inside the container.

An image can be created with the corresponding Dockerfile and the command Docker build, like it is shown below.

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

It is valuable to know the Dockerfile construct, because it is finally responsible for integrating secrets into an image. From a logical view, there are two categories in a Dockerfile which contain commands that are responsible for integrating static files. First, *direct integration* which contains the actions COPY and ADD. The difference between the two commands lies in the range of functions. ADD can as an example request an URL or unzip an archive directly to the endpoint.

The second category *indirect integration* is a bit more comprehensive. This category contains 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 in the Dockerfile.

At this time it is known what a Docker image is and how it can be built. Also, it is known to integrate static files into an image in an indirect and direct way. The next subsection will introduce the important topic metadata of an image which will be definitely used in the following chapters.

2.5.3 *Docker Metadata*

Another important feature is the metadata of an image. Every Docker image contains several meta-information of the build process. These meta-informations are directly accessible through the history feature. These informations are locally provided and accessible for the root user. It is assumed in this work that no manipulation of the metadata was performed locally. An integrity check would be desirable and need further investigation.

Using Ubuntu 18.04 as an example, the meta-information looks like this below 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

This listing might be confusing but it is helpful to just get an overview how it is structured, and how it can be helpful for the next chapters. It can be seen that the schema of this meta information is similar to JSON. The only difference is that numeral values are not marked in quotations. In this structure there are many attributes with their corresponding values. Attributes like "Created", "CreatyBy" are first followed by "Id" and many more. Important informations are most of all Dockerfile instructions like COPY, ADD an RUN, since they are responsible for integrating static files into the image. In this example it can be seen that only an ADD command is used. A COPY method is not 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 visible.

In theory it is easy to analyse these informations because it is readable for computers because of the javascript object notation similarity. That makes this history feature of Docker very important especially when it comes to the (pre-)processing of Docker images. Important properties can be extracted, such as Dockerfile keywords and corresponding parameters.

This chapter provided a basic knowledge about the architecture Docker images and the interaction with Overlay2. This knowledge is fundamental to have an idea how the file structure is working behind the scenes. It gives some ideas for the theoretical concept to analyse such an image for static secrets. Before this is possible some related work is shown about key leak techniques. Key leak techniques are essential in order to detect secrets.

3 KNOWN KEY LEAK TECHNIQUES

Scientific work on detection of secrets in source codes has been done in [6]. Different procedures are shown, which can be used to detect different types of secrets. Mainly API tokens from Amazon and Facebook were handled in the paper. They explained methods like sample selection using keyword search, pattern-based search, heuristics-driven filtering and source-based program slicing.

The first approach focuses on fixed strings in sensitive key 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 exemplary keyword approach applies well when keys use fixed string prefixes. It is not sufficient for private keys that have no fixed prefixes.

Due to this fact, the referenced paper is more focusing on pattern-based search. The pattern-based search is not restricted to strings, but to regular expressions and is suitable for strings with a fixed schema. As in most cases, a pattern-based search also has its drawbacks in the form of false-positives. This is why in [6] heuristics and source slicing are used to reduce false-positives and to increase the efficiency. Heuristics have been tested by looking at cases where a matching secret key appears within 5 lines of each other. This approach is usually precise in relation to the results achieved, but it is possible to miss leaked keys where the credentials are not close together. A different tested heuristically approach to reduce false-positives is trying and guessing whether they are auto-generated or hand-written, as they noticed that they are common false-positives. A further framework was used for this approach and the false-positive rate was successful decreased.

The source-based program slicing approach is a very efficient approach. In case of API keys from Amazon and Facebook they had a 100% efficiency detection-rate with the use of a customized program slicing method. Source-based program slicing is a complex feature which is evaluated in [11] in depth.

Due to the fact, that heuristics-driven filtering and source-based program slicing is used to reduce the amount of false positives and to increase the efficiency, the following concept will only use the main mechanism of the work in [6]. This includes the well working keyword search and pattern-based search. The reason for this is to prove that the detection of secrets in container images is possible in general. When this is possible an extension of heuristics-driven filtering and source-based program slicing will in theory

further improve the results. The following chapter will first adapt these key mechanism like keyword search and pattern-based search in order to create a theoretical concept to detect secrets in container images with docker as example.

4

A THEORETICAL CONCEPT TO ANALYSE A DOCKER IMAGE

This chapter will evaluate and develop a theoretical concept to detect secrets in Docker images. It starts with setting a scope of secrets to detect. The section then will introduce which method has to be used to gain file access to images in general. Finally, 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, a scope of secrets to discover is necessary. In this section, this is briefly and meaningfully defined.

Software developers and system engineers are the main stakeholders who create container images regularly. The technical environments are often cloud services nowadays. Many cloud service players are Amazon, Microsoft and Google. There are many more service providers available. Common features of those global players shows the following list

- 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. Apart from the obvious features, there are many other APIs accessible. As an example. Google offers more than 100 API's for developers. Fortunately each cloud provider has also Identity and Access Management (IAM) integrated for providing the principle of least privilege. But that's 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. Whether the token comes from Google, Microsoft or Amazon only plays a secondary role. Important with the token is to recognize a corresponding architecture or scheme when it comes to the analysis. 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 be also 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 is usually RSA. RSA is widely used and still the state of the 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, it becomes 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. That makes the place and name of the private key arbitrary. The key file can be placed and named where ever the developer sees the necessity. It just needs a corresponding correctly configured client to find the key file. 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 and needs also much 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 makes off with the private key file. The passphrase is just a key 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. In order to use the private key for public-key encryption, it is first needed to decrypt its file using the decryption key. If an additional passphrase is included above the private key, it must also be included in 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 and 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 itself 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, this leads to the possibility to compromise the passphrase of the private key part. Passphrase are arbitrarily chosen and does not follow a syntax. Therefore no key based search oder pattern-based search is possible to find these passwords. This is another hurdle to find the password, but 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 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 exist 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 section 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 belongs to get an universal access to an image in order to perform a depth analysis. The architecture of Docker images leads at the end to three obvious possibilities which are described in the following subsections. These different approaches have advantages and disadvantages. The key feature of the access method is a well working interaction with the necessary analysing module. When the file access method needs big effort or it modifies an image in order to embed necessary actions that much, it can have a bad impact for the whole scanning workflow in general. This has to be considered when the decision is made which access method has to be used finally.

4.2.1 *Additional image layer*

The access is made through a new additional layer which contains and adds a program to the obtained image. When the image is initiated and started as a container, the program can work on the entire file system and get the job done. Before the new layer is added, preprocessing on the original image is mandatory and this first result has to be temporary saved. That includes all the metadata informations which are necessary for the scan. These metadata were explained in 2.5.3 and these metadata will be used by the processing module later in 5.3.2. A change of the base image would lead to a lost of these important informations. When 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 export 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 whole needs a manual overlay-mount on the host system itself. Necessary informations to mount the overlay correctly is proved by the the background chapter 2.5. 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 examined through the provided informations which are available locally. That includes especially the lowerfile of each Docker layer. This information has to be used in order to get finally 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. Also it needs kind of 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 has only the image as a base and does not modify anything on the image and only needs a start of a container temporarily. Unlike in 4.2.1 the container has not 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 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. Finally 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.

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 is the following. First of all the abstractive flowchart 4.1 will be explained in order to get an overview of 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 takes care about old Docker images and downloads the target image from a container registry. Afterwards the preprocessing module extracts 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. When no keywords like that are existing, no further image investigation will take place. 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 2.5.3. If the analysis should take place, then the image will be mounted by the image-mount module to be able to work on it. The necessary mounting informations are already available as mentioned in 2.5.1. 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. Afterwards a result is 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. That includes working with standardized structures and scalable microservices.

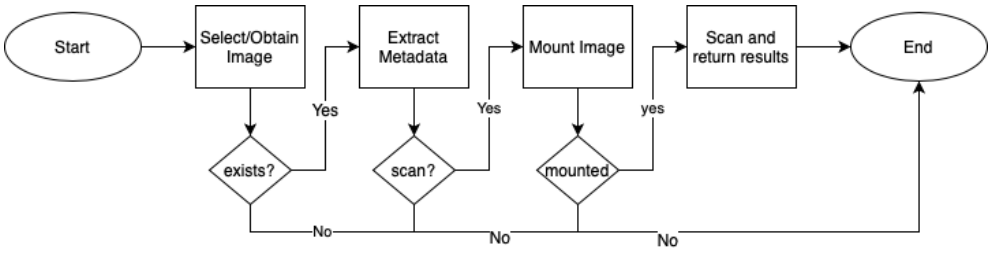

Figure 4.1: Abstract view of analysing process

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 provides already 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 theoretical, 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 continues 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 module is very important as it forms the basis for the file scan module. First of all, it decides whether the specified image needs to be scanned or not and if necessary, it decides which part needs to be scanned for embedded secrets. 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 2.5.3.

In case of further investigation, the image pre-processing module has to handle the Dockerfile actions ADD/COPY and RUN differently. This belongs to the different category which are already known from the background chapter. In the case 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. Then it can be seen where the file is copied to. Especially the destination path is the interesting part. 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 paths where secrets live. 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 problem and can be 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.

Next there is the RUN command. RUN implies the simple bash command which exist already in each metadata entry in the history. That 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 list shows some 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. 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. Also it is important to recognize only these 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. That forms the next potential targets, which must be considered by the scanning module.Also here when the targets are examined, the special interrelation between folders like duplicated folders has to be considered as well like 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, 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 which is not listed from the static root folder, deviates from the original file system and sets a certain difference. This difference sets finally 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 the advantage is 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 use only trustworthy base images, since not only key problems exist. Also other types of

security vulnerabilities may be included such as manipulated application packages as an example. Also It is 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 allows 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 next.

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. Since this module mainly takes care of providing the real data, this is an important point. In fact, this would have 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 do that too. 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, but also 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 2.4. When the mounting process fails, this mounting point has to be removed completely and the mounting procedure is repeated once. If there are still problems, the mount process is aborted and the termination of the program is initiated. If the mount process was successful the scan module can be triggered in order to detect potentially secrets. This module will be explained next.

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 rather to 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}
```

In case of RSA, it is not clear which is the name and the file extension of the private key. That is 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 within a folder there can be further folders in which there can be nested 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, this 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 **??** shows a more detailed sequence of analysis. It's a a pure logical sequence without going into technical details and implementations with help of example technologies.

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 a theoretical concept to detect embedded secrets in Docker images. The next chapter concentrates and describes the practical realization.

5 PRACTICAL REALIZATION

This chapter will demonstrate one possible practical realization of the theoretical concept. The demonstration is structured in the following sections.

SUBSECTION 5.1 system-environment

SUBSECTION 5.2 prototype strucutre

SUBSECTION 5.3 implementation core modules

5.1 INTRODUCTION SYSTEM-ENVIRONMENT

Containerization is successfully established in Linux environments, but also available in others, like Windows and MacOS. In Windows and MacOS, containerization via Docker is implemented through the use of an emulated Linux underneath. Due to the recommendation to use Docker with Linux, the prototype is developed for a pure Linux environment. 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 easily provided by customising the path definitions. More informations about customisation and scalability is provided and demonstrated by the feature in ref bla bla.

The landscape of the system environment, including working environment is shown in Figure 5.1. The working machine is a MacOS which pro-

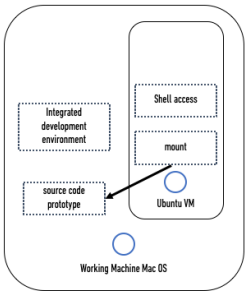

Figure 5.1: System- and working environment

vides locally a headless Ubuntu as a virtual machine, which is a Debian derivative. 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. That corresponds to the idea that the prototype is operated and tested on a plain Linux system.

For this prototype Python is used as programming language. The decision of using Python can be made very straight forward in this context. First, it exists a very useful Python library for the Docker Engine API. This API allows everything the Docker command does. This means run containers, managing containers, obtaining various information and much more which is available in the official docs [3]. Also, Python is an interpreted programming language. It allows it 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 build the key features, which makes the pace of coding and testing easier and faster during the development process in the working environment. Furthermore, Python provides an easy syntax which is readable by any english speaker. It is like pseudocode reading, which makes it easier for non Python developers to adapt this prototype in a bigger project with another technology stack. Last but not least, Python provides a helpful module called virtual environment (venv). Virtual environments are a way to create a dedicated Python environment that allows packages to be installed, modified, and used without disturbing the global Python binary. This feature will be used to manage the necessary packages like Docker other used packages. In particular, the function is of benefit when the tool is delivered to a remote system. This tool then works independently of existing Python instances. In this context it is worth to know, that Python in version 2.x is not supported anymore since January 2020 [2]. The conclusion is that the latest version of Python is used. The same applies to the package manager Pip. The package manager should correspond to Python in a compatible version. Exactly versions of the this prototype is Python 3.7 and Pip3.

Now the system environment is set and which programming language has to be used as well. The next section will provide the general project structure of the prototype.

5.2 PROTOTYPE STRUCTURE

The following project structure in listing 5.1 gives an overview of the prototype how it is built 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: Prototype structure in Python

The root folder contains the main program the analysing_manager.py, a modules folder and the venv directory. The venv directory builds the virtual environment feature of Python, while the modules folder contains the essential modules which represents the core function of the prototype, namely obtain, preprocessing, mount and scan. These python-modules represents the modules in 4.3 from the theoretical concept. These are managed and used by an analysing_manager module which lies in parent (root) folder.

The analysing_manager module is mainly in interaction with the core modules. The core modules are deliberately created as distinct units. That makes it easier to maintain the prototype. Changing a single core module instead of a large monolith promises more flexibility and counteracts problems. A replacement of core modules is also possible since the interface to the analysing_manager is untouched. Through the analysing manager module exists a central endpoint to request the prototype. This is especially useful for a potential microservice. Restpoints have a single place to interact with and to start the analysing. Also other common technologies like RPC's from Apache thrift or Googles protocol buffer can be used to trigger this single analysing endpoint.

When the analysing manager is triggered with an image name as input, the procedure which is shown belowwill automatically start.

```
analysing = AnalManager(img_name)
analysing.prepare_environment()
analysing.preprocess()
if analysing.necessary:
     analysing.mount()
     analysing.examine()
```

First it starts with analysing.prepare_environment() where the obtaining module 5.3.1 is doing its job. The Docker image will be downloaded by this module. Afterwards the preprocessing will take place through the call of the preprocess method. The preprocessing will be done by the implemented preprocessing module 5.3.2. Finally the analysing manager can decide if a further investigation is necessary. If so the mount and scan module will be instantiated. The synchronous mount call, as well as the examine call is is executed consecutively. The final result of the scan module is just printed out into the stdout stream of Linux. When secrets are found, the secret will be printed to the stdout stream. When no secret was found, 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 of the prototype in general. The following section will introduce the implementation of the core modules.

5.3 IMPLEMENTATION CORE MODULES

This section demonstrates the prototypical realization of the main modules, namely obtain, preprocessing, mount, scan and their interaction through the associated manager. Each module is dedicated to one section. The order of the modules to be described is based on the sequence of the analysis process from the theory chapter 4.3. 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.

First it starts with the obaint module.

5.3.1 *obtain.py*

The obtaining module has to take care about garbage of the environment and pulling the target image. This module is build very slim and contains only a few methods.

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

The procedure consists of two steps, which are represented by the methods stop_all_containers(self) and remove_old_containers(self). First, possible running containers needs to be stopped and afterwards all locally existing images and therefore all overlay directories on the filesystem has to be removed. This is programmatically done through the equivalent Docker stop and Docker remove command as shown in the code-snippet 5.2. These commands are accessible through the available Docker SDK, which has to be installed and integrated into the virtual environment before. The code-snippet itself is explanation enough. It is just 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, which tries to download the image with the given name. If no tag is specified, the latest tag of the image will be used. This is achieved by simple string manipulation since it is known that the image name and tag are separated by colon as described in 2.5.1. 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. The corresponding code snippet to this function can be seen in the code-snippet 5.2 as well.

```
#   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

5.3.2 *preprocessing.py*

The pre-processing module decides whether an image needs to be scanned and if so, it decides which parts need to be scanned. 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. To get an overview of the logic, the following function list helps. 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 collect_metadata function is the first important method. It initilizes the Docker environment in order to fetch the locally provided target image. Afterwards the extraction of this fetched image takes place. This is done through the equivalent Docker history command from the Docker SDK. The history provides metadata informations, which are explained in 2.5.3. This metadata will be saved 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 contains_key_actions method. As developed in the concept in 4.3.2, an image has to be scanned when keywords such as ADD, COPY or any RUN commands have been used during the building process. This can be determined with help of a proper data-structure compared against the metadata information of the image. That data-structure includes proper keywords and

an additional related variable if this keyword has been used. The Python dictionary is a data-structure to provide one or more key:value pairs. The key:value pair represents the keyword with a status respectively. 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 5.3. When an entry from the dictionary is detected, the corresponding value will be updated to true. Finally the whole data_structure will be checked and a boolean is set and returned. The method contains_key_actions(self) returns true if any key values are set to true. That means in general that a further investigation of the image is mandatory. Only when every value is still false, the boolean is set to False too and no further investigation is necessary.

```
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

The next important functions of this preprocessing module are fetch_direct_copy_targets and fetch_indirect_copy_targets. Both functions have in common to extract and to return detected path(s) used by COPY, ADD or RUN. The direct copy function takes care about fetching targets which has been integrated via ADD and COPY. The indirect methods takes care about the indirect integration via openssl, wget, git clone and ssh-keygen. The implementation of both methods looks different. The method for direct copying 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"
```

If this pattern matches in the meta information, a string slicing takes place in order to extract the target path. Simple string slicing with determination of the position of special delimiter signs is possible. This belongs to the fixed syntax in the meta information. The syntax is readable because of the JSON similarity. The examination of the target path realised in Python can

be seen in the code-snippet 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

In principle, each match from the Regex pattern is processed further. A list forms the data structure to which the determining target folders are appended. The determination starts with examining the target_path. When the target_path is the root directory, the next steps are omitted and the next round of the loop starts. This belongs only to the last entry, which 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.

When the loop is continuing normally, a temporary metadata is extracted in order to set the current WORKDIR for the corresponding target_path. Since the WORKDIR variable can be set more often in the build-process and can therefore occur more often in the metadata, the WORKDIR variable must be determined exactly. 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 target_path. The WORKDIR variable is important when the target_path is relativ instead of absolute as known from **??**.

After setting this WORKDIR it is important to distinguish between the root path, relativ paths and absolute paths. This can be examined as seen in the pseudo snippet with analysing the first character of the already known target_path. When the root is recognised, a simple adding of '/.' to the target_list is made. If a relative path is recognized a concatenation will take place in order to set the correct target folder. One exception is the point, which represents just the examined WORKDIR variable. In this case the WORK is the target directory. Finally the full examined target directory to scan is appended to the list. The array is returned after this processing.

As already known fetch_indirect_copy_targets(self) searches in the meta information for special programs. These programs are recognized in a regex pattern which was defined in the theoretical concept 4.3.2.

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

When any of these commands are recognized, a helper function to this associated command is called. The helper function is mainly responsible for the string slicing. It expects the output option of the related tool and the current index of the option, where the option has been found. The output option may differ depending on the command. Thats is why a helper function comes in place which is called by the parent method. The helper function will determine and return the output path finally. The helper function may be called multiple times, depending of the amount of matches in the meta-data. 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. As already known, this applies to COPY and ADD as well and will be handled in this method RUN in the same way. Just as with COPY and ADD, absolute and relative paths play a role here. These are treated identically, because they all have the same goal. A full in depth insight of the corresponding work between the fetch_indirect_copy_targets(self) and the helper functions can be seen in source code of the attached CD.

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

At this point the analysing_handler module can trigger to pull an image and manage that preprocessing of that image. The target directories are examined and the necessary mount module can do its job. 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 creating the working directory environment. As written in 2.4 there are a couple of working directories necessary to mount an Overlay2 file system. Every folder in except of the lower directories, which represents the image

itself will be created or deleted. The creation and deletion is programmatically done via the available os package and is not worth to show at this point, since it is just a creation and delete process triggered. It is important to note that the lower directories are not created or deleted. The lower directories are automatically deleted when the obtaining module starts the cleaning up the garbage.

The method mount_overlay first examines the lower directories of the overlay hierarchy. When these are examined, every information to mount an overlay is provided. The examination of the lower directories is done by a helper function. As known from the background chapter, every Docker image-layer is connected through a lowerfile. The lowerfile of the highest level of the image contains the most lower directories. The main task of the helper function is to determines this lowerfile. An additional Python package called "path" helps to walk through the local provided Docker layers in order to provide access to all lowerfiles. The helper function returns finally the content of the lowerfile of the highest layer of the Docker image. The examination is shown in the code-snippet 5.5 below and deserves an explanation.

```
def get_lower_directories(self):
    result = []
    for root, dirs, files in os.walk(self.overlay_path):
        if self.lower_file in files:
            fo = open(f"{root}/{self.lower_file}", "r")
            result.append(fo.readline())

    return max(result, key=len)
```

Listing 5.5: Python snippet - get lowerdirs

Initially an empty array is created. Afterwards a loop goes over every Docker image layer on the filesystem, which is represented by the overlay_path variable. In each folder exists this lowerfile. This lowerfile will opened in readonly mode and the raw content finally appended to the array. When the for loop is finished, a quantity calculation of the characters in each array entry can be performed. The entry with the most characters represents the final lowerfile. If this is examined and the content returned, the information of the lowerfile can be directly used as a parameter for the mounting process after by the method mount_overlay(self). The mount command can be finally by a system call which looks like below.

```
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(self) and create_overlay_dirs(self) are triggered first. Afterwards a helper function to get a correct order of lower directories is called. With that gained information the actual mount process can be executed. The name of that overlay_name variable is important since it is used to unmount the process like 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*

Finally the scan module holds three important functions, excluding constructor method:

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

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

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

The idea behind this scan is to combine each prefix with the already determined target path list. In the prototype, two nested for-loops serve as the basis for this. The prefix and the element list form the cartesian product. 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.

```
for prefix in fix_strings:
            for dir in self.target_list:
                if dir is '/':
                    target_root = 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}' "'{}'" \;")
```

It is important to note that the local dir variable can be the root folder. In this case a special action takes place. Before the scan will start, a target_root will be examined by the function get_root_diff(). This function returns a deviation Linux root file system. This deviation forms the target to scan on root level. The scan method remains the same, no matter if the scan at root level or elsewhere takes place. During the search, the Linux find standard util searches for any kind of file (including hidden files) in the provided target order per iteration. The path builds the known overlay directory with the examined and prepared target folder, which is a merged union point. Each file which is found will be piped into the grep command which uses the first dimension of the cartesian product, the RSA prefix. Because of the grep options the result is finally printed to the stdout buffer.

The scan_for_aws_key(self) function doesn't includes a data_structure to hold prefixes. Instead it only needs a working pattern. This regex pattern is created by analysing the structure of the AWS token, which is explained in the theoretical concept. From a programmatically perspective, the main difference is the very slim scan function, since there is nor for in for loop necessary. This is beneficial to the analysis time and the performace. Because of the very similar implementation, nor further description is made at this point.

After the prototype has been described, it can be applied and tested. This will happen in the following evaluation chapter.

6 EVALUATION

This chapter will evaluate the prototypical implementation of the previous chapter 5. It starts with locally self-made Docker images up to public Docker images from a container-registry, which will be processed by the prototype. The self-made images will purposely contain the RSA and AWS secrets, while it is not clear if the public ones will contain these types of secrets. The goal is to prove that the prototype is working for other public images as well. The results of the locally images are shown quickly and a final result in relation to public images is briefly and concisely presented. These results provide a good basis for a subsequent discussion, which takes place in the last section of the chapter. The next section of this chapter will start as mentioned with creating Docker images locally.

6.1 SELF DEVELOPED IMAGES

The creation of Docker images containing the secrets to detect needs a directive. An arbitrary development of Dockerfiles and thus Docker images leads finally to chaos and misleading scan-results. The section structures the creation of image as follows. Basically there will be two categories of Docker images. The first one only contains the RSA secrets and the second one contains only AWS tokens. Each category deserves the creation of two images. In other words, there will be 2 Dockerfiles for the RSA and the AWS category. The reason for this is the creation of images that use either the direct or indirect method of static integration of secrets. In total there are 4 Docker images, which have to be created by developing a corresponding Dockerfile. It is important when developing a Dockerfile to consider the different cases. First, WORKDIR changes has to take place, since they are commonly used by developers. Finally it is important to use absolute and relative destination paths. These different cases and tasks will be considered now starting with the creating a Docker image, containing RSA keys.

```
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

Both variants of the created images have different properties that had to be considered. Both variant consists of WORKDIR changes, relative and absolute paths. The images are handed over to the analysemanager one after the other. Each time the Analysis Manager is restarted, the entire analysis workflow provided by the theory chapter is run through. This includes per analysis a new garbage collection, fetching the image, mounting and finally scanning the image. The program is finally started by this following command.

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

After starting this procedure the result is printed out. To be able to see the status of the process, console output is made at important points and written to the standard buffer

6.2 PUBLIC AVAILABLE IMAGES

6.3 RESULTS

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] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: a z3-based string solver for web application analysis." In: (Aug. 2013). DOI: 10.1145/2491411.2491456.

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