UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Fingerprinting and secret extraction in Docker multi-container applications via memory deduplication

Fingerprinting und Extrahieren von Geheimnissen in Docker Multi-Container-Anwendungen durch Speicherdeduplizierung

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Volodymyr Bezsmertnyi

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Die Arbeit ist im Rahmen einer Tätigkeit bei der Firma cbb software GmbH entstanden.
This thesis was created within an activity at the company cbb software GmbH.

Lübeck, den 18. Dezember 2019

Abstract

Docker is a widely utilized software engineering tool, that is supposed to ensure better portability of an application by packing it with all needed data and dependencies in a single file (Docker image). Then Docker images can be distributed to the customer or onto a server and launched in a Docker container as a screened process. A containerized process enhances security of the system as well, since it restricts any direct interaction with an operating system and only an OS kernel is shared between containers. If an attacker takes over a container, it mitigates a possible damage to the system by limiting the attackers abilities. Because most of the servers are virtualized by using virtual machines, there are use cases such as Container-as-a-Service, where a Docker container can be launched inside a virtual machine. An underlying hypervisor may have a memory deduplication activated across all virtual machines in order to save physical memory of the system. This introduces new threats to the virtual machines and processes running inside them due to the side-channel that comes with enabled memory deduplication. We show, that even if Docker container technology is supposed to harden security of the system, it is still possible for an adversary to fingerprint the system and extract some sensitive information, if a memory deduplication feature is enabled by a hypervisor and the attacker resides on the same physical machine. The result of this work is, that Docker container technology does not protect the system against attacks that are based on the memory deduplication side-channel.

Zusammenfassung

Docker ist ein weit verbreitetes Software-Engineering-Tool, das eine bessere Portabilität einer Anwendung gewährleisten soll, indem es sie mit allen benötigten Daten und Abhängigkeiten in einer einzigen Datei (Docker-Image) packt. Danach können Docker-Images an den Kunden oder auf einen Server verteilt und in einem Docker-Container als abgeschotteter Prozess gestartet werden. Ein containerisierter Prozess erhöht auch die Sicherheit des Systems, da er jede direkte Interaktion mit einem Betriebssystem einschränkt und nur ein Betriebssystem-Kernel zwischen Containern geteilt wird. Wenn ein Angreifer einen Container übernimmt, mildert der Container mögliche Schäden am System, indem er die Fähigkeiten des Angreifers einschränkt. Da die meisten Server mit Hilfe virtueller Maschinen virtualisiert werden, gibt es Anwendungsfälle wie Container-as-a-Service, in denen ein Docker-Container innerhalb einer virtuellen Maschine gestartet werden kann. Ein zugrunde liegender Hypervisor kann die Speicherdeduplizierung über alle virtuellen Maschinen hinweg aktiviert haben, um physischen Speicher des Systems zu sparen. Aufgrund des Seitenkanals, der durch eine aktivierte Speicherdeduplizierung möglich ist, führt dies zu neuen Bedrohungen für die virtuellen Maschinen und Prozesse, die in ihnen laufen. Wir zeigen, dass es für einen Angreifer immer noch möglich ist, das System zu fingerprinten und einige sensible Informationen zu extrahieren, wenn eine Funktion zur Speicherdeduplizierung von einem Hypervisor aktiviert ist und der Angreifer auf demselben physischen Rechner sitzt, obwohl die Docker-Container Technologie die Sicherheit des Systems erhöhen soll. Das Ergebnis dieser Arbeit ist, dass die Docker-Container Technologie das System nicht vor Angriffen schützt, die auf dem Seitenkanal der Speicherdeduplizierung basieren.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 18. Dezember 2019

Contents

1 Introduction

With popularity of useful techniques such as continuous integration/delivery (CI/CD) and DevOps, developers also need powerful tools that allow them to efficiently support and maintain e.g cloud-based infrastructures or to provide multiple services on a single machine. Such use cases require virtualization, which is tightly bound to performance limitations of hardware servers. Running a service on a virtual machine, which is one of several possible virtualization techniques, is very expensive from a performance point of view, because one physical host machine must simultaneously handle multiple full instances of heavyweight guest OS producing large overhead. With this background, containerization tools like Docker suit the requirements mentioned above very well and therefore become very helpful. Even though container security has not been completely examined, containers are already widely deployed for software development and providing microservices.
Although Docker is getting more popular, its security has not been thoroughly studied. Containerized processes are basically simple processes of an OS and therefore share an OS kernel. This fact is the biggest concern of security researches, since, in this case, process isolation relies on the OS capabilities of decoupling the process from the system while providing shared hardware resources at the same time. This work focuses on the concrete containerization tool - Docker in a Linux environment. This tool is written in Go and based on *libcontainer*, a library which allows to create and handle the containers on an abstract level. The library utilizes Linux kernel features such as namespaces, control groups and others in order to isolate a process.
The use of containers may be found in different scenarios. In one of the scenarios treated in this work, containers are used to provide microservices on servers of a cloud computing provider. Typically the cloud provider virtualizes server functionality and does not want customers to run their services directly in the host OS, because then boundaries between multiple customers/functionalities on the same physical server are thin and various threats to customers are getting available. Therefore, virtual machines are used to separate different environments from each other. This means, that there are multiple running VMs on a single physical server. This provides more security to customers and more scalability to cloud providers. In such manner cloud providers can create a new VM on the server for each customer or functionality. Finally the customer can order some computer resources in a cloud, install needed software and launch his services. So security

and scalability are established at the same time. Now it is up to the customer to decide how to deploy his services.

One pretty efficient, scalable and popular way to supply certain application functionality are microservices realized with Docker containers. Microservices are used in software architectures, where application functionality can be decomposed to standalone components which can be developed and deployed independently of the main application, as far as the communication interface is specified[mic]. This approach improves modularity of the application and allows more transparent architecture and testing. Therefore Docker is suited perfectly for such a use case. Developers can freely choose programming languages and tools for implementing required functionality. Then they can build and ship an image with all binaries and dependencies to a Docker repository or directly onto a server. Before launching a Docker container, it may be configured to fit into the applications architecture. The container configuration includes settings of CPU, memory and kernel resources, mounted volumes/folders, some network and IPC settings. Also the containers can be organized in one or multiple networks, which should be created before the containers are launched. All these settings are specified, when the container gets started. Thanks to the CLI of the Docker daemon, some scripts may be prepared for launching big applications consisting of multiple containers, networks and volumes. In this way Docker multi-container applications can be deployed relatively easy, which is a sign of good portability. If an application needs more distributed computing power, all needed Docker images can be pulled from a repository and directly launched in a container (of course, if the application architecture supports such cases), resulting in better scalability.

Nowadays everyone can launch Docker containers in a cloud or Docker hostings. Most likely, the container is going to be started by a Docker Engine inside a virtual machine and therefore has a certain probability to land on the same server among some other potential victim containers. In this case enabled memory deduplication can become rather a problem than a performance enhancing feature. Thus, all neighbor containers or virtual machines can become a target for the particular types of attacks. Since the side-channel is a pure software side-channel, the attacks, that we will present, can be launched independently of underlying hardware. That is why it is important to understand implications of memory deduplication, since if enabled it opens a side-channel and a vector for potential attacks.

The goal of this paper is to examine resistance to fingerprinting and key extraction attacks based on the memory deduplication side-channel launched from a Docker container. Given a system with an enabled memory deduplication and multiple virtual machines which provide microservices using Docker containers, an attacker with the ability to execute arbitrary code can utilize the memory deduplication side-channel for detecting the presence of libraries and their specific versions. System fingerprinting is a part of the reconnaissance phase, in which the attacker collects all available information about the system, in order to discover and exploit found vulnerabilities. So, first the attacker tries to find out, what software applications are running as Docker containers. By knowing versions of used software and libraries, the easiest thing she can do is checking, which known vulnerabilities a specific version has and then directly exploit them. Another thing, that the attacker can do to the system, is to examine of used libraries or software components. And so if the memory layout a particular software has certain properties, it may become possible for the attacker to extract some sensitive information. Both attacks can be done from a Docker container inside either the same or neighbor virtual machine and by abusing the same side-channel resulted from the memory deduplication mechanism under Linux - Kernel Same-page Merging(KSM), a system-wide feature that enhances memory performance by merging same pages into one and marking it with the Copy-On-Write (COW) flag. If an attacker writes into a deduplicated page, it takes significantly longer than writing to the unique page, because the page should be copied first. So, by measuring the time differences the attacker can detect if certain pages are present in the memory. We show, that even if Docker is supposed to harden security of a system by delimiting an executing process from an OS and other processes in the system, an adversary is still able to fingerprint the system and even extract some secrets from it, when the host OS has enabled memory deduplication and an attacked application has certain properties of a memory layout which are discussed in the next sections. For this we first provide some background information about virtualization technologies, Docker components and use cases, memory deduplication mechanism KSM and the side-channel which is implicated by enabled KSM. Furthermore we define an attacker model which is appropriate for the given use cases. As relevant attacks based on the memory deduplication side-channel we choose fingerprinting and secret extraction, therefore we analyze and describe both attacks. Then we present the results of tests in different configuration setups. Finally we conclude and summarize an outcome of this work.

2 Background

The following subsections provide information about topics that are essential for understanding the explained attacks, techniques and configurations. Since this work is about testing a security relevant side-channel in a system, in particular in Docker containers inside a virtual machine, it is important to understand concepts of virtualization technologies. Also a memory deduplication concept is described which is responsible for the side-channel. Furthermore a description of the side-channel, that all attacks are built on, is provided.

2.1 Virtualization Technologies

This subsection provides a brief introduction into the virtualization technologies such as hypervisor-based and container-based virtualization. The purpose of these technologies is to isolate systems or application spaces by controlling their resources while running on the same hardware. This can be useful for different purposes like, for example, Infrastructure-as-a-Service (IaaS), where a customer can rent an OS on the server and run his own services among other customers. IaaS is a good example, where a hypervisor may take care of the virtualization and separate environments of different customers. On the other hand, if one needs computing power, there are plenty Cloud Computing providers, where one can launch an application as a standalone container. In this case, the hardware resources will be leveraged by container-based virtualization. More detailed description about commonly used virtualization technologies follows in further subsections.

2.1.1 Hypervisor-based Virtualization

The main component of this technology is a **hypervisor**. A hypervisor allows running multiple virtual *guest* machines with dedicated OS on a single physical *host* machine.
There are two types of hypervisors: Type 1 (native or bare-metal) and Type 2 (hosted). Hosted hypervisors provide an additional abstraction layer between host and guest OSs and handle resources like CPU time and memory accesses on a software level. However, a native hypervisor runs directly on top of the underlying hardware and may utilize hardware virtualization features like Intel VT, which adds new instructions to x86 Intel processors that help to separate the execution of guest OS and native hypervisor instructions. In both cases, to run any service on one machine a complete virtual machine with a

guest OS should be mounted or installed, which includes kernel and all dependencies (see Figure 2.1a). The biggest disadvantage here is that multiple guest OSs and a lot of dependencies (which might repeat across different services) consume a great amount resources and time to start up a service. Therefore, the overall performance of a single service cluster decreases dramatically.

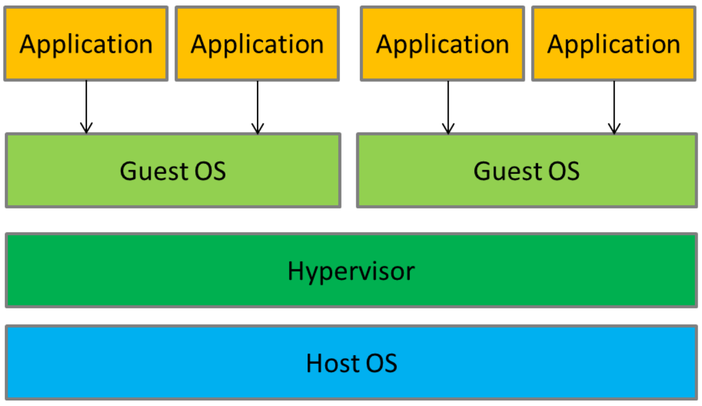
If an attacker wants to break out through the hosted hypervisor into the host OS, she should do it from regular user space or, what would happen in most cases, first execute a privilege escalation in order to become root on the guest OS and only then find and exploit a vulnerability in the hypervisor, if there is no exploit available from user space. The fact, that this combination of events is necessary, significantly lowers the probability of a breakout[LC]. Vulnerabilities that allow an attacker to bypass the hardware hypervisor, are found roughly every two years[BBB$^+$]. As a result, the hypervisor-based virtualization provides high security, but has high performance costs.

An example for a hypervisor is **Kernel-based Virtual Machine (KVM)**. KVM is a bare-metal hypervisor which is a virtualization module of the Linux kernel. In order to run a virtual machine, KVM requires certain hardware features such as Intel VT or AMD-V. If such features are present, KVM utilizes them and therefore virtual machines may have near-native performance. Also a software-based emulator **QEMU** is often used in cooperation with KVM. QEMU manages virtual machines from user space and can emulate different hardware types like CPU and I/O devices. KVM and QEMU are used together to run and handle resources for different guest operating systems. It is worth mentioning, that KVM includes a memory deduplication mechanism for memory performance optimization. The concept of memory deduplication is explained in Section 2.3.

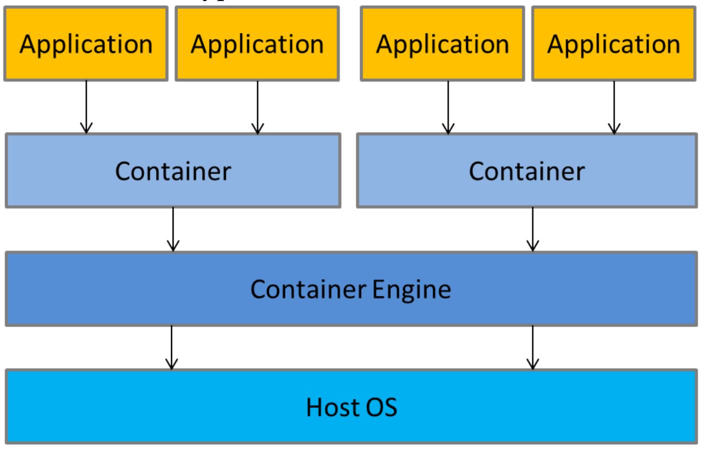
2.1.2 Container-based Virtualization

In contrast to the hypervisor-based solutions, a container itself is just a simple process running on a host OS and sharing the kernel with an underlying OS, hence rights separation and process isolation happen on a software level as shown in 2.1b. Since containers share the memory of the host OS, there are no duplicated kernels in memory, which makes containers more lightweight and portable than hypervisor-based virtualization. Even containers that are using different OS kernels are not causing much trouble, since Windows and Linux containers can run simultaneously on Windows machines[Pat]. Performance of containers is nearly equal of native applications and overall better than of a hypervisor-based solution as shown in[PZW$^+$07]. As a result, container-based virtualization has some important advantages and more and more developers start using it in their projects.

However, since containers run on the same kernel, escalating privileges in one container is sufficient to access the host OS. Privilege escalation vulnerabilities in the Linux kernel



(a) Hypervisor-based virtualization



(b) Container-based virtualization

Figure 2.1: Hypervisor-based and container-based virtualization.

are found roughly once a year[BBB^{+}]. Based on this and according to[LC], the probability of escaping a container is two times higher than of escaping a hypervisor. This is the trade-off that comes with simplified deployment and performance gain.

2.2 Docker Overview

Docker is a widespread containerisation tool with a constantly growing ecosystem and integration with multiple services. It is useful for different purposes including e.g. cloud computing virtualization, server consolidation and sand-boxing of untrusted code execution. Because of its popularity and widespread use, Docker has also become a target for various attacks. For example, the recent vulnerability CVE-2019-5736 allowed an attacker to gain root access in the container. A lot of hosts with an exposed remote Docker API were attacked and the attacker turned the machines into a botnet [noa]. The bug in the

runc library, that resulted in this vulnerability was patched in the next version of the library. The hosts with the vulnerable library version should have upgraded their library package.
Since Docker is often used in collaboration with various components like image repositories and orchestration tools, the attack surface is getting wider and Docker security properties are getting more important. Although Docker can run Windows and Linux containers, this thesis is focused on a Linux x86_64 Docker installation. This section provides a brief overview of the Docker architecture and its main components as in a Figure 2.2.

1. **Docker images** are built from Dockerfiles. Each Dockerfile contains a base image (usually a distribution of an OS) and consecutive commands, which instruct Docker how to modify the system in order to get it to the right configuration. Each command generates a new image, on which the next command will be applied. In such manner, developers can automatically build and configure their applications. Similar to the classical virtual machine configuration, images can be modified while they are already running in the container, and the image state may be stored as a new image. Images can be downloaded from online repositories called registry (e.g Docker Hub) or created and uploaded to the repository in order to share or populate and deploy the service on other hosts.

   Regarding the registries, Martin et.al. mentioned, relying on other works, that images and Dockerfiles from Docker Hub are rarely updated and very often contain unpatched medium- and/or high-priority vulnerabilities [MRCDP]. This knowledge opens more opportunities for an attacker and increases a chance of a penetration by just exploiting the already known vulnerabilities. On top of that, the image registry is similar to a package manager, therefore it inherits all possible attack vectors that the package managers can have. For example, a man-in-the-middle may sit between host and registry and distribute malicious images.

2. **Docker engine** takes advantage of two kernel features - control groups *cgroups* and *namespaces*. Namespaces copies parts of the kernel into the container, which creates an illusion for the application running in the container, that it operates on its own kernel. The following namespace types are exposed to the containers: process ID, mount, IPC, user and network. *Cgroups* on its own controls and limits the amount of resources that each container may access and use. It prevents the container from consuming more resources than it is allowed to consume. The following resources are controlled by cgroups: CPU, memory, network bandwidth, disk and priority.

   Control groups and namespaces together build a big part of the container security.

The study conducted by Lin et al. [LLW+] has shown how important and, indeed, security relevant these kernel features are for containerization tools in general. The authors performed an extensive analysis of the Linux container security. They collected more than 200 publicly available exploits and vulnerabilities that could be relevant to use cases, where a container may be deployed. The main idea was to test, how a container counters and mitigates exploits in comparison to regular execution in a host OS. For example, the privilege escalation exploits, the most dangerous attacks, were mostly blocked by namespaces and croups, however 4 of 37 exploits were successful under default container configuration. Another important conclusion was, that namespaces and cgroups are most beneficial, if they are properly configured and utilized with other security features like Seccomp, Linux capabilities and MAC (Mandatory Access Control).

3. **Docker daemon** is a running service on the host which provides REST API and responds on *Docker client* requests such as related to container lifecycle, images, network etc. It handles the communication between client and engine and runs with root privileges in order to manage important assets (e.g. network IP tables) for creating a proper isolation of the processes. Since a container is an isolated running process, it exists as long as the process runs. So, when the containerized process terminates, its Docker container terminates as well.

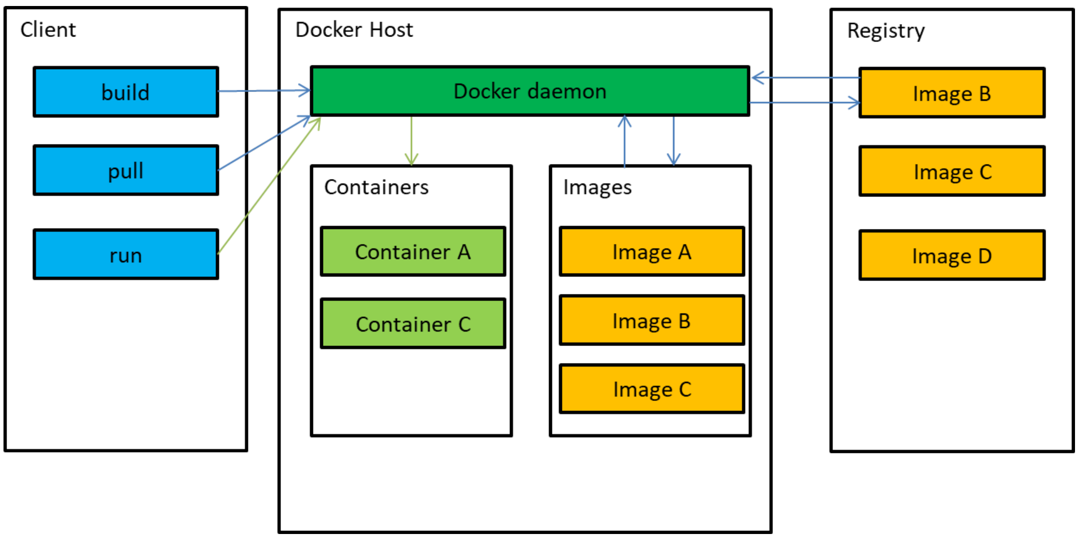


Figure 2.2: An overview of Docker components.

Docker Use Cases in Real World

There are several commercial models for Docker deployment available in the real world. One model allows the customer to have a dedicated virtual machine and to do with it whatever he wants. In our case the customer used to run own applications either local or in web. Furthermore Docker containers are utilized for CI/CD or microservice deployment. A good example of a provider with this kind of model is the Microsoft Azure platform. Customers can rent a virtual machine in a cloud on hour basis and use it for their own purposes.
Another model is a so called Docker-Hosting or Container-As-A-Service. Such hostings are specialized on providing a platform with preinstalled tools for managing applications which are built and shipped as Docker images. Using Docker-Hosting the customer can build and store Docker images (for example, for continuous integration/delivery/deployment), run automated tests, perform software/image versioning and finally launch an application in a Docker container[HMA19]. This means, that it is possible, that a container scheduler places containers of different customers on the same physical server or even in the same virtual machine and launched by the same Docker engine. For example, Amazon provides a service called 'AWS Fargate', where customers can run Docker containers without setting up any virtual machines, scaling and laborious configuration.
Another Docker usage example, where our attacks could have success, targets the simple case, when companies or private individuals own a server and run an application there. For better security, server owners can also use a hypervisor and split server resources between multiple virtual machines. From there on Docker containers can be launched in order to provide certain microservices. This way the server has been virtualized and partitioned into virtual machines with different purposes that are shielded from each other.
Docker containers may also be deployed in the industrial world. Consider a factory which has its own corporate network. In the past, the network security in the factories was not a big concern and the networks were not designed to be completely secure. Nowadays, it is common in the industry to see legacy systems with security problems. Such systems often contain different machines, devices and office computers in a single network. This means, if an attacker breaks into the corporate network, she can access every device and cause a lot of damage. This can be prevented by grouping all devices into different subnetworks of the same device type or purpose. Subnetworks may be created by connecting devices to a single node with its own virtualized network. Thereby, the number of devices that are visible to the attacker is decreased and they are represented by one network interface, building together a network hierarchy. Docker containers could help with network virtualization in the nodes and serve as a bridge between two subnetworks. Therefore, they are relevant to the industrial world and may have practical application.

2.3 Memory Deduplication

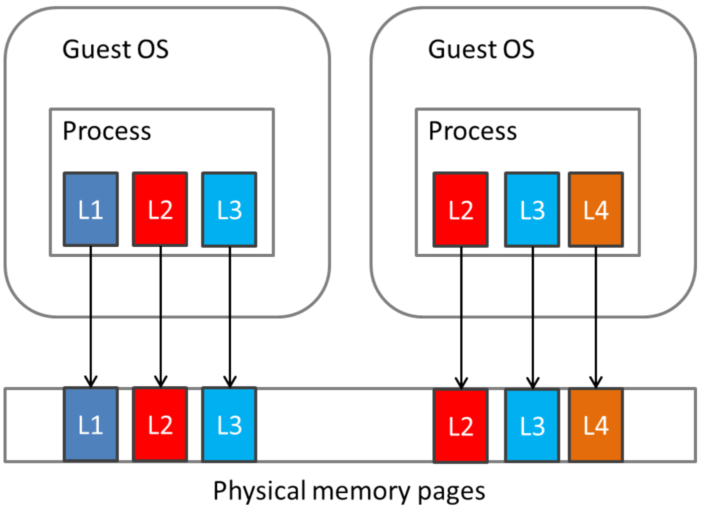
The most important factor for the presented attacks is an enabled memory deduplication mechanism in the hypervisor. In general it is a technique utilized by modern hypervisors (e.g. KVM) to reduce the memory fingerprint of the system. Running multiple virtual machines produces a high memory consumption. A remarkable property of running multiple instances of the same OS or its distribution is that the kernels of same operating systems are mostly equal, meaning there are a lot of physical memory pages with same content. So the idea is to find equal physical pages in the system and *merge* them to a single page, such that the merged page is shared between all virtual machines. The merged page is marked with a Copy-On-Write (COW) flag to insure, that the page will be explicitly copied for the VM which writes to it.

2.3.1 Kernel Same-page Merging

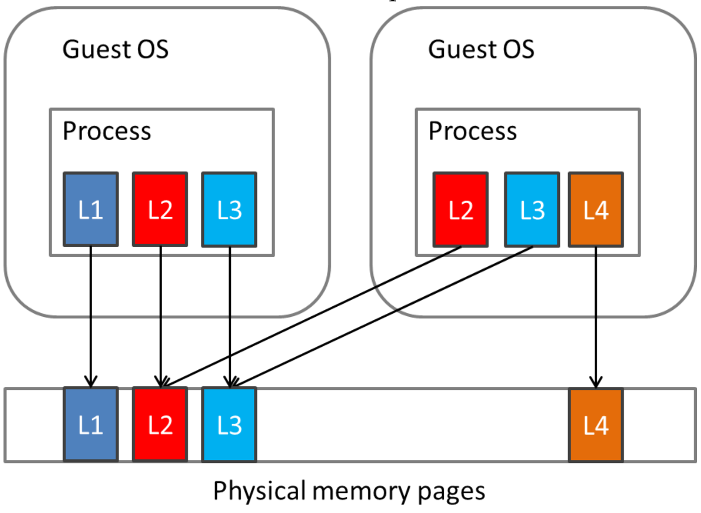
In our test system we run KVM with an enabled Kernel Same-page Merging (KSM) - memory deduplication feature utilized by KVM which comes with Lunix kernel since version 2.6.32 [AEW09]. If KSM is enabled, it periodically goes through certain pages and computes a hash value over the page content. If some pages have the same hash value (meaning the same page content), they will be merged and only one copy of the page will be shared. However, KSM scans pages that are marked with a special flag. Internally KVM marks all memory pages, that QEMU requests for a VM, with a special flag. This is done via *madvise* call. On Linux one can call *madvise* with a `MADV_MERGEABLE` flag and a pointer to a page and in such manner any page can be merged, if there are other equal pages like shown in Figure 2.3. In contrast to the Windows memory deduplication feature, KSM scans a fixed number of pages every given interval. The scanning interval, the number of pages to scan and some other settings can be configured either manually or by *ksmtuned* tool, which, if enabled, adjusts these parameters depending on the current load of the system.

2.3.2 Memory Deduplication Side-channel

At first glance memory deduplication is a neat and useful feature, but it opens another timing side-channel. And the reason for this is a Copy-On-Write mechanism. It is triggered, whenever any of the virtual machines writes to the deduplicated page. In this case the target page should be copied in a new physical page, taking certain amount of time. So, this time difference between writing into a deduplicated and a non-deduplicated page can be noticed by an attacker. If writing into the page took more time than some predefined threshold, then an attacker can deduce that, the page was copied on write and hence present in the system.



(a) Before deduplication



(b) After deduplication

Figure 2.3: Memory deduplication example. Equal pages L2 and L3 are merged and references got updated.

3 Related Work

Lindemann et al. [LF] presented a novel timing-based side-channel attack for identifying software versions running in co-resident VMs. First, the unique page (signature) should be identified for each version of a software. Then the signature is written to the system memory and after a memory deduplication pass happens, a spy process overwrites the signature page while measuring the time needed to write. If writing process has lasted longer than a threshold, the COW is detected which means a certain software version is present in the system. Multiple measurements help to reduce the noise and to increase the probability of detection. The attack was tested on a machine with KVM and with enabled KSM, where victim and spy VMs were running.
Bosman et al. [BRBG] demonstrated, that the memory deduplication side-channel can be used for a more advanced attack. The authors developed 3 attack primitives that allowed them to leak secret information byte for byte. By knowing the data around the secret the attacker can craft some probing pages by concatenating known data before the secret with different last bytes. In this way, the attacker can guess first bytes of the secret by changing last bytes of the probing pages. If one of those probing pages got deduplicated, the attacker guessed the first bytes of the secret correctly. The whole secret can be disclosed incrementally part for part. They used these primitives for leaking a randomized heap pointer and having that performed a rowhammer attack. As a proof of concept, the attack was implemented in Javascript and took place in the Microsoft Edge browser. The used attack techniques may be reimplemented in the native code and utilized read an arbitrary process data.
Irazoqui et al. [IIES15] were able to reliably detect various cryptographic libraries and even their versions from co-resident virtual machines. For this they utilized a *Flush+Reload*, a cache side-channel technique. But *Flush+Reload* on itself is not sufficient, because in order to detect the presence of certain data in another process/VM via *Flush+Reload*, the data need to be shared between the victim and the attacker. Therefore the attack requires a memory deduplication to be enabled in the system. The libraries can be detected, when they are deduplicated by a memory deduplication mechanisms which can be enabled for better memory performance gain, because then only single copy of the library resides in the system and is shared between multiple virtual machines. So now, when a certain library is shared and, hence, present in a system, its presence can be detected by flushing libraries signature functions that are always or usually called, when

any process wants to access cryptographic functions of the library. After that, the signature functions are reloaded and the time needed to reload is measured. If the measured time is shorter than a test threshold, then this means, that the presence of the library is successfully detected. For detecting a specific library version, unique addresses across all versions should be defined. By flushing and reloading version specific addresses, the specific library version can be determined. Furthermore, the authors were able to determine an IP address of the co-located virtual machine. This was done by sending TLS communication requests to all possible IP addresses in a local network and detecting via *Flush+Reload*, whether the TLS handling function was triggered.
There is also a paper [SIYA11] by Suzaki et al. which discusses fingerprinting and software detection in Linux and Windows virtual machines. It provides an overview of existing memory deduplication mechanisms, describes attack methods and shares encountered challenges. The authors were able to detect *sshd* and *apache2* in Linux VM. They achieved this by loading an ELF binary of respective executables in the RAM of a neighbor VM and measured the time needed to overwrite the complete file. Afterwards they compared the times before and after executable invocation and could see significant differences in time. Similarly to Linux they proceeded with Windows, where they observed a similar behavior by detecting running instances of Mozilla Firefox and Microsoft IE. Throughout the experiments the authors noticed, that even though ASLR changes certain pages of mapped library, there is still enough material, by which a software can be identified. Also, the page cache of the guest OS was causing false positives, such that loaded executables should be gzipped in order to not always be deduplicated. The attack was also reproducible in Javascript and an attacker could detect previously downloaded images and thus conclude, which web sites were visited by a victim.

4 Attacker Model and Environment

Based on the presented use cases of Docker from previous chapter, we specify an attacker model which suits scenarios for the real world. We also define capabilities of an attacker and her positioning in a system. Given this an attacker wants to achieve certain goals, in particular fingerprint a system and extract sensible information.

4.1 Attacker Model

Here we introduce an attacker model and environments, in which all attacks of this paper will be tested and simulated. Generally an introduction includes a description of the attacker: her positioning, what she tries to achieve and what she is capable of; and the victim which operates under certain circumstances. The victim and the attacker are basically operating on the same physical server, which is virtualized by a hypervisor. Furthermore the victim possesses a virtual machine, where a Docker multi-container application is running. Memory deduplication mechanism is available in the hypervisor and turned on.
Now we define, what a Docker multi-container application actually is. First of all Docker engine is up and running. There are multiple containers registered in the Docker engine and are potentially running. Some virtual networks are preconfigured and can contain a subset of available containers. Some of those containers have access to the outside host network (e.g. Internet, A1 container in Figure 4.1), some of them operate locally within an isolated network communicating with some other containers and providing local services for other containers within the same network (B containers in Figure 4.1). Containers may also share the same logical volume on the host machine and perform an inter process communication in such manner (shared volume SV in Figure 4.1).
In all our scenarios the victim has a static position in the system and the attacker's targets are containers located in the victims virtual machine. However, there are some variations for the attackers positioning. At this point the setting branches off into several use cases, including real world models described above and some variations of it. Hence the attacks described in this paper may be applied to all of the following setups:

1. First model from the previous section: the attacker locates in a Docker container which is running in a co-resident virtual machine and she has full control of the container with an ability to execute arbitrary code. In such setup she could get into

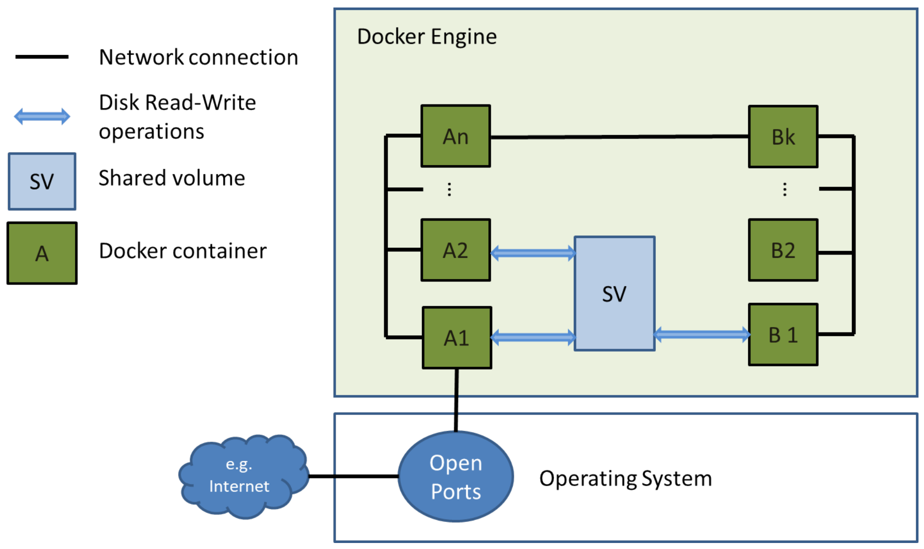


Figure 4.1: A Docker multi-container application.

the container due to an exploitable vulnerability in an application logic or vulnerable versions of used software or libraries. This setup is schematically depicted in Figure 4.2a.

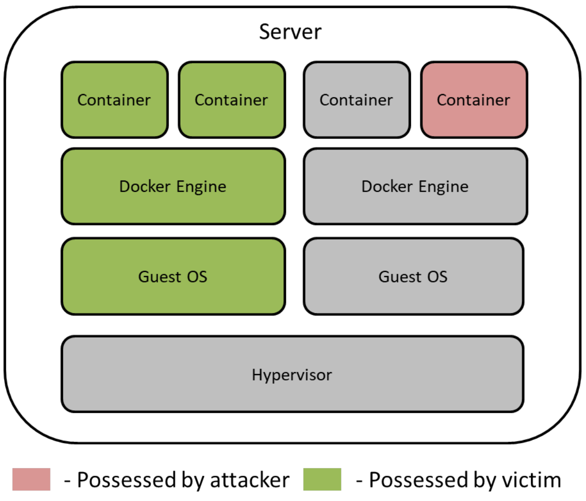
2. Second model from the previous section: here the attacker is located directly in a co-resident virtual machine. Practically this could be a rogue cloud provider, since he has control over physical server and hypervisor and can create new virtual machines for himself. Alternatively this could be a virtual machine from another customer of the platform which has bad intentions regarding a neighbor virtual machines. Anyway, from there she can execute own code or launch an attack. This setup is schematically depicted in Figure 4.2b.

3. Similar to the first setup, the attacker is located in a Docker container which is a part of the victims application. This might be a result of a successful penetration into the victims application such that she can execute arbitrary code. Hence now she has an access to some resources of the application including shared volumes and networks. She can perform better reconnaissance using network scanners like *nmap* and even read or modify contents in shared volumes. This setup is schematically depicted in Figure 4.2c.

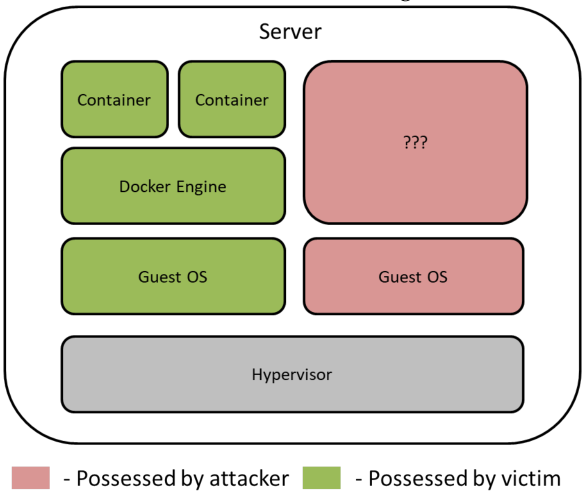
4.2 Attackers Objectives

Having described above setups, we can define attackers objectives. As a part of this thesis the attacker wants to perform a fingerprinting attack on a Docker multi-container appli-

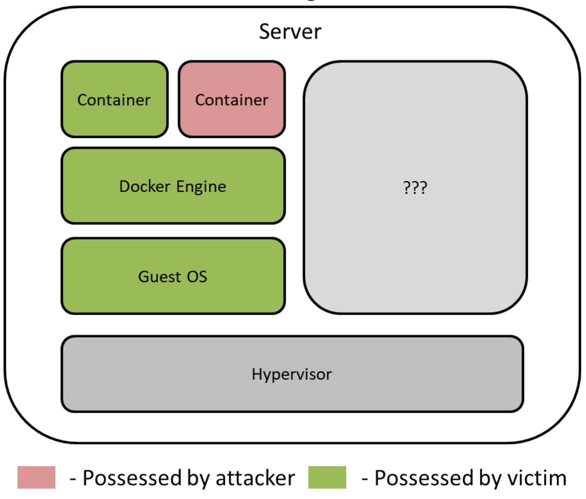
cation. The goal here is to determine, what software applications and security critical libraries are in use and present across containers and/or on the host system. Good targets would be cryptographic libraries, server software etc. and, of course, their versions. If the attacker gets more information about the victim system during the scan phase, then having versions of the running software she can prepare exploits of already known vulnerabilities for a particular application. In this work the main tool for performing fingerprinting attack is a memory deduplication side-channel. This side-channel has been already utilized by Lindemann et al. [LF] to fingerprinting a system from within a virtual machine. The were able to extract information from a co-resident VM about the Apache web server, SSH daemon and even their versions by detecting a unique software signature bytes in the memory. The experimental setup included several guest VMs, that were controlled and running in Linux Kernel-based Virtual Machine (native hypervisor) with enabled KSM feature. The approach used in that paper can be applied and reimplemented for a Docker container setup.
So, after fingerprinting, the attacker knows the targets for an attack and is interested in breaking encrypted data or retrieving sensible information. She knows, that for example cryptographic libraries may store some secrets in the RAM. That is why another topic of this thesis is secret extraction from within a Docker container. Again, the memory deduplication side-channel will be used for a leaking a secret from a neighbor container. However, this time the attack is more targeted than the fingerprinting and requires precise leakage of bytes from generic places. There is a paper by Bosman et al. [BRBG] which targets the problem of retrieving concrete bytes from a memory page with a secret via memory deduplication. The paper provides some novel techniques to incrementally brute force the memory by creating pages with different bytes in the specific places of the page. The authors could successfully launch their attack within a browser and retrieve a password hash from the *nginx* server. The idea for this work is to reproduce these techniques in the Docker environment.



(a) In the container in the neighbor VM



(b) In the neighbor VM



(c) In the neighbor container

Figure 4.2: Attackers positioning in a Docker multi-container application.

5 Fingerprinting a System

There are some state of the art methods to approach a system fingerprinting problem via memory deduplication. One of this utilizes a microarchitectural timing side channel in order to determine, if a particular function was loaded into the last level CPU cache shared between guest OSs by a victim[IIES15]. We however concentrate us on another approach. It includes generating signatures for each software version an attacker wants to detect. The attack uses a deduplication oracle to ascertain the presence of a particular software in the system. This chapter describes the used approach more detailed. We explain, what kind of preparation an attacker needs for a successful attack. The preparation consists basically of determining the KSM pass duration and of generating software signatures. Futhermore we discuss key steps of the attack itself. And finally we show some limitations and downsides of the attack.

5.1 Preparation

Before fingerprinting the system, the attacker needs to do some preparation that is basically divided into two steps. The first step includes finding out the interval KSM needs to go through all marked as mergeable pages and, if needed, to deduplicate them. If KSM is enabled, then an attacker definitely will be able to detect memory deduplication and after finding out the KSM pass interval can move on to the next step, otherwise KSM is *probably* not utilized by a host system. Within the first step the attacker can determine the time required by KSM to deduplicate a single page. The second step would be generating signatures for the versions that are likely used by the victim container. Here is a concrete description of the steps to take:

5.1.1 Determining KSM Pass Duration

In order to determine the duration of a KSM pass, the attacker can write a simple test program for it. The problem here is that the attacker does not know the exact KSM pass duration, but there is a kind of oracle that gives an information, whether a deduplication happened or not. In our case the oracle is a primitive that measures the time to overwrite a single byte in a page that is supposed to be deduplicated. For this the attacker allocates a page (an aligned memory buffer of 4096 bytes) with, for example, `posix_memalign(void **memptr, size_t alignment, size_t size)` C function which aligns the buffer depending on the `alignment` argument. If `alignment` is a page size (4096 bytes), then `memptr` will point to the page aligned buffer which is essentially the start of a page. So, in such manner the attacker allocates two pages, generates some random data and copies it into two allocated pages. This is done in order to minimize a probability of an occasional deduplication with a page that is not under attackers control, because then it is very unlikely, that there exists another page in the system with the same 4096 random bytes. So now, in order to detect a deduplication, the attacker waits for a certain time and after that measures the time needed to overwrite a single byte of any allocated page. If deduplication and, hence, COW have happened, the measured time will be significantly bigger (around 20 000 - 30 000 nanoseconds) than an usual write operation to a memory location (around 30 - 300 nanoseconds).

Now it is about to find out the time, which the attacker needs to wait for KSM to happen. Normally KSM spends from 20 seconds to 15 minutes to scan all pages depending on the system load. The difficulty here is to determine the interval ideally accurate to one second and as fast as possible. For fingerprinting it is not so important to know the exact interval, because the attacker only needs a single deduplication pass in order to detect presence or absence of a bunch of pages in the system, if she loads all needed signatures at once. However it can be crucial, if multiple measurements should be performed and in this case the deviation from the true interval will be multiplied and added to the overall attack duration. This may become a problem, if the attacker is limited in time. So, it is worth to determine right at the beginning the true interval or the interval with least deviation. The Algorithm 1 finds out the KSM pass duration using described above primitive `testDeduplication` which returns true, if deduplication has occurred, otherwise false. The algorithm is similar to exponential search. The main idea is to exponentially increase the waiting interval until deduplication got finally detected. In this case the algorithm resets an exponent and continues to increase the interval from the previous interval in which deduplication did not happened. The algorithm executes as long as the waiting interval is 1 second longer than the actual KSM pass duration. So by running this algorithm the attacker finds out the minimal KSM pass time.

However the algorithm has a bit of limitation in itself. It will not terminate, if KSM is not enabled, because `testDeduplication` will never return true and, thus, the algorithm will never reach the `break` statement. On the opposite side, the algorithm will terminate at some point, if KSM is enabled and utilized by the hypervisor. There are some other ways to find out, whether KSM is enabled, for example, by checking out the presence of '1' in the file */sys/kernel/mm/ksm/run* on the host OS. But the attacker does not have access to the host OS from a Docker container. So she cannot know for sure, if the KSM service is running. What she can do, is to prevent the *currentInterval* variable from exceeding certain upper bound, such that if *currentInterval* exceeds a upper bound, the program will terminate. The upper bound can be set to some experience value, for example 15 minutes. For systems in the real world 15 minutes is a realistic assumption even for some bigger systems. In such manner the algorithm will terminate, if KSM is disabled in the system.

Algorithm 1: Determining KSM pass duration

```
currentInterval = 0
exponent = 1
oldInterval = 0
while true do
    wait(currentInterval)
    if testDeduplication() then
        if currentInterval - oldInterval == 1 then
            break
        exponent = 1
        currentInterval = oldInterval
    oldInterval = currentInterval
    currentInterval += exponent
    exponent *= 2
return currentInterval
```

5.1.2 Generating Signatures

As the KSM pass duration was found out, the attacker needs other important component in order to proceed with the actual fingerprinting attack. For each library or software the attacker wants to detect, she should generate an unique signature of each version she expects to find in the victim system. For this we followed the instructions from Lindemann et al. paper[LF]. The signatures are generated from binary files without any execution. Any executable or library in Linux are built in a specific way called Executable and Linkable Format (ELF). Any ELF binary consists of executable headers, program headers, sections and section headers. The only part of the ELF binary that resides in the physical memory after being loaded are sections. So, the pages of the sections are the material for creating a signature. The utility from the paper takes different binary versions of a given software and parses respective headers in order to extract only loadable sections from the ELF binary. Having the sections the tool divides them into 4KB pages and creates an unique signature from them. Firstly internal duplicates and pages consisting of zeroes or ones are removed from the binary. Afterwards the pages that exist in the other versions are taken out as well. In the end the rest of the ELF binary contains only pages that are unique across all sections and is considered as a final version signature. Now the attacker can move on to the actual attack.

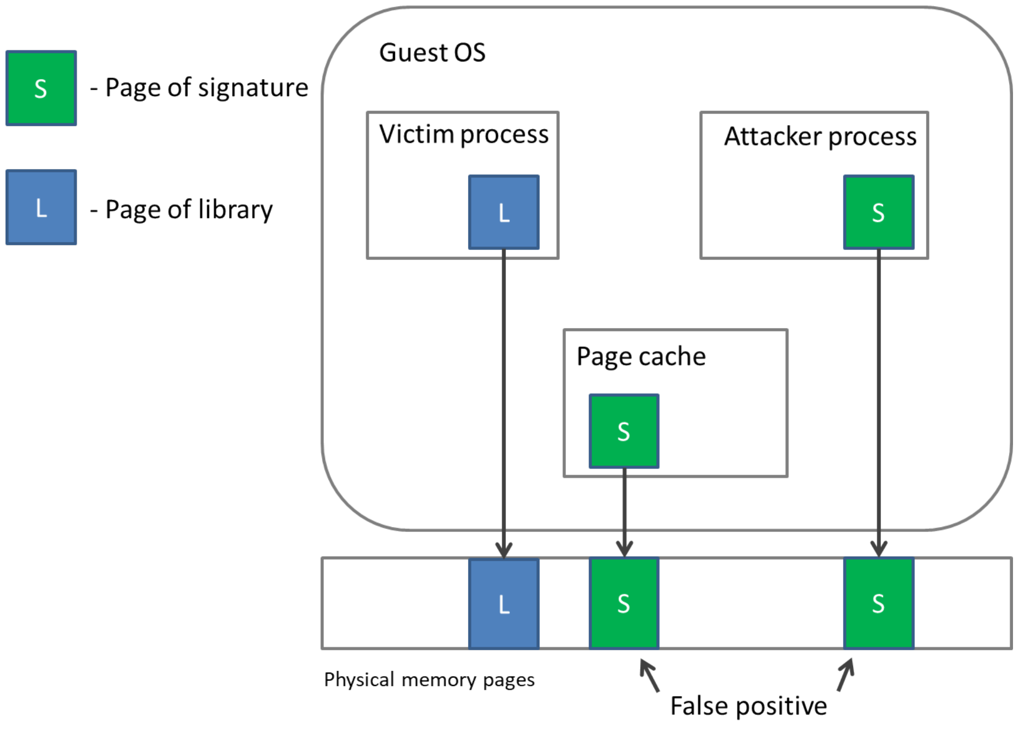
5.2 Attack Description

A flow diagram of detecting a single signature is presented in Figure 5.2. The attacker is located in a Docker container and can execute own arbitrary code. What she also needs are the signatures. They can be transmitted to the occupied container either within a binary exploit program or as separate files. If the signatures are stored in files, the exploit program loads all available signatures via *posix_memalign* into page aligned buffers similar as the test page from the first preparation step. Otherwise it copies signatures, that are embedded into binary file as resources, to the page aligned buffers. After doing this the program sleeps for the time of KSM deduplication pass that was determined during the preparation phase. When the KSM deduplication pass happened, the program starts a timer and overwrites a **single** byte in each page of the buffer with the signature, meaning overwriting the byte in the buffers at index 0, 4096, 8192 and so on. Overwriting a single byte of the page is still efficient, because it triggers the COW mechanism and is an improvement comparing to the measurement program from Lindemann et al. paper[LF], where they overwrite the complete signature. This optimization leads to smaller measurement times and lower noise, since the measurement time does not include the time needed for overwriting another 4095 bytes of a single page. After the signature got overwritten, the program stops the timer and the measurement time is calculated. Before exiting the program all allocated buffers with signatures are zeroed out. Now the attacker can decide, whether the parts of the signature were deduplicated by comparing the time with a predetermined threshold. The attacker can generate a file with random data of size same as size of the signature and perform a measurement on it. The measurement time of the random file is a threshold for the signature. In order to get better results of the attack, multiple measurements can be performed.

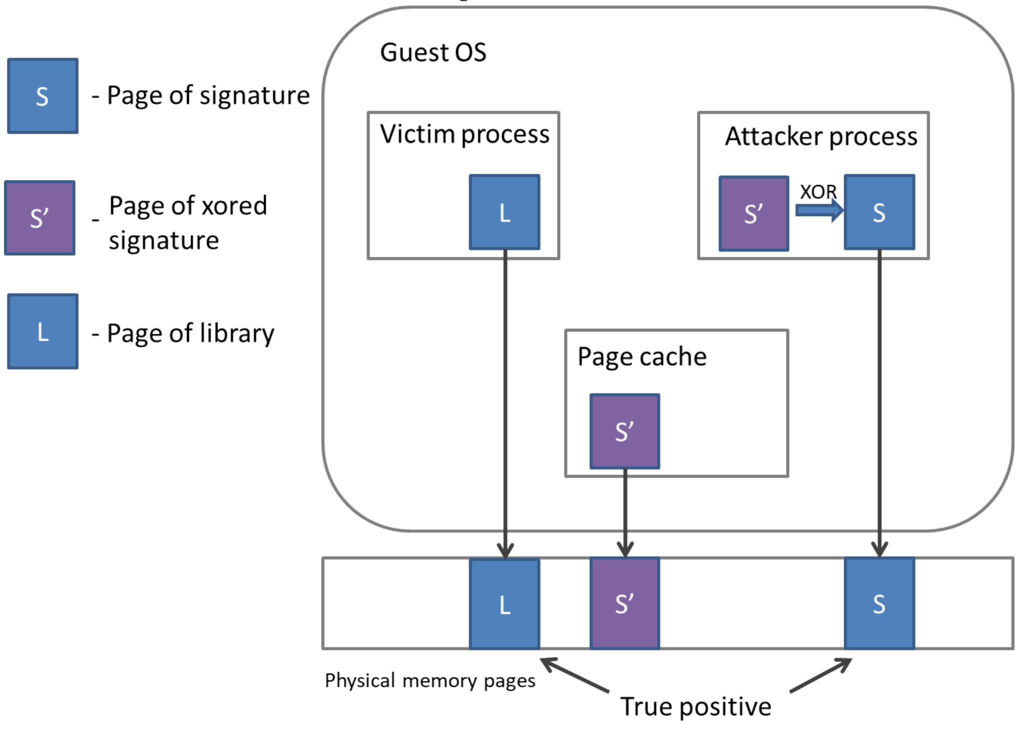
5.3 Limitations

This naive attack approach has some downsides. Even if the signature was zeroed out in the attacker process memory and therefore cannot be deduplicated in the next measurement run, a copy of the signature file may still reside in a **page cache** of a guest OS, as also noticed in [SIYA11]. A page cache is a standard feature of any modern OS. It caches content of a file from a secondary storage like HDD or SSD. So it is like a cache between an OS and physical storage implemented on a software level. Even if the system is not using a certain file, this file may still reside in RAM with an idea, that some file content can be requested soon. This feature enhances performance of a file system and leads to faster file access time, if the requested file was already in the page cache. For our attack this means, that next measurements will deliver false positives for all signatures of versions that indeed have never been in the system, just because the page aligned buffer allocated by our exploit program will be deduplicated with a page cache version of itself (green pages in Figure 5.1a). However, this issue is easy to obviate. Similar to the runtime packer technique used in [SIYA11], one can obfuscate the signature content by, for example, xoring every byte with a fixed value. And while loading the signatures into the memory, the signature bytes can be xored with the same fixed value back, resulting in a true signature (blue pages in Figure 5.1b). So, the true signature resides in the attacker process memory while waiting for a deduplication pass and after the measurement has finished, the obfuscated pages of the signature are in the page cache instead of the true signature pages. In such manner false positives caused by the deduplication of pages from page cache can be avoided.

The page cache causes another problem that also results in false positives. Consider a situation, when the victim starts any program, then its file gets cached by the page cache and parts of it, such as code and data, are loaded into the memory for further execution. Finally the victim terminates the process and the pages loaded for the program may now be freed and reused by another process. However the programs page cache version is still present in the memory and can be deduplicated, if the attacker makes a measurement again. This would produce a false positive, just because the targeted software **was in use some time ago** and the page cache would trigger a positive result. So the attacker learned something that is not true and will assume, the victim currently runs a particular software, although the software is not running anymore. Despite that the fingerprinting attack is still reliable in a Docker multi-container setting. Docker containers are service-oriented processes, meaning containers are more likely to be restarted, such that a certain service can still be provided and a software will continue to persist in the memory.



(a) Regular situation



(b) With obfuscated signature

Figure 5.1: How page cache results in detection of wrong pages and a solution for this problem.

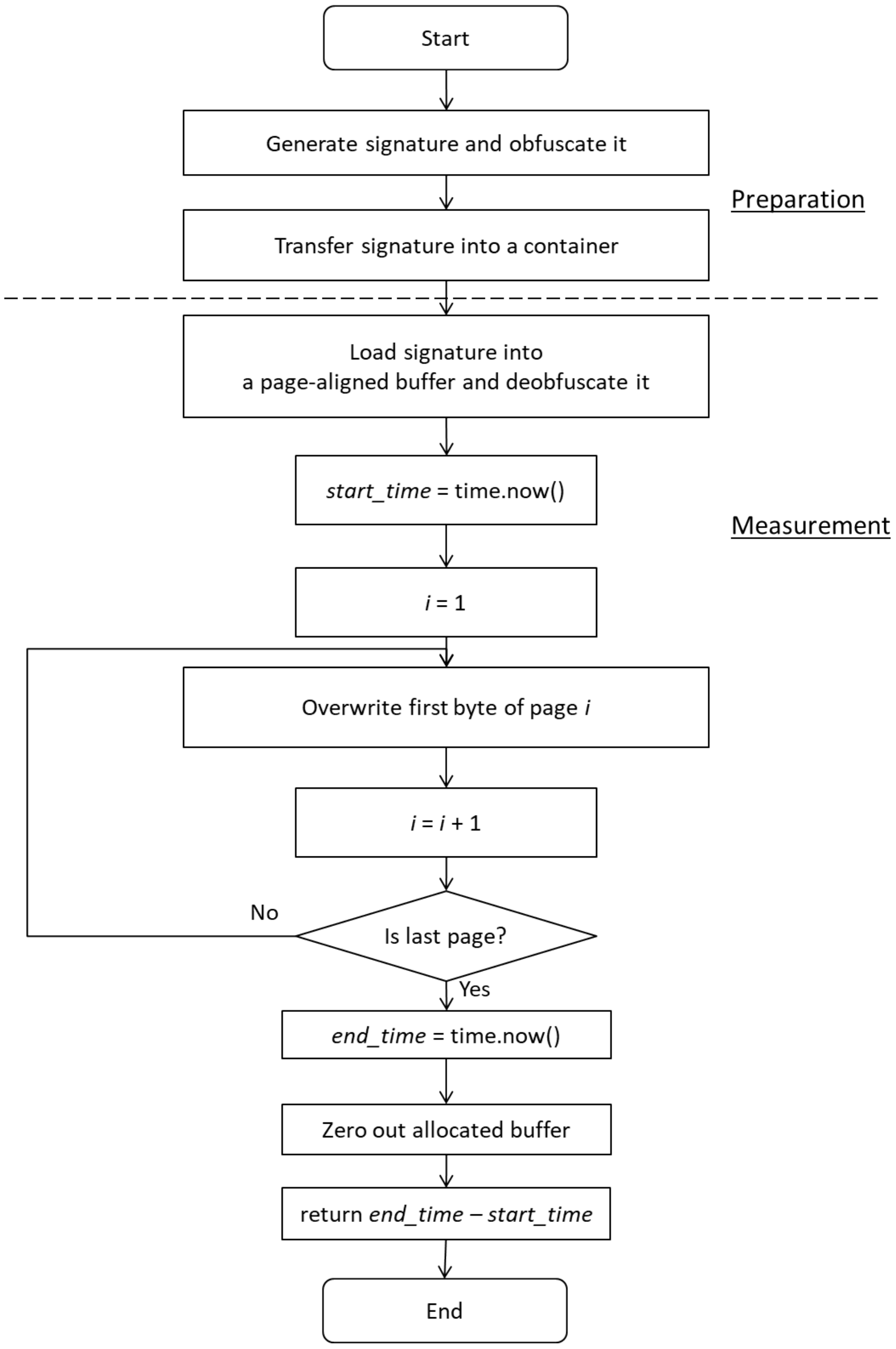

Start
Generate signature and obfuscate it
Transfer signature into a container
Preparation
Load signature into a page-aligned buffer and deobfuscate it
start_time = time.now()
Measurement
i = 1
Overwrite first byte of page i
i = i + 1
Is last page?
No
Yes
end_time = time.now()
Zero out allocated buffer
return end_time – start_time
End

Figure 5.2: Flow diagram of the fingerprinting attack with one signature: preparation and measurement stages.

6 Secret Extraction

Another objective of this thesis is to show, that Docker containers are also vulnerable to memory disclosure attacks based on the memory deduplication side-channel. This is done by attacking a vulnerable example application. Here describe we the properties, that a vulnerable application should have in order to be successfully attacked. Later on we explain the attacker primitives, that are utilized by an adversary for constructing a working exploit. As a reference we take the paper by Bosman et al. [BRBG], where its authors provided useful techniques and a comprehensive description of a system-wide exploitation. Furthermore we show an example attack which serves as a prove for the fact, that Docker containers are unprotected against memory deduplication side-channel.

6.1 Properties of an Exploitable Application

Not any secret in an application can become a target for this kind of secret extraction attack. To disclose a secret its application should have certain characteristics and memory layout. In order to determine if an application is exploitable, an attacker should manually inspect target software either by finding certain consistent memory behavior in a memory image of a process or by looking into a source code and inserting some debug statements, in case an attacker has an access to the sources of the software. One can claim an application and a certain secret as vulnerable to memory disclosure attacks based on memory deduplication side-channel, if an application and target secret satisfy following conditions:

- There exists an interface in the application, such that the attacker can enter her own data bytes. The amount of data the attacker can feed in is at least 8KB. The attacker needs this constraint to be fulfilled in order to be able to control a memory layout and be aware of the way her data is being laid out in the memory.. Also after the data was received and written in the memory, it should remain unmodified for the time of one memory deduplication pass or if it is modified, the attacker should know, how the modification is done and be able to reproduce the modification process by herself. This is required in the brute force stage of the attack for the ability to build brute force pages with data controlled (known) by the attacker.
- Another condition is the placement of the secret which the attacker wants to dis-

close. Essentially the secret should be located after the attackers input and ideally with none or small offset. This implies indirectly, that with this property the attacker gains an ability to move the secret back and forth by reducing or increasing the amount of input data. Therefore it allows the attacker to launch an alignment probing primitive discussed later. Because then the attacker can enter such amount of data, that first bytes are attacker controlled and the few remaining bytes are unknown.

- An optional condition that may simplify the attack is a static or computable offset of attackers data relative to page beginning. In this case, if the attacker knows, that her input data is always going to be located at the same page offset, then it becomes relatively easy to calculate the amount of input data. Since the attacker aims to place a page with known data in the victim application and consequently tries to brute force unknown bytes in the end of the page, she should craft a page with known data and **prepend** to it the amount of bytes needed to fill the space between data beginning at the known offset and the page ending. This condition is optional, because without having this property the attacker can still generate desired memory layout by using another attack primitive, which however makes an exploit consume more resources and, thus, a bit more complex.

If a target application and a secret fulfill the above constraints, it becomes possible for the attacker to apply attack primitives and build an exploit. Combined all together these conditions are sufficient to leak secret bytes in a particular software. Now it is time to introduce the attack primitives that are essential for constructing an exploit.

6.2 Attack Primitives

Following techniques that were used in the work of Bosman et al. [BRBG] can be applied on a vulnerable application that has the properties described above. They also allow an attacker to control, predict and generate a certain memory layout. Controlling the memory layout makes it possible for an attacker to craft probing pages with proper page content and brute force a secret byte for byte. In a context of memory deduplication, brute forcing a certain page means to put in the attackers memory the pages with all possible combinations of **unknown** bytes. Then these pages will be tested, whether one of them got deduplicated. If so, then it means, the page is present in the system and an attacker can determine the values of bytes previously unknown. However, it does not make sense for an attacker to brute force whole pages, because there are too many possible combinations of bytes in a single page. Therefore it is easier and more practical to brute force pages that consist of a lot of known data and a small part of unknown data and doing this incrementally. It is also not always guaranteed, that the page, which an attacker wants to brute force, begins exactly at a page boundary. The following primitives were designed to address these problems and create a reliable way to disclose parts of page content:

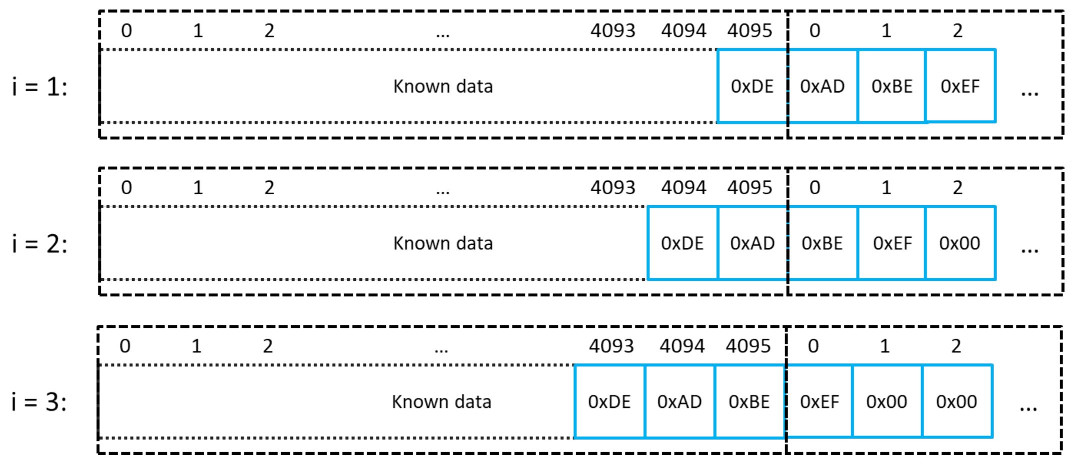
- *Alignment probing*(Figure 6.1a): this is a primitive that actually allows incremental brute forcing. The idea is to place a page in the victims memory, that a first part consists of the known dummy bytes and a second part consists of the secret bytes. Given such memory layout an attacker can brute force the rest of unknown bytes which indeed are part of the secret. The amount of secret bytes should be feasible for brute forcing a page within an acceptable amount of time or with an acceptable amount of pages. Now, when an attacker knows a small part of the secret, she can put a page with a smaller amount (amount is reduced by amount of known secret bytes) of dummy data into the victims memory, such that the next unknown part of the secret lies at the end the page with known dummy data. Brute force pages are adjusted respectively, such that the last bytes of the page will be brute forced and again, they are also placed again in the attackers memory and tested against being deduplicated. Another part of the secret is disclosed and the attack goes on, until complete secret is revealed. Using this strategy an attacker can incrementally disclose bytes of the secret.
- *Partial reuse*(Figure 6.1b): this technique is useful for disclosing small secrets (e.g. randomized heap pointer) that can be brute forced with two memory deduplication passes. If during an initial analysis of the target software memory image an attacker notices, that memory from last interaction is reused for a next interaction and the secret is located in the end of the page, then she can overwrite the first part of the secret with her own known data and brute force the remaining part. In the next iteration, when the previous situation happens again, she brute forces the first part, because this part is surrounded by known data (her own data and then second part of the secret).
- *Heap spray*: this primitive is used in cases, where the alignment of attackers input data is unpredictable. The idea is to flood the victim application with input data, such that afterwards the victims heap consists of multiple pages with attacker provided data. By flooding enough data pages it is possible to achieve target alignment meaning, that attackers input data starts at the page beginning and the secret is located after this data. Although it is not guaranteed that the attackers data will not be overwritten, in practice only a small part of the memory area with the attackers data is reused for other purposes due to reuse patterns of standard memory allocator

`malloc`[BRBG] and, thus, a target alignment is retained.

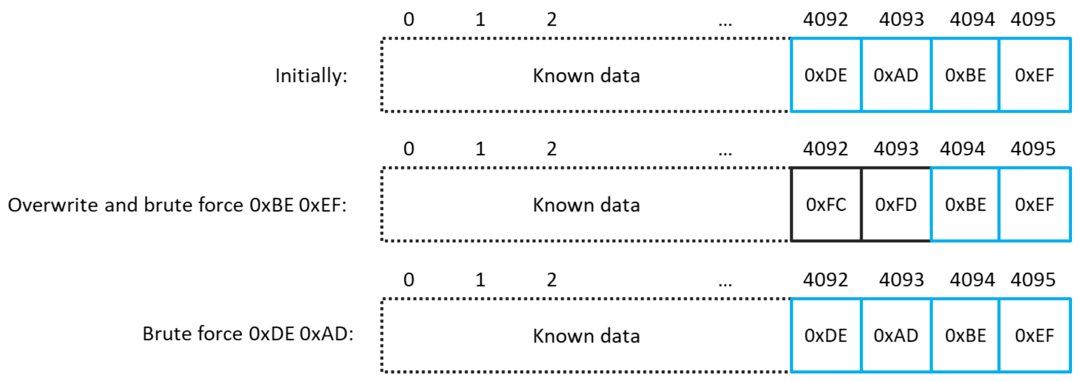
The reason of, why the above primitives can be successfully executed, is mainly the memory allocator used by the software, especially a custom one. Custom allocators usually tend to not zero out the freed memory. Due to a custom allocation strategy, the allocator can generate memory reuse patterns. An attacker can find out the used allocation strategy. This facts allow an attacker to predict a placement of her input data. In such manner an attacker can utilize the allocators behavior for attackers favor.

6.3 Attacking Example Application

In order to show, that an application inside a Docker container is still vulnerable to memory disclosure attacks via the memory deduplication side-channel, we implement an attack on a vulnerable example application. For this we created a simple server-like program with described above properties. One can connect to the server and provide an authentication string which serves as a password. The password itself consists of ASCII letters and numbers and is null-terminated. A protocol that the server speaks with a client is line based, meaning each message ends with a line-break. If a client sends the correct string, the server responds with th 'OK' string, otherwise with the 'FAILED' string. A buffer for the input data is page aligned. The server stores a password needed for the authentication in a file that is loaded and put into memory directly after clients input data. The example application allows an attacker to choose alignment probing as a basis for the attack and it is essential for brute forcing the password byte for byte by shifting the secret in the memory.
The attack starts by sending 4095 bytes of input to the victim application. The input is then page aligned and the last byte of the page is the first byte of the server password, because the password is placed directly after the input data. The password character set consists of 62 characters, so the attacker prepares 62 brute force pages that differ by last byte and waits for KSM to pass. By testing, whether one of the brute force pages was deduplicated, the attacker determines the first byte of the secret. In the next iteration the attacker sends 4094 bytes to the server. Now, when the attacker knows the first byte of the secret, she can guess the next one by setting the 4095-th byte of brute force pages to known byte. These steps are repeated until the whole secret is revealed meaning a zero byte is guessed.



(a) First three iterations of alignment probing example



(b) Example of partial reuse

Figure 6.1: Attacking primitives for secret extraction.

7 Experiment Results and Mitigations

The attacks we described before were implemented and tested in different setups and configurations. The system configurations differ by the location of an attacker as it was stated in the chapter 4 about the attacker model. We also provide a description of our experimental setup for different attack scenarios. Furthermore we present our results of both executed attacks. After that we discuss and propose mitigations to this attack vector.

7.1 Experimental Setup

The following experimental setup was taken as a reference system. The test machine has an Intel Core i3-5010U CPU at 2.10GHz and 4GB of DDR3 RAM. As host operating system we chose Ubuntu 18.04. Since Ubuntu is a Linux system, its kernel already contains the KVM module, which we also utilize in our experiments as a bare-metal hypervisor. KSM is enabled and scans 1000 pages every 100 milliseconds. With QEMU version 2.11.1 we created two virtual machines with a single CPU core and 1GB RAM assigned to them. Afterwards Ubuntu Server 19.04 with Docker Engine version 18.09.7 was installed into both virtual machines. For both attacks, fingerprinting and secret extraction, we launch in a guest OS a victim container with an application, we want to attack. In case of fingerprinting we pull an already created image with the latest (version 2.4.41) Apache server from Docker Hub and start a Docker container for it. For the secret extraction attack we launch an example exploitable application from the previous chapter in a Docker container. Overall we tested our attacks with different attacker positions. We refer to this positions as following:

- Type 1 (Figure 4.2a) position is where the attacker locates in a container in a neighbor guest OS.
- Type 2 (Figure 4.2b) position is where the attacker locates in a neighbor guest OS.
- Type 3 (Figure 4.2c) position is where the attacker locates in a neighbor container in the same guest OS.

These types are sorted by level of separation. So type 1 provides maximum separation, since the victim and the attacker are separated by virtual machine and Docker container boundaries. And type 3 provides the least separation, since the Docker containers are basically processes within the same OS.

7.2 Fingerprinting Attack Results

The ttackers objective is to detect the Apache Web server latest version 2.4.41 and distinguish it from version 2.4.37 which is not used at all and therefore not present in the victim system. For this we generate signatures of respective versions and do XOR operations with fixed value on each byte of the signatures. All needed files, including signatures and an attacking program, were transferred into all types of positions. The attacking program loads signatures in its process space and does XOR on each signature byte with the fixed value that was used before. After waiting for 40 seconds, the first bytes of all signature pages were overwritten and the time was measured. The attack was executed and **succeeded** in every position regardless of the level of separation. That means, we were able to differentiate between two versions of the Apache Web server. More precisely the times to overwrite a signature of version 2.4.41 were around 100 ms against 3 ms for version 2.4.37. The difference between both times was big enough, such that we could easily distinguish the versions.

7.3 Secret Extraction Attack Results

The idea of this attack was to extract a correct authentication string that is used in the example victim application. Similar to the fingerprinting attack we place out attacking program into all types of positions. However, now we need, that attacker's and victim's applications are in the same network. So we configure virtual machines and Docker containers, such that attacker's and victim's applications may communicate with each other through a common network. In our attacking program we implemented the alignment probing primitive. We attack a victim by sending to the victim application a correctly calculated amount bytes and crafting proper pages for a brute force. As a result we uccessfully managed to exfiltrate a seven characters long authentication string from the victim container. Overall the attack took 320 seconds and we used 248KB of RAM.

7.4 Mitigations

There are several strategies for mitigating a memory deduplication attack vector. They vary between completely disabling the attackers ability to perform any attack based on memory deduplication and choosing performance over security. So it is always a tradeoff between security and performance. The first obvious mitigation strategy suggests to completely disable memory deduplication in a system. Thus, this completely takes away the attackers ability to attack the system over this particular attack vector and none of the described attacks will have any chance for success. This strategy however removes possible memory performance improvements, which is not always acceptable, because then it requires more physical RAM for the system. Next strategy allows a deduplication of zero pages only. Such strategy keeps a certain degree of increased memory performance[BRBG] by the memory deduplication mechanism while totally limiting an attacker in performing his attacks. In this case an attacker can only detect a presence of zero pages which essentially provides no additional information to an attacker. The last strategy permits only a deduplication of read-only pages. This gives better memory performance comparing to the previous strategies, since then more pages can be potentially deduplicated. Nonetheless this solution allows an attacker to perform the fingerprinting attack. For example, a software version can be detected by its unique page of data section which is, indeed, read-only. This fact may seem undesirable, however, one can not completely prevent a system from being fingerprinted, because there are other ways to fingerprint a system, like port scanning a public server and other techniques. So this mitigation can have its own use case. Simultaneously it completely mitigates an opportunity for a secret extraction attack, since a secret on a heap cannot be shifted in memory and therefore can not be deduplicated and guessed byte for byte, because heap pages are writable. So everyone can decide which mitigation strategy to pick by weighting memory performance and security implications.

8 Conclusions

8.1 Summary

In this work we set a goal to examine the memory deduplication attack vector and its relevance in Docker multi-container applications. For this we dove into virtualization technologies, in particular hypervisor-based and container-based virtualization, in order to understand their use cases and differences. We gave an overview of a widely used containerization tool called Docker and of KSM, a memory deduplication mechanism in KVM with a side-channel that comes with it. After we defined use cases and our attacker model, we started with a fingerprinting attack description. As next important class of attacks we inspected a secret extraction attack. When we implemented and successfully executed both attacks in different attacker scenarios, we discovered, that a memory deduplication side-channel is still a threat to a system with enabled memory deduplication mechanism like KSM, even if a victim process is running in a Docker container in a different virtual machine. Considering this fact we discussed some variants of mitigations and came to the conclusion, that an answer to a question which mitigation to pick, depends on an application operator, an attacker model for this particular application and how much security risk an operator is ready to attend.

8.2 Discussion and Open Problems

The current approach of finding software that is vulnerable to a secret extraction attack is laborious and requires manual inspection of memory images and, in some cases, reading source code. So the next question that a reader might ask himself is, how can an adversary generically build an exploit for secret extraction in any application. Maybe there are certain types of software which are especially vulnerable to memory deduplication attacks. A more advanced question would be, whether there is a way to automatically find vulnerable software. Implementation, practical evaluation or improvement of suggested mitigation strategies can also be seen as good next topics, because many cloud providers tend to disable memory deduplication, even if this feature is useful and can save a good portion of physical RAM. Another open topic would be to investigate, how to do a secret extraction via `FLUSH+RELOAD` technique and memory deduplication, since there is already a fingerprinting attack that utilizes both features.

References

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. 01 2009.

[BBB$^+$] Salman Baset, Stefan Berger, James Bottomley, Canturk Isci, Nataraj Nagaratnam, Dimitrios Pendarakis, J. R. Rao, Gosia Steinder, and Jayashree Ramanatham. Docker and container security white paper.

[BRBG] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. page 18.

[HMA19] Mohamed K. Hussein, Mohamed H. Mousa, and Mohamed A. Alqarni. A placement architecture for a container as a service (caas) in a cloud environment. *Journal of Cloud Computing*, 8(1):7, May 2019.

[IIES15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know thy neighbor: Crypto library detection in cloud. *PoPETs*, 2015(1):25–40, 2015.

[LC] Tao Lu and Jie Chen. Research of penetration testing technology in docker environment. In *Proceedings of the 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*. Atlantis Press.

[LF] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18*, pages 183–192. ACM Press.

[LLW$^+$] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference on - ACSAC '18*, pages 418–429. ACM Press.

[mic] Microservices. https://en.wikipedia.org/wiki/Microservices. Accessed: 2019-09-27.

[MRCDP] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem – vulnerability analysis. 122:30–43.

[noa] Docker API vulnerability allows hackers to mine monero. `https://www.scmagazine.com/home/security-news/vulnerabilities/docker-api-vulnerability-allows-hackers-to-mine-monero/`. Accessed: 2019-09-27.

[Pat] Randy Patterson. Running docker windows and linux containers simultaneously. `https://devblogs.microsoft.com/premier-developer/running-docker-windows-and-linux-containers-simultaneously/`. Accessed: 2019-09-27.

[PZW$^+$07] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. HP Laboratories, 2007.

[SIYA11] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.