Hypervisor- vs. Container-based Virtualization

Michael Eder
Betreuer: Holger Kinkelin
Seminar Future Internet WS2015/16
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: edermi@in.tum.de

ABSTRACT

For a long time, the term *virtualization* implied talking about hypervisor-based virtualization. However, in the past few years container-based virtualization got mature and especially Docker gained a lot of attention. Hypervisor-based virtualization provides strong isolation of a complete operating system whereas container-based virtualization strives to isolate processes from other processes at little resource costs. In this paper, hypervisor and container-based virtualization are differentiated and the mechanisms behind Docker and LXC are described. The way from a simple chroot over a container framework to a ready to use container management solution is shown and a look on the security of containers in general is taken. This paper gives an overview of the two different virtualization approaches and their advantages and disadvantages.

Keywords

virtualization, Docker, Linux containers, LXC, hypervisor, security

1. INTRODUCTION

The paper compares two different virtualization approaches, hypervisor and container-based virtualization. Container-based virtualization got popular when Docker [1], a free tool to create, manage and distribute containers gained a lot of attention by combining different technologies to a powerful virtualization software. By contrast, hypervisor-based virtualization is the alpha male of virtualization that is widely used and around for decades. Both technologies have advantages over each other and both come with tradeoffs that have to be taken into account before deciding which of the both technologies better fits the own needs. The paper introduces hypervisor- and container-based virtualization in Section 2, describes advantages and disadvantages in Section 3 and goes deeper into container-based virtualization and the technologies behind in Section 4. There is already a lot of literature about hypervisor-based virtualization whereas container-based virtualization started to get popular in the last few years and there are fewer papers about this topic around, so the main focus of this paper is container-based virtualization. Another focus of the paper is Docker which is introduced in Section 4.3, because it traversed a rapid development over the last two years and gained a lot of attention in the community. To round the paper out, general security risks of container-based virtualization and Docker and possible ways to deal with them are elaborated in Section 5. Section 6 gives a brief overview to related work on this topic.

2. DISTINCTION: HYPERVISOR VS. CONTAINER-BASED VIRTUALIZATION

When talking about virtualization, the technology most people refer to is hypervisor-based virtualization. The hypervisor is a software allowing the abstraction from the hardware. Every piece of hardware required for running software has to be emulated by the hypervisor. Because there is an emulation of the complete hardware of a computer, talking about *virtual machines* or *virtual computers* is usual. It is common to be able to access real hardware through an interface provided by the hypervisor, for example in order to access files on a physical device like a CD or a flash drive or to communicate with the network. Inside this virtual computer, an operating system and software can be installed and used like on any normal computer. The hardware running the hypervisor is called the *host* (and the operating system *host operating system*) whereas all emulated machines running inside them are referred to as *guests* and their operating systems are called *guest operating systems*. Nowadays, it is usual to get also utility software together with the hypervisor that allows convenient access to all of the hypervisor's functions. This improves the ease of operation and may bring additional functionalities that are not exactly part of the hypervisor's job, for example snapshot functionalities and graphical interfaces. It is possible to differentiate two types of hypervisors, type 1 and type 2 hypervisors. Type 1 hypervisors are running directly on hardware (hence often referred to as *bare metal hypervisors*) not requiring an operating system and having their own drivers whereas type 2 hypervisors require a host operating system whose capabilities are used in order to perform their operations. Well-known hypervisors or virtualization products are:

- KVM [2], a kernel module for the Linux kernel allowing access to virtualization capabilities of real hardware and the emulator usually used with KVM, qemu [3], which is emulating the rest of the hardware (type 2),
- Xen [4], a free hypervisor running directly on hardware (type 1),
- Hyper-V [5], a hypervisor from Microsoft integrated into various Windows versions (type 1),
- VMware Workstation [6], a proprietary virtualization software from VMware (type 2)
- Virtual Box [7], a free, platform independent virtualization solution from Oracle (type 2).

From now on, this paper assumes talking about type 2 hypervisors when talking about hypervisors.
Container-based virtualization does not emulate an entire computer. An operating system is providing certain features to the container software in order to isolate processes from other processes and containers. Because the Linux kernel provides a lot of capabilities to isolate processes, it is required by all solutions this paper is dealing with. Other operating systems may provide similar mechanisms, for example FreeBSD's jails [8] or Solaris Zones. Because there is no full emulation of hardware, software running in containers has to be compatible with the host system's kernel and CPU architecture. A very descriptive picture for container-based virtualization is that "containers are like firewalls between processes" [9]. A similar metaphor for hypervisor-based virtualization are processes running on different machines connected to and supervised by a central instance, the hypervisor.

3. USE CASES AND GOALS OF BOTH VIRTUALIZATION TECHNOLOGIES

Hypervisor and container-based virtualization technologies come with different tradeoffs, hence there are different goals each want to achieve. There are also different use cases for virtualization in general and both hypervisor and container-based virtualization have therefore special strengths and weaknesses relating to specific use cases. Because of abstracting from hardware, both types of virtualization are easy to migrate and allow a better resource utilization, leading to lower costs and saving energy.

3.1 Hypervisor-based virtualization

Hypervisor-based virtualization allows to fully emulate another computer, therefore it is possible to emulate other types of devices (for example a smartphone), other CPU architectures or other operating systems. This is useful for example when developing applications for mobile platforms — the developer can test his application on his development system without the need of physically having access to a target device. Another common use case is to have virtual machines with other guest operating systems than the host. Some users need special software that does not run on their preferred operating system, virtualization allows to run nearly every required environment and software in this environment independently from the host system. Because of the abstraction from the hardware, it is easier to create, deploy and maintain images of the system. In the case of hardware incidents, a virtual machine can be moved to another system with very little effort, in the best case even without the user noticing that there was a migration. Hypervisor-based virtualization takes advantage of modern CPU capabilities. This allows the virtual machine and its applications to directly access the CPU in an unprivileged mode [10], resulting in performance improvements without sacrificing the security of the host system.

Hypervisors may set up on the hardware directly (type 1) or on a host operating system (type 2). Figure 1 shows a scheme where the hypervisor is located in this hierarchy. Assuming a type 1 Hypervisor, all operating systems were guests in Figure 1 whereas a type 2 hypervisor was on the same level than other userspace applications, having the operating system (not shown in the figure) and the real hardware on the layers below it.

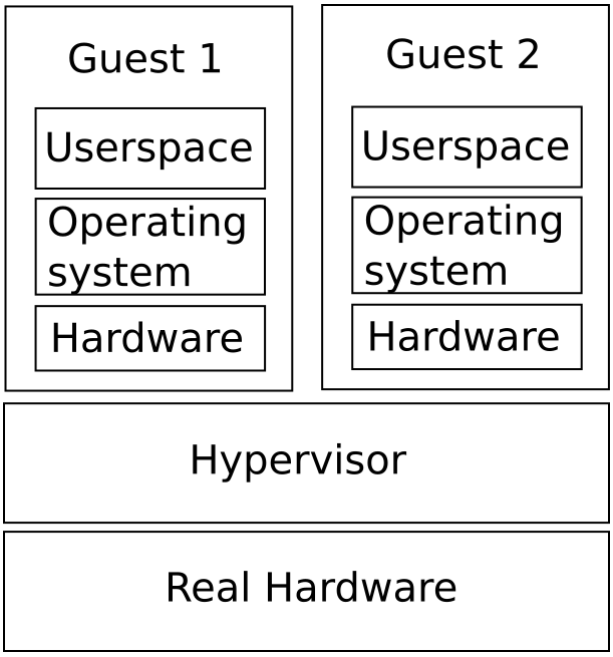

Figure 1: Scheme of hypervisor-based virtualization. Hardware available to guests is usually emulated.

3.2 Container-based virtualization

Container-based virtualization utilizes kernel features to create an isolated environment for processes. In contrast to hypervisor-based virtualization, containers do not get their own virtualized hardware but use the hardware of the host system. Therefore, software running in containers does directly communicate with the host kernel and has to be able to run on the operating system (see figure 2) and CPU architecture the host is running on. Not having to emulate hardware and boot a complete operating system enables containers to start in a few milliseconds and be more efficient than classical virtual machines. Container images are usually smaller than virtual machine images because container images do not need to contain a complete toolchain to run a operating system, i.e. device drivers, kernel or the init system. This is one of the reasons why container-based virtualization got more popular over the last few years. The small resource fingerprint allows better performance on a small and larger scale and there are still relatively strong security benefits.

Shipping containers is a really interesting approach in software developing: Developers do not have to set up their machines by hand, instead of that a build system image may be distributed containing all tools and configuration required for working on a project. If an update is required, only one image has to be regenerated and distributed. This approach eases the management of different projects on a developers machine because one can avoid dependency conflicts and separate different projects in an easy way.

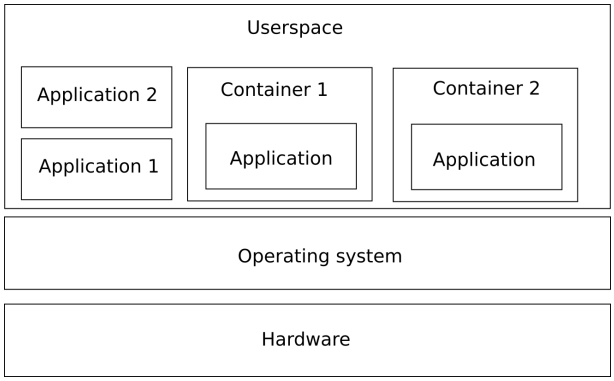

Figure 2: Scheme of container-based virtualization. Hardware and Kernel of the host are used, containers are running in userspace separated from each other.

4. FROM CHROOT OVER CONTAINERS TO DOCKER

Container-based virtualization uses a lot of capabilities the kernel provides in order to isolate processes or to achieve other goals that are helpful for this purpose, too. Most solutions build upon the Linux kernel and because the paper's focus lies on LXC and Docker, a closer look at the features provided by the Linux kernel in order to virtualize applications in containers will be taken. Because the kernel provides most of the capabilities required, container toolkits usually do not have to implement them again and the kernel code is already used by other software that is considered stable and already in production use. In case of security problems of the mechanisms provided by the kernel, fixes are getting distributed with kernel updates, meaning that the container software does not need to be patched and the user only has to keep its kernel up to date.

4.1 chroot

The **ch**ange **root** mechanism allows to change the root directory of a process and all of its subprocesses. Chroot is used in order to restrict filesystem access to a single folder which is treated as the root folder (/) by the target process and its subprocesses. On Linux, this is not considered a security feature because it is possible for a user to escape the chroot [11].

Apart from that, chroot may prevent erroneous software from accessing files outside of the chroot and — even if it is possible to escape the chroot — it makes it harder for attackers to get access to the complete filesystem. Chroot is often used for testing software in a somehow isolated environment and to install or repair the system. This means chroot does not provide any further process isolation apart from changing the root directory of a process to a different directory somewhere in the filesystem.

Compared to a normal execution of processes, putting them into a chroot is a rather weak guarantee that they are not able to access places of the filesystem they are not supposed to access.

4.2 Linux containers

Linux containers [12] (LXC) is a project that aims to build a toolkit to isolate processes from other processes. As said, chroot was not developed as a security feature and it is possible to escape the chroot — LXC tries to create environments (*containers*) that are supposed to be escape-proof for a process and its child processes and to protect the system even if an attacker manages to escape the container. Apart from that, it provides an interface to fine-grained limit resource consumption of containers. Containers fully virtualize network interfaces and make sure that kernel interfaces may only be accessed in secure ways. The following subsections are going to show the most important isolation mechanisms in greater detail. The name *Linux containers* (*LXC*) may be a little bit confusing: It is a toolkit to create containers on Linux, of course there are other possibilities to achieve the same goal with other utilities than LXC and to create isolated processes or groups of them (*containers*) under Linux. Because LXC is a popular toolkit, most of the statements of this paper referring to *Linux containers* apply to LXC, too, but we are going to use the term *LXC* in order to specifically talk about the popular implementation and *Linux containers* in order to talk about the general concept of process isolation as described in this paper on Linux. The capabilities introduced in the following are difficult to use meaning that proper configuration requires a lot of reading documentation and much time setting everything up. Deploying complex configuration of system-level tools is a hard task. Easily adopting the configuration to other environments and different needs without endangering security of the existing solution is nearly impossible without a toolkit providing access to those features. LXC is a toolkit trying to fulfill these requirements.

4.2.1 Linux kernel namespaces

By now, processes are only chrooted but still see other processes and are able to see information like user and group IDs, access the hostname and communicate with others. The goal is to create an environment for a process that allows him access to a special copy of this information that does not need to be the same that other processes see, but the easy approach of preventing the process to access this information may crash it or lead to wrong behaviour. The kernel provides a feature called namespaces (in fact, there are several different [13] namespaces) in order to realize these demands. Kernel namespaces is a feature allowing to isolate processes, groups of processes and even complete subsystems like the interprocess communication or the network subsystem of the kernel. It is a flexible mechanism to separate processes from each other by creating different namespaces for processes that need to be separated. The kernel allows passing global resources such as network devices or process/user/group IDs into namespaces and manages synchronization of those resources. It is possible to create namespaces containing processes that have a process ID (PID) that is already in use on the host system or other containers. This simplifies migration of a suspended container because the PIDs of the container are independent from the PIDs of the host system. User namespaces allow to "isolate security-related identifiers and attributes, in particular, user IDs and group IDs [...], the root directory, keys [...], and capabilities" [14]. Combining all these features makes it possible to isolate processes in a way that

- process IDs inside namespaces are isolated from process IDs of other namespaces and are unique per-namespace, but processes in different namespaces may have the same process IDs,
- global resources are accessible via an API provided by the kernel if desired,
- there is abstraction from users, groups and other security related information of the host system and the containers.

In fact, kernel namespaces are the foundation of process separation and therefore one of the key concepts for implementing container-based virtualization. They provide a lot of features required for building containers and are available since kernel 2.6.26 which means they are around for several years and are used in production already [15].

4.2.2 Control groups

Control groups (further referred to as cgroups) are a feature that is not mandatory in order to isolate processes from other processes. In the first place cgroups is a mechanism to track processes and process groups including forked processes. [16] In the first instance, they do not solve a problem that is related to process isolation or making the isolation stronger, but provided hooks allow other subsystems to extend those functionalities and to implement fine-grained resource control and limitation which is more flexible compared to other tools trying to achieve a similar goal. The ability to assign resources to processes and process groups and manage those assignments allows to plan and control the use of containers without unrestricted waste of physical resources by a simple container. The same way it is possible to guarantee that resources are not unavailable because other processes are claiming them for themselves. At first this might look like a feature primarily targeting at hosts serving multiple different users (e.g. shared web-hosting), but it is also a powerful mechanism to avoid Denial of Service (DoS) attacks. DoS attacks do not compromise the system, they try to generate a useless workload preventing a service to fulfill its task by overstressing it. A container behaving different than suspected may have an unknown bug, be attacked or even controlled by an attacker and consuming a lot of physical resources — having limited access to those resources from the very beginning avoids outages and may safe time, money and nerves of all those involved. As said, controlling resources is not a feature required for isolation but essential in order to compete with hypervisor-based virtualization. It is very common to restrict resource usage in hypervisor-based virtualization and being able to do the same with containers allows to better utilize the given resources.

4.2.3 Mandatory Access Control

On Linux (and Unix), everything is a file and every file has related information about which user and which group the file belongs to and what the owner, the owning group and everybody else on the system is allowed to do with a file. *Doing* in this context means: reading, writing or executing. This *Discretionary Access Control* (*DAC*) decides if access to a resource is granted solely on information about users and groups. Contrary, *Mandatory Access Control* (*MAC*) does not decide on those characteristics. Mandatory Access Control is a set of concepts defining management of access control which is gaining more and more attention over the past years. The need for such systems is generally to improve security and in this case MAC is used to harden the access control mechanisms of Linux/Unix. Instead, if a resource is requested, authorization rules (*policies*) are checked and if the requirements defined by the policy are met access is granted. There are different approaches on how to implement such a system, the three big ones are namely SELinux [17], AppArmor [18] and grsecurity's RBAC [19]. MAC policies are often used in order to restrict access to resources that are sensitive and not required to be accessed in a certain context. For example the *chsh*[1] userspace utility on Linux has the setuid bit set. This means that a regular user is allowed to run this binary and the binary can elevate it self to running with root privileges in order to do what it is required to do. If a bug was found in the binary it may be possible to execute malicious code with root privileges as a normal user. A MAC policy could be provided allowing the binary to elevate its rights in order to do its legitimate job but denying everything root can do but the library does not need to do. In this case, there is no use case for the chsh binary to do networking but it is allowed to because it can do anything root can do. A policy denying access to network related system components does not affect the binary but if it was exploited by an attacker, he wouldn't be able to use the binary in order to sniff traffic on active network interfaces or change the default gateway to his own box capturing all the packets because a MAC policy is denying the chsh tool to access the network subsystem. There are a lot of examples where bugs in software let attackers access sensitive information or do malicious activities that are not required by the attacked process, e.g. CVE–2007–3304[2] which is a bug in the Apache webserver allowing an attacker to send signals to other processes which gets mitigated already by enabling SELinux in enforcing mode [10]. Especially network services like web- or mail servers, but also connected systems like database servers are receiving untrusted data and are potential attack targets. Restricting their access to resources to their minimal requirements increases the security of the system and all connected systems. Applying these security improvements to containers adds another layer of protection to mitigate attacks against the host and other containers from inside of the container.

4.3 Docker

LXC is a toolkit to work with containers saving users from having to work with low level mechanisms. It creates an interface to access all the features the Linux kernel is providing while reducing the learning curve for users. Without LXC, a user needs to spend a lot of time to read the kernel documentation in order to understand and use the provided features to set up a container by hand. LXC allows to auto-

[1] chsh allows a user to change its login shell. This is a task every user is allowed to do on its own but requires modification of the file /etc/passwd which is accessible for root only. chsh allows safe access to the file and allows a user to change his own login shell, but not the login shells of other users

[2] Common Vulnerabilities and Exposures is a standard to assign public known vulnerabilities a unique number in order to make it easy and to talk about a certain vulnerability and to avoid misunderstandings by confusing different vulnerabilities

mate the management of containers and and enables users to build their own container solution that is customizable and fits even exotic needs. For those who want to simply run processes in an isolated environment without spending too much time figuring out how the toolkit works, this is still impractical. This is were Docker [1] comes in: It is a command line tool that allows to create, manage and deploy containers. It requires very little technical background compared to using LXC for the same task and provides various features that are essential to work with containers in a production environment. Docker started in March 2013 and got a lot of attention since then, resulting in a rapid growth of the project. By combining technologies to isolate processes with different other useful tools, Docker makes it easier to create container images based on other container images. It allows the user to access a public repository on the internet called *Docker Hub* [20] that contains hundreds of ready usable images that can be downloaded and started with a single command. Users have the ability to change and extend those images and share their improvements with others. There is also an API allowing the user to interact with Docker remotely. Docker brings a lot of improvements compared to the usage of LXC, but introduces also new attack vectors on a layer that is not that technical anymore because now users and their behaviour play a bigger part in the security of the complete system. Before talking about that in greater detail, some major improvements over LXC Docker introduced are outlined.

4.3.1 One tool to rule them all

LXC is a powerful tool enabling a wide range of setups and because of that, it is hard to come by when not being deep into the topic and the tools. Docker aims to simplify the workflow. The Docker command line tool is the interface for the user to interact with the Docker daemon. It is a single command with memorizable parameter names allowing the user to access all the features. Depending on the environment, there is little to no configuration required to pull (download) images from the online repository (Docker Hub) and run it on the system. Apart from not being required, it is of course possible to create a configuration file for Docker and specific images in order to ease administration, set specific parameters and to automate the build process of new images.

4.3.2 Filesystem layers generated and distributed independently

One major improvement over classical hypervisor-based virtualization Docker introduced is the usage of filesystem layers. Filesystem layers can be thought of as different parts of a filesystem, i.e. the root file system of a container and filesystems containing application specific files. There may be different filesystem layers, for example one for the base system which may be always required and additional layers containing the files of e.g. a webserver. This means that the webserver that is ready to run can be distributed as an independent filesystem layer. By way of illustration a web service is consisting of a SQL database server like MySQL, a webserver like Apache, a programming language and the framework a web application is built on like python using Django and a mail transfer agent like sendmail is assumed. Of course it is possible to a certain extent to split those tools and run them in different environments (mail, web and database may run on completely different machines), this example assumes that the setup is the developers setup and differs from the deployment in production use. So, after installing Docker, it is possible to fetch the filesystem layers of all of those tools and combine them to one image that runs all the software presented above and may now be used in order to develop the web app. The web app itself may now be the content of a new filesystem layer allowing to be deployed in the same manner later on. If a new version of one of the components is released and the developer wants to update its local layer, only the updated layer has to be fetched. The configuration of the user usually remains in a separate layer and it is not affected by the update. Because there is no need to fetch the data for other layers again, space and bandwidth is saved. If another user wants to use the web app the developer pushed to the Docker Hub, but he prefers PostgreSQL over MySQL and nginx over Apache webserver and the web app is capable of being used with those alternatives to the developer's setup, he may use the respective filesystem layers of the software he wants to use.

4.3.3 Docker Hub

As said, one of the novelties Docker introduced to the virtualization market was the Docker Hub [20]. It is one of the things nobody demanded because it was simply not there and everybody was fine reading documentation and creating virtual machines and container images from scratch over and over again. The Docker Hub is a web service that is fully integrated into the Docker software and allows to fetch images that are ready to run from the Docker Hub. Images created or modified by users may be shared on the Docker Hub, too. The result is that the Docker Hub contains a lot of images of different software in different configurations and every user has the ability to fetch the images and documentation of the images, use and improve them and get in touch with other users by commenting on images and collaborating on them. The Docker Hub can be accessed via a web browser, too. To ease the choice of the *right* images of often used images like Wordpress, MongoDB and node.js, the Docker Hub allows such projects to mark their images as *official* images that come directly from declared project members. The ability for everyone to push their images to the Docker Hub led to a great diversity and a lot of software available as container images. Even more complex or very specific configurations of some programs can be found there. The Hub introduces also some security problems that are covered in Section 5. Users are not tied to the Docker Hub. It is possible to set up private registries that can be used natively the same way, but are e.g. only accessible for authenticated users in a company network or in order to build a public structure that is independent to the already existing structure.

4.4 Managing great numbers of containers

Running containers is cheap regarding resource consumption. Building large-scale services relying on container structures may involve thousands of containers located on different machines all over the world. Today this may still be a corner case, but some companies are already facing management problems because they have too much containers to administrate them by hand. Google introduced Kubernetes [21] in order to ease the administration, group containers and automate working with groups of containers.

5. SECURITY OF CONTAINER-BASED VS. HYPERVISOR-BASED VIRTUALIZATION

Hypervisor-based virtual machines are often used in order to introduce another layer of security by isolating resources exposed to attackers from other resources that need to be protected. "Properly configured containers have a security profile that is slightly more secure than multiple applications on a single server and slightly less secure than KVM virtual machines." [10] The better security profile compared to multiple applications next to each other is because of the already mentioned mechanisms making it harder to escape from an isolated environment, but all processes in all containers still run together with other processes under the host kernel. Because KVM is a hypervisor emulating complete hardware and another operating system inside, it is considered even harder to escape this environment resulting in a slightly higher security of hypervisor-based virtualization.

It is possible to combine both technologies and due to the small resource fingerprint and the introduced security layer of container-based virtualization, this is considered a good idea. Nevertheless, container-based virtualization is not proven to be secure and container breakouts already happened in the past [22].

Multiple security issues arise from the spirit of sharing images. Docker has for a long time not had sufficient mechanisms to check the integrity and authenticity of the downloaded images, meaning that the authors of the image are indeed the people who claim they are and that the image has not been modified without being recognized, to the contrary their system exposed attack surface and it may has been possible for attackers to modify images and pass Docker's verification mechanisms [23]. Docker addressed this issues by integrating a new mechanism called *Content Trust* that allows users to verify the publisher of images [24]. There is nothing like the Docker Hub for hypervisor-based virtualization software, meaning that it is usual to build the virtual machines from scratch. In case of Linux, on the one hand it is common that packages are signed and the authenticity and integrity can be validated, on the other hand each virtual machine contains a lot of software that needs to be taken care off by hand all the time.

One of the biggest problems of Docker is that it is hard to come to grips with the high amount of images containing security vulnerabilities. Origin of this problem is that images on the Docker Hub do not get updated automatically, every update has to be built by hand instead. This is a rather social problem of people not updating their software neither updating the containers they pushed to the Docker Hub in the past. This is a somehow harder problem compared to the same phenomenon of software not being up to date on a virtual machine: There is actually no update available on the Docker Hub, the latest image contains vulnerabilities. On virtual machines when running an operating system that is still taken care of, someone needs to log in and install the security updates, which may be forgotten or simply ignored because of various reasons, but building the fixed container as a user is usually way more effort.

A study [25] found out that "Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities". The numbers were generated by a tool scanning the images listed on the Docker Hub for known vulnerabilities. More than 60% of official images, which means they come from official developers of the projects contained in the container, contained medium or high priority vulnerabilities. Analyzing the vulnerabilities of all images on the Hub showed that 75% contained vulnerabilities considered medium or high priority issues. Overcoming these problems requires permanent analysis of the containers which means scanning them for security problems regularly and inform their creators and users about problems found. This means that it may be easier to come by the problem of outdated containers because the fact that they are stored centrally allows to statically scan all images in the repository.

Mr. Hayden points out how to build a secure LXC container from scratch that can be used as foundation for further modifications [10].

6. RELATED WORK

One of the use cases for hypervisor and container-based virtualization is improving security by isolating processes that are untrusted or are connected to other systems and expose attack surface, i.e. a webserver on a machine connected to the internet. Of course, there are different ways to improve security that may be applied additionally to virtualization.

The already described Mandatory Access Control (see Section 4.2.3) goes already into the direction of hardening the operating system. This means to add new or improve already present mechanisms in order to make it harder to successfully attack systems. On Linux, the grsecurity [19] project is well-known for their patch set adding and improving a lot of kernel features, for example MAC through their RBAC system, Address Space Layout Randomization (ASLR) and a lot of other features making it harder to attack the kernel.

Another approach of separating applications can be found in the Qubes OS [26] project. They build an operating system based on Linux and the X11 window system on top of XEN and allow to create *AppVMs* that run applications in dedicated virtual machines — somehow similar to the container approach, but with hypervisor-based virtualization instead of mechanisms built inside a standard Linux kernel. According to their homepage, it is even possible to run Microsoft Windows based application virtual machines.

MirageOS [27] is also an operating system setting up on XEN, but deploying applications on MirageOS means deploying a specialized kernel containing the application. Those unikernels [28] contain only what is needed in order to perform their only task. Because there are no features not absolutely required, unikernels are usually really small and performant. MirageOS is a library operating system that may be used in order to create such unikernels. The probably biggest weakness of this approach is that applications need to be written specifically to be used with unikernels.

7. CONCLUSION

Container-based virtualization is a lightweight alternative to hypervisor-based virtualization for those, who do not require

or wish to have separate emulated hardware and operating system kernel — a system in a system. There are many scenarios where speed, simplicity and only the need to isolate processes are prevalent and container-based virtualization fulfills these needs. By using mostly features already present in the Linux kernel for several years, container-based virtualization builds on a mature codebase that is already running on the user's machines and kept up to date by the Linux community. Especially Docker introduced new concepts that were not associated and used together with virtualization. Giving users the social aspect of working together, sharing and reuse the work of other users is definitely one of Dockers recipes for success and a completely new idea in the area of virtualization. Because it is relatively easy to run a huge number of containers, there are already tools allowing to manage large groups of containers.

When it comes to security, there is no need for a "vs." in the title of this paper. Generally, it is of course possible to run containers on an already (hypervisor-) virtualized computer or to run your hypervisor in your container. The hypervisor layer is considered a thicker layer of security than the application of the mechanisms described above. Nevertheless, applying these lightweight mechanisms adds additional security at literally no resource cost.

Both approaches and of course their combination is hardware-independent, allows a better resource utilization, improves security and eases management.

8. REFERENCES

[1] *Docker project homepage*, https://www.docker.com/, Retrieved: September 13, 2015

[2] *KVM project homepage*, http://www.linux-kvm.org/page/Main_Page, Retrieved: September 16, 2015

[3] *qemu project Homepage*, http://wiki.qemu.org/Main_Page, Retrieved: September 16, 2015

[4] *XEN project homepage*, http://www.xenproject.org/, Retrieved: September 16, 2015

[5] *Microsoft TechNet Hyper-V overview*, https://technet.microsoft.com/en-us/library/hh831531.aspx, Retrieved: September 16, 2015

[6] *VMware Homepage*, http://www.vmware.com/, Retrieved: September 16, 2015

[7] *VirtualBox project homepage*, https://www.virtualbox.org/, Retrieved: September 16, 2015

[8] M. Riondato *FreeBSD Handbook, Chapter 14. Jails*, https://www.freebsd.org/doc/handbook/jails.html, Retrieved: September 26, 2015

[9] E. Windisch: *On the Security of containers*, https://medium.com/@ewindisch/on-the-security-of-containers-2c60ffe25a9e, Retrieved: August 18, 2015

[10] M. Hayden: *Securing Linux containers*, https://major.io/2015/08/14/research-paper-securing-linux-containers/, Retrieved: August 18, 2015

[11] chroot (2) manpage, release 4.02 of Linux man-pages project, http://man7.org/linux/man-pages/man2/chroot.2.html, Retrieved: September 16, 2015

[12] *LXC project homepage*, https://linuxcontainers.org/, Retrieved: September 13, 2015

[13] namespaces (7) manpage, release 4.02 of Linux man-pages project, http://man7.org/linux/man-pages/man7/namespaces.7.html, Retrieved: September 16, 2015

[14] user_namespaces (7) manpage, release 4.02 of Linux man-pages project, http://man7.org/linux/man-pages/man7/user_namespaces.7.html, Retrieved: September 16, 2015

[15] Docker Docs: *Docker Security*, https://docs.docker.com/articles/security/, Retrieved: August 18, 2015

[16] P. Menage, P. Jackson, C. Lameter: *Linux Kernel Documentation*, https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, Retrieved September 13, 2015

[17] *SELinux project homepage*, http://selinuxproject.org/page/Main_Page, Retrieved: September 26, 2015

[18] *AppArmor project homepage*, http://wiki.apparmor.net/index.php/Main_Page, Retrieved: September 26, 2015

[19] *grsecurity project homepage*, https://grsecurity.net/, Retrieved: September 16, 2015

[20] *Docker Hub*, https://hub.docker.com/, Retrieved: September 26, 2015

[21] *Kubernetes project homepage*, http://kubernetes.io/, Retrieved: September 16, 2015

[22] J. Turnbull: *Docker container Breakout Proof-of-Concept Exploit*, https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/, Retrieved: August 18, 2015

[23] J. Rudenberg: *Docker Image Insecurity*, https://titanous.com/posts/docker-insecurity, Retrieved: August 18, 2015

[24] D. Mónica: *Introducing Docker Content Trust*, https://blog.docker.com/2015/08/content-trust-docker-1-8/, Retrieved: August 18, 2015

[25] Jayanth Gummaraju, Tarun Desikan and Yoshio Turner: *Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities*, http://www.banyanops.com/blog/analyzing-docker-hub/, Retrieved: August 18, 2015

[26] *Cubes OS project homepage*, https://www.qubes-os.org/, Retrieved: September 16, 2015

[27] *MirageOS project homepage*, https://mirage.io/, Retrieved: September 16, 2015

[28] *XEN Unikernels wiki page*, http://wiki.xenproject.org/wiki/Unikernels, Retrieved: September 16, 2015