Fraunhofer
AISEC

FRAUNHOFER INSTITUTE FOR APPLIED AND INTEGRATED SECURITY

THREAT ANALYSIS OF CONTAINER-AS-A-SERVICE FOR NETWORK FUNCTION VIRTUALIZATION

Executive Summary

Network function virtualization (NFV) strives to transition hardware-based network equipment into virtualized environments. It promises to increase efficiency of current networks, reduce cost for network operators and foster innovative business models. The main component of NFV are virtualized network functions (VNFs). These components realize functionality previously bound to hardware-based network equipment in software. To maximize efficiency and utilization, VNFs require a flexible, scalable and fault-tolerant deployment and infrastructure layer.

Recently, container-based virtualization is maturing into production-grade platforms and becomes a new option that promises to fulfill the demands set forth by an NFV architecture. The concepts of loosely coupled software components and microservices matches the requirements of VNFs. However, containers pose new security challenges. The use of containers for the deployment of VNFs requires a domain-specific threat analysis.

In this report, we describe a Container-as-a-Service platform, analyze security threats and detail possible mitigations. Our analysis will show that there are a number of threats. Among them, the escape from containers is the most devastating. However, many threats can be mitigated by applying security best practices and guidelines. Moreover, the surrounding ecosystem for container virtualization provides new tools that assist in securing deployments. For example, secrets and configuration management make it feasible to distribute credentials and apply configurations efficiently and securely. At the same time, container-based virtualization provide benefits to VNFs for example by leveraging their fault-tolerance and scalability.

In conclusion, we believe that container-based virtualization can offer a suitable infrastructure for VNF deployment. As long as providers and tenants account for threats and adopt appropriate mitigations, risks can be managed. The final judgment depends on the specific use case and its security requirements.

Contents

1 Introduction **4**

2 Container-based Virtualization **6**
- 2.1 Linux Namespaces and Control Groups 6
- 2.2 Container Runtime and Images 7

3 Threats Analysis and Attack Vectors **9**
- 3.1 Container-as-a-Service Architecture for NFVI 9
- 3.2 Threat Model 10
  - 3.2.1 Actors 10
  - 3.2.2 Assets 11
  - 3.2.3 External Dependencies 11
  - 3.2.4 Entry Points 12
- 3.3 Attack Vectors 12
  - 3.3.1 Attacks against VNFs and Network Services in Containers 13
  - 3.3.2 Attacks against the Container Management 15
  - 3.3.3 Exploiting Access to the Container Management 16
  - 3.3.4 Exploiting Access to Containers 17

4 Mitigation Strategies **21**
- 4.1 Securing VNFs 21
- 4.2 Securing the Container Management 27
- 4.3 Enforcing Container Isolation 28

5 Conclusion **32**

Bibliography **32**

Fraunhofer AISEC **36**

The Authors **37**

1 Introduction

Network function virtualization (NFV) is a new paradigm to design and operate telecommunication networks [1]. Traditionally, these networks rely on dedicated hardware-based network equipment and their functions to provide communication services. However, this reliance is becoming increasingly inflexible and inefficient, especially in dealing with traffic bursts for example during large crowd events [2]. NFV strives to overcome current limitations by (1) implementing network functions in software and (2) deploying them in a virtualized environment [3]. The resulting virtualized network functions (VNFs) require a virtual infrastructure that is flexible, scalable and fault-tolerant.

The growing maturity of container-based virtualization and the introduction of production-grade container platforms promotes containers as a candidate for the implementation of NFV infrastructure (NFVI) [4]. Containers offer a simplified method of packaging and deploying applications and services. In addition, they facilitate continuous product iterations by bringing development and operations closer together. Embracing container-based virtualization and its associated paradigms, such as micro-services, creates lightweight and reusable software components. These components are becoming the new building blocks for highly scalable distributed systems.

In the wake of container-based virtualization, a large software ecosystem has developed covering aspects of provisioning, networking, orchestration, service discovery, monitoring and automation. In particular, new orchestration and management software such as Kubernetes[1], Apache Mesos[2] and Docker Engine's swarm mode[3] provide an interface to descriptively compose distributed systems. They automate the scheduling, configuring, scaling, synchronizing and repairing of distributed systems, resulting in Container-as-a-Service (CaaS). The benefits of container technologies and their ecosystem correspond well to the requirements of NFVI as well as NFV management and orchestration (NFV-MANO), in which ETSI explicitly states that virtualization can be based on hypervisors or containers [4]. Thus, data center providers or network operators could offer container-based NFVI platforms where tenants can deploy, manage and orchestrate containers hosting their VNFs in a NFVI-as-a-Service (NFVIaaS) business model.

However, the adoption of container-based virtualization for NFVI also creates new challenges. First, it is currently unclear how a suitable architecture of a container-based NFVI platform would look like. Integrated container solutions, such as Kubernetes, often promote deployments of container engines in a virtual infrastructure rather than on bare-metal systems. The combination of VMs and containers may introduce additional challenges while providing greater flexibility.

Second, new paradigms like micro-services demand the adoption of new development and architecture approaches. This may be more suitable for newly developed VNFs, than for existing

[1] https://kubernetes.io/
[2] https://mesos.apache.org/
[3] https://docs.docker.com/engine/swarm/

monolithic network services, where a re-design may take a considerable amount of time. Moreover, the micro-service paradigm with its decomposition of software into loosely coupled components creates highly dynamic distributed systems. This represents a challenge for deployment, orchestration, management and monitoring. Fortunately, the software ecosystem around containers has developed tools and products that provide a high degree of automation. However, operators and tenants must still be able to comprehend the deployed system, identify unexpected behavior, and resolve problems.

Finally, the overarching challenge of container-based NFVI platforms is the evaluation of security. A thorough understanding of the threats associated with such a platform and its ecosystem is essential for its adoption. To this end, it is important to illustrate and discuss threats against containers, the architecture of a container-based NFVI platform and the required management interfaces. From such a discussion, possible mitigation strategies can be developed.

In this paper, we therefore focus on the security implications of virtualizing network functions using containers. In Chapter 2 we give a detailed description of container-based virtualization with its inherent strength and weaknesses. Based on this overview, we conduct a threat analysis of a CaaS platform by establishing a threat model and possible attack vectors in Chapter 3 and present possible mitigation strategies in Chapter 4. Finally, we conclude the deliverable in Chapter 5.

2 Container-based Virtualization

Container-based virtualization is a relatively new form of virtualization that quickly gains industry adoption. A container provides a self-contained execution environment for an application that runs isolated from other applications. This environment usually consists of the application itself and its system library dependencies. The resource isolation and separation between application processes is achieved by leveraging the capabilities of the underlying OS kernel. In essence, containers are a logical bundle of an application and all its necessary dependencies.

The distinctive property of containers is that they do not host their own OS kernel. Instead, the kernel is shared among all containers on a system. This shared kernel manages devices, provides an interface to kernel subsystems and enforces resource isolation. As a result, containers operate solely in user space. This is in contrast to VM-based virtualization, where a VM is a virtual representation of a physical machine. It is provided with virtualized hardware and must bootstrap a complete operating system. Thus, a VM must host an OS kernel.

This seemingly small difference provides containers with a number of advantages over VMs. First, containers can be smaller because they do not need to include a OS kernel. Second, container have a smaller resource footprint during execution because they do not have to execute kernel functions. Finally, containers are faster to spawn and terminate because they are system process and do not need to bootstrap a virtual system. However, there is a major drawback in that containers are not portable across different operating system. They are bound to the architecture and execution environment of the operating system they were developed for.

While many of the aforementioned concepts are available on several major operating systems, such as Windows, Unix and Linux, the following sections focus on Linux as the target and basis of container virtualization.

2.1 Linux Namespaces and Control Groups

The Linux kernel has been an important catalyst for the development and mainstream adoption of container-based virtualization. It implements the technical foundation that facilitates container execution using *namespaces* and *control groups* (*cgroups*).

Since their introduction into the Linux kernel, they have become synonymous for container-based virtualization on Linux. Nowadays, almost every container execution engine relies on the namespace and cgroup features to execute containers.

A namespace is an abstraction that allows the definition of a logical view on a global system resource, in which the instance of a namespace is an identifier for such a logical view. Each process references a particular namespace and can only access system resources associated with this referenced namespace instance. Two process must share the same namespace instance for them to have the same visibility of system resources. On the other hand, if two processes have distinct namespaces their access appears as though they are the sole user of that system resource. This later property also permits two process to request the same resource virtually using two distinct namespace instances. The Linux kernel will remap theses virtualized process resources to the global system resources.

Currently, the Linux kernel implements seven distinct namespaces. These namespaces are mount, PID, network, user, UTS, IPC and cgroup. For example, the mount namespace provides a unique view on mount points to a process. Thus, each mount namespace instance can reference a distinct mount hierarchy preventing any access from other processes in a different namespace instance. Similarly, the PID namespace provides separate process ID hierarchies. This allows two processes with different PID namespace instances to have overlapping PIDs. The user namespace creates new user and group IDs. Thus, a process can run as virtual root within its namespace but only have normal user privileges over global resources. The network namespace initiates unique networking buffers, routing and firewall tables. From the perspective of a process, it would have its own unshared network stack. Finally, the UTS namespace provides distinct host names, IPC separated inter process communication endpoints and queues, and cgroup isolates cgroup associations. The details of the latter are explained shortly.

While Linux namespaces separate resources seen by each process, they do not protect against excessive resource usage. To this end, Linux implements twelve different control groups. A control group (cgroup) is associated with a particular type of resource and controls how much of this resource can be used by a specific process. For instance, one can limit how much main memory a process is allowed to allocate. Once more is requested, the process is either killed or heavily penalized until sufficient memory can be made available.

The behavior of the individual control groups are very different, ranging from restrictions of resource usage to aspects of quality-of-service for network traffic. For example, while the *devices* cgroup restricts access to specific device nodes, the *cpu* cgroup ensures that a process gets its share of the CPU cycles when the system is under heavy load.

The feature of namespaces and control groups allows the kernel to separate the resources seen by individual processes. Thus, an application can run independent of other applications on the same host. If the application is also bundled with all dependencies such as libraries, a software or application container is formed. This is one of the reasons why OS-level virtualization is synonymous with containers. Another benefit of a software container is that they resolve dependency issues between applications. If two applications require the same library in different non-compatible versions, they can run side by side because they would only see their respective library.

2.2 Container Runtime and Images

The Linux kernel implementation of namespaces and cgroups provides the execution environment for containers. Container runtimes such as Docker, CoreOS rkt and LXC/LXD provide user space tools that allow to configure namespaces and cgroups in order to form a container.

While these tools bootstrap the execution environment for containers, containers themselves are bundled applications with their dependencies. These redistributable application packages consist of the standard Linux libraries at the basis. On top of them, developers can deploy and implement their applications. In its most basic form a container is a directory, which contains the Linux file system hierarchy used by the containerized application.

However, this approach of using directories is cumbersome. Instead, container images have been devised. A container image is nothing else than an archived directory with important technical additions. First, container images are usually arranged in layers. Instead of having one big archive, different aspects are separated into individual images. A full container can be initialized by stacking multiple layers, allowing them to be reused for multiple containers.

Depending on the platform used, these files can be bundled into redistributable archives. For example, Docker uses a format that creates reusable layers. Each layer contains a fixed set of libraries or packages such as the standard C library. In addition, layers are independent. Thus, once a layer has been created it can be reused for other applications. Docker will collect all layers and combine them into the final container image. This permits greater reuse of existing resources and makes each component of a software stack more independent. A developer could update any layer given compatibility is assured and create a new updated version of an application.

Another important factor in the adoption of redistributable containers is the development of a common container image format. Therefore, Docker and other industry partners founded the Open Container Initiative (OCI)[1] in 2015. Participating organizations thrive to create a common and interoperable specification of a container runtime (*runtime-spec*[2]) as well as an image format (*image-spec*[3]). The image-spec defines the format and the associated meta-data of a container. All OCI compliant container engines support containers provided in this format. Thus, OCI containers such as those created by Docker are can be deployed and executed on most container platforms.

Container-based virtualization virtualization leverages the kernel of the host operating system. Therefore, any application running as a container has to rely on the same kernel. Thus, containers on Linux must run binaries compiled for it. This limits the universality of container images because they have been created for a specific underlying OS.

[1]https://www.opencontainers.org
[2]https://github.com/opencontainers/runtime-spec
[3]https://github.com/opencontainers/image-spec

3 Threats Analysis and Attack Vectors

NFV combines various technologies and paradigms such as networking, cloud computing and the use of software to realize network functions. As a result, threats originate from each of these parts and from their combinations.

For example, the chaining of VNFs to realize a network service means that each component in the chain has to work as intended. Any fault or threat to any one component can compromise the whole service. This is in contrast to monolithic implementations of traditional network services, where risk is only regarded for the whole service. The NFV approach implicates additional and more subtle threats due to its use of distributed functionality but also gains certain security advantages. It has increased flexibility and is ability to response faster to security incidents, i.e. if only one component in the chain is affected, only this component needs to be updated.

To gather a comprehensive overview of threats, we present a generic architecture of a Container-as-a-Service platform for NFVI. Based on this architecture, we continue to describe the assumed threat model. We conclude this chapter with a discussion of attack vectors considering our architecture and threat model.

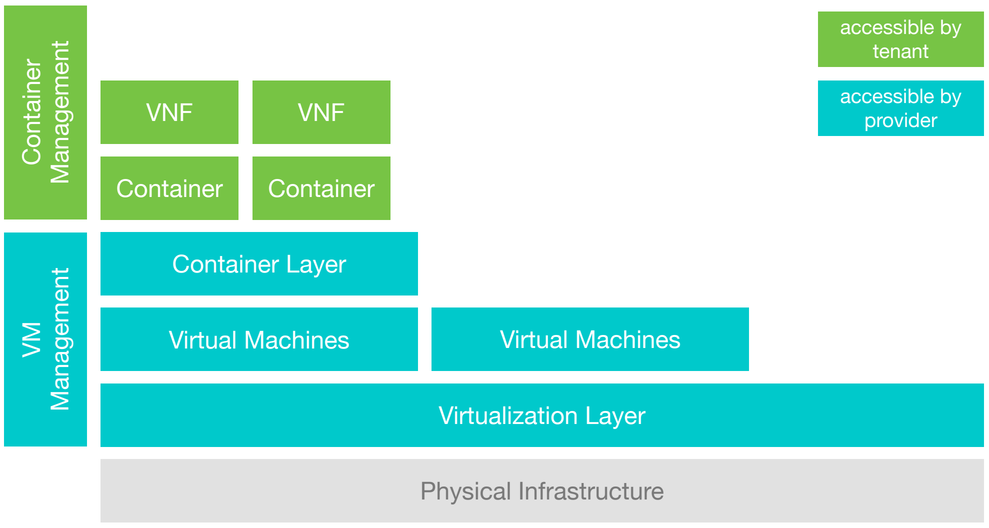

Figure 3.1: Generic CaaS platform for NFVI

3.1 Container-as-a-Service Architecture for NFVI

To provide a common basis for our evaluation, we assume the following infrastructure in a Container-as-a-Service (CaaS) scenario (see Figure 3.1). This generic infrastructure is based on talks and workshops with different experts from network vendors, operators and cloud providers. It has to be noted, that while we assume that the provider and tenant in our scenario are organizationally separated, these roles can be fulfilled within the same company, i.e. by different teams.

The provider is using a virtualized environment to create a multitude of VMs, each running a Container Layer, such as Docker or rkt. The provider is internally using a VM Management system, such as OpenStack[1], but this is not accessible by the tenant. To provide the container service to the tenant, he manages a Container Management platform, such as Kubernetes[2], OpenShift[3] or Docker Datacenter[4]. The tenant can use this platform to launch, destroy, orchestrate and monitor its own containers. The Container Management platform can also be seen as a connection point to a Virtualized Infrastructure Manager (VIM) component according to the ETSI NFV definition. The VIM could use the API offered by, e.g. Kubernetes to automatically launch and destroy VNFs, as instructed by the NFV Orchestrator or VNF Manager. Alternatively, the Container Management platform could also implemented the necessary VIM interfaces.

3.2 Threat Model

The methodology for establishing the presented threat model is loosely based on the *Application Threat Modeling* approach established by OWASP[5]. The definition of our threat model relies on four main components:

- **Actors** describe entities playing a role in our modeled infrastructure, either as a consumer or as a provider.
- **Assets** are components or information worth protecting, usually denoted by maintaining a certain *protection goal*, such as confidentiality, integrity or availability.
- **External Dependencies** describe factors that are external to the model but that have certain influence on the model itself, i.e. assumptions made or external regulations.
- **Entry Points** are the origin of an attack vector, i.e. attack surfaces.

3.2.1 Actors

In our threat analysis, we primarily focus on the benefits and challenges of using containers as a basis for NFVI. Therefore, we restrict our attention to platform providers and tenants as the actors in our threat model. From the perspective of providers, the main concern is how strict isolation between tenants can be enforced with containers and how providers can protect themselves against threats originating from rogue tenants. Tenants, on the other hand, demand assurance that providers are able to guarantee isolation of tenants, and need to know how they can take advantage of containers to improve the security of their services.

[1] https://www.openstack.org
[2] https://kubernetes.io
[3] https://www.openshift.com/
[4] https://www.docker.com/enterprise-edition
[5] https://www.owasp.org/index.php/Application_Threat_Modeling

We further assume that providers adhere to common cloud security guidelines. They behave honestly and apply state of the art security mechanisms to manage their infrastructure and virtualized environment based on VMs. However, we assume that providers are not fully familiar with the threats associated with deploying a container layer on top of VMs. They may also be unfamiliar with the specific security requirements of a container specific management interface available to tenants.

For the tenants, we assume that that perceived security benefits are overestimated and risks of containers are underestimated. Moreover, tenants may be unfamiliar how containers are effectively applied to deploy VNFs and what security practices are left as their responsibility. However, we assume that tenants adhere to general best practices with respect to software development and take sufficient precautions in securing their access credentials to the container management interface. In our analysis, we merely recognize the fact that skilled adversaries may still circumvent these precautions.

3.2.2 Assets

The assets in our threat model evolve around different types of data (stored or in transit) as well as the ability to control and use certain components of the overall service.

The first group of assets includes various types of credentials such as user names, passwords, private keys or session tokens. Especially because of their automated and distributed nature, network functions may often contain certain types of ephemeral credentials to access other parts of the system. The main protection goal relevant for this group is confidentiality, because an attacker could leverage this data to get unprivileged access to other parts of the system.

Another type of data encompass all business data and network data. This is any data pertaining to clients and business information, either retained in storage accessible by VNFs or ephemeral data transmitted to and from a network function. Depending on the sensitive nature of this data, either confidentiality and/or integrity can be seen as the most important protection goal for this kind of data.

To the management plane, we ascribe the ability to deploy, manage and orchestrate services. Tenants must be able to connect to the container management interface, deploy new VNF containers, maintain and orchestrate them, as well as be able to monitor and trouble shoot problems. Providers must retain the ability to identify and fix problems with the underlying virtualization layer. They must also be able to identify security incidences and take actions accordingly.

3.2.3 External Dependencies

We assume that tenants and providers use base images from trusted sources. For tenants, these are the base container images, such as images from an official Docker repository[6], which they implement their VNFs. Providers need to use trusted base images for their VMs, i.e. supplied by Linux distribution vendors. Moreover, providers could actually provide base images to tenants that are optimized for the underlying architecture.

[6]https://hub.docker.com/explore/

Another external dependency for providers is that we assume that they are providing a hardened infrastructure in accordance and compliance with security best practices or standards, such as ISO 27001. This point may extend to tenants, in case tenants process certain information within their VNFs, such as credit card information or financial data. In these cases, tenants must be compliant to the respective standards.

Lastly, certain VNFs may require direct access to the (virtualized) hardware, such as network cards or GPU. In this case, we assume, that the provider has taken appropriate measures to isolate such specialized offerings from the rest of the CaaS cluster.

3.2.4 Entry Points

As defined in our architecture, the VM management and virtualization layer are internal to the provider and not directly accessible by tenants. Therefore, entry points are limited to interfaces provided to tenants, services implemented in the VNFs and the containers themselves. Adversaries can exploit possible vulnerabilities in the VNFs or gain access to containers in case they expose accessible interfaces. In addition, adversaries can gain access through the container management interface. Finally, adversaries could establish a foot hold by impersonating a valid tenant. This would co-host a container of their choosing onto the CaaS platform creating a vantage point to investigate vulnerabilities of the internals of the CaaS platform.

3.3 Attack Vectors

Based on the established threat model, we now further explore possible attack vectors to compromise the security of the defined CaaS architecture. We have divided the attack vectors into four main categories roughly representing possible attack stages.

- First, adversaries can attack and exploit the VNFs or other software running in containers by mounting attacks over the network. Possible goals would be to hijack containers or cause denial-of-services.
- Second, adversaries can attack the container management by finding vulnerabilities in the user authentication and remote API. The intended goals would be to hijack accounts tenant accounts in the container management and perform denial-of-service again.
- Third, adversaries can use hijacked accounts on the container management to perform additional attacks. They can ex-filtrate valuable information, inject malicious container images and abuse resources for nefarious purposes.
- Finally, adversaries can use a hijacked or rented container on the CaaS platform to perform covert attacks against co-hosted containers. They can eavesdrop and spoof network traffic or try to escape container confinement and take over the underlying host.

Table 3.1 provides an overview of threats and attacks that we have considered in each category. Many of the discussed threats and attacks are common to cloud computing and have been subject in previous publications (e.g. [5, 6, 7, 8, 9]). Therefore, we are focusing our discussion on attacks and threats that are challenges on a CaaS platform or where a CaaS platform can provide benefits. In the remaining section, we will describe the threats and attacks.

Threat	Description
Attacks against VNFs and Network Services in Containers	
T1.1	Exploit software vulnerabilities and misconfiguration
T1.2	Compromise auxiliary network services
T1.3	Perform denial-of-services
Attacks against Container Management	
T2.1	Compromise credentials
T2.2	Bypass authentication and authorization
T2.3	Denial-of-service against container management
Exploiting Access to the Container Management	
T3.1	Ex-filtrate and manipulate data and sensitive information
T3.2	Manipulate configurations
T3.3	Abuse of tenant resources
T3.4	Inject malicious container images
Exploiting Access to Containers	
T4.1	Ex-filtrate sensitive information through side-channel attacks
T4.2	Escape container confinement
T4.3	Spoofing and eavesdropping on network traffic
T4.4	Attack internal network services

Table 3.1: Overview of threats and attacks considered in our analysis

3.3.1 Attacks against VNFs and Network Services in Containers

For attackers, VNFs and network services running on containers of tenants are prominent targets. First, they expose possible entry points on the local network and the Internet. This makes it easy to discover them during reconnaissance. For example, port scans are a prevalent technique to find possible entry points [10]. They have become so ubiquitous that nowadays they are routinely used to scan the entire Internet and probe for vulnerable services at a large scale [11], even listing them in special search engines for devices such as Shodan[7]. Thus, attackers can identify potential targets quickly.

Second, attackers have a greater chance of finding and exploiting vulnerable systems among the ones hosted in containers by tenants. For tenants, the primary focus is on the business value derived from developing and operating their VNFs. If security is not within this business focus, it inevitably

[7] https://www.shodan.io

becomes secondary. Moreover, tenants may simply lack the necessary resources to realize security measures properly and defend against targeted attacks effectively, especially if they are a small or medium enterprise (SME) [12]. This creates opportunities for attackers to compromise vulnerable systems successfully.

Finally, VNFs and their data are valuable targets. Attackers gain access to sensitive business data and can disguise malicious activities in the future. In the following, we enumerate the main categories of attack vectors that we believe attackers will be exploiting. These attacks are focused on compromising VNFs and containers or disrupting their services.

T1.1 Exploit software vulnerabilities and misconfiguration Software vulnerabilities and misconfiguration are one of the main security weaknesses that attackers will exploit to compromise systems. They are the underlying cause for many of the common security threats [13, 6]. This is unsurprising because secure software development and proper implementation of security mechanisms is very difficult. Thus, VNFs are subject to the same risks. Vulnerabilities can potentially exist in any component of the VNF's software stack, including third-party dependencies. They could potentially even exist in a container base image, if it stems from an untrusted or unmaintained source.

If attackers can find exploitable software, they can disrupt the offered network service and potentially take over the whole container. The actual consequences may vary. For example, an attacker may gain the ability to alter data transmitted to and from the service, redirect traffic, extract business information or subvert the intended service function. This could be particularly damaging for a security focused NFV service such as a VPN gateway or intrusion prevention system. In addition, an attacker taking over the underlying container obtains the ability to perform additional attacks. Especially in a highly distributed scenario, such as multiple VNFs chained together, attacking the "weakest" link may prove to be very effective.

This threat can be mitigated by: M1.1, M1.2, M1.3, M1.4, M1.6.

T1.2 Compromise auxiliary network services of VNFs and containers In addition to the main functionality of the VNF itself, tenants may deploy additional network services with their containers. These services can be built into the VNF itself for example to allow monitoring or remote configuration. Moreover, tenants may unnecessarily install remote access services such as SSH into their containers to administer them remotely. If these network services are directly accessible over the Internet (or from another tenant of the CaaS), they provide an additional entry point for adversaries. For example, attackers can try to guess access credentials or exploit known vulnerabilities in the network services. Once an attacker gained access to the container through these service, additional attacks become possible, similar to T1.1.

This threat can be mitigated by: M1.1, M1.2, M1.3, M1.4.

T1.3 Denial-of-Service against VNFs and containers Rather than trying to get unprivileged access to the container hosting the VNF, an attacker could launch a denial-of-service (DoS) against it. With it, an attacker can degrade service quality or disrupt services completely. While DoS attacks are not specific to container environments, two important properties can amplify DoS attacks.

First, containers running on a host share the underlying system resources. Thereby, containers facilitate greater density because they have a smaller resource footprint. However, greater density also increases the chance of resource contention and starvation. A well-executed DoS attack can cause a ripple effect that negatively affect all co-hosted containers and VNFs. For example, a malicious container can deplete the entropy pool for the pseudo-random number generator [14].

Secondly, container-based virtualization on Linux uses control groups to arbitrate resource usage between containers by enforcing limits. However, the initial implementation of cgrous (i.e. cgroups v1) had a number of shortcomings that negatively affected the enforceability of resource limits (e.g. [15, 16, 17, 18]). Moreover, strict resource limits combined with improper estimation of resource usage can lead to exploitable weaknesses. For example, the memory cgroup utilizes an out-of-memory killer to terminate processes that demand more memory than their conceded maximum. In particular, software such as web servers [17], that generate dynamic responses and thereby have a hard to predict resource usage, can be susceptible. An attacker could misuse the out-of-memory process termination by causing short-term request peaks that trigger the out-of-memory killer. This inadvertently facilitates denial-of-services rather than prevent them.

This threat can be mitigated by: M1.4, M1.6.

3.3.2 Attacks against the Container Management

A CaaS platform requires a container management interface for tenants to deploy, manage, orchestrate and monitor their containers. This interface is essential for the provider and the tenant. If the provider implements an interface that is vulnerable to attacks, attackers can gain access to the CaaS platform. In addition, any vulnerability would strain the trust relationship between provider and tenants. For the tenant, this interface is possibly the only connection to their deployment. Thus, losing it would mean loss of control over their deployments. As a result, protecting the container management interface is an important task for the provider.

T2.1 Compromise credentials The container management interface encompasses a great deal of privileges because anyone gaining sufficient access is able to deploy new instances and disrupt existing NFV services. It may also be possible for an adversary to submit compromised container images that unsuspecting tenants then use to initiate NFV services. Moreover, adversaries can use the same access to extract business data. For example, they may be able to create backups of container instances or they can export container images. The impact of compromised credentials is exacerbated by the fact that weak and insufficient safe guarding of credentials is recognized as one of the top threats in cloud computing [6].

This threat can be mitigated by: M2.1, M2.2.

T2.2 Bypass authentication and authorization Usually the container management is exposed to the tenant in a web front-end or REST API. In case these interfaces contain software vulnerabilities or implement authentication and authorization insufficiently, an attacker would be able to gain access to the container management and pose as a tenant. It is also possible that an attacker gains the ability to submit requests without prior authentication and authorization. The Cloud Security

Alliance (CSA) has named this one of the top threats in cloud computing in 2016 and it extends from the cloud to the container world [6].

This threat can be mitigated by: M2.1, M2.3.

T2.3 Denial-of-service against container management A denial-of-service attack against the container management can interfere with the ability of tenants to control and maintain their deployments. This can lead to the inability to react to changing resource requirements. In addition, the container management is the external API to interact with the CaaS platform. Thus, other services may become inaccessible as well. For example, tenants may be unable to retrieve logs. An attacker could use this opportunity to hide additional attacks on container instances.

This threat can be mitigated by: M2.4.

3.3.3 Exploiting Access to the Container Management

Once an attacker has gained access to the container management, i.e. by utilizing T2.1 or T2.2, additional attacks become feasible. Due to the central role of this interface, these attacks can cause a lot of damage for either tenant or provider. This problem is exacerbated if tenants and providers consider the container management interface as the only security boundary. In that case, internal access to data and services may be less guarded. This would simplify the work of adversaries, as they do not have to overcome additional security measures once they gained access.

T3.1 Ex-filtrate and manipulate data and sensitive information With access to the container management, an adversary gains access to all resources a tenant uses or has used on the CaaS platform, unless additional safe-guards, such as access control, i.e. using different users and roles, are in place. First, there could be logs, snapshots and persistent data volumes. An attacker can export this data from the platform and extract usable information. This type of data may proof very valuable because data may be historical and not within the immediate attention of tenants. Thus, attackers may find data from older or no longer existing deployments. Persisted data could also potentially include configuration data and possibly credentials. These would provide an attacker with details on additional targets and possible attack vectors. Ex-filtrated credentials would gain an attacker direct access to sessions, VNFs and containers. An attacker could also download container images used by the tenant. These images contain the code of VNFs and usually represent confidential business data for the tenant. Finally, an attacker could simply delete all data to disrupt services and cause business loss as tenants would for example lose important customer data.

This threat can be mitigated by: M1.2, M1.5.

T3.2 Manipulate configurations An adversary can also manipulate configurations through the container management. This could be used to implement back doors and other vulnerabilities that the adversary can use to maintain long-term access. For example, an adversary could open additional ports on a container that could then be exploited.

This threat can be mitigated by: M1.5.

T3.3 Abuse of tenant resources The container management is also the interaction interface between tenants and the CaaS platform. Common operations are the creation and configuration of compute, network and storage resources. An attacker can use the same operations to create and abuse resources from the CaaS platform. For example, an attacker could instantiate a container on behalf of the compromised tenant and use it for other nefarious purposes [6].

In addition, the container management may have an interface to connect to containers remotely. Thus, an attacker can connect to containers to install back doors that are more persistent or extract data from a running container instance. The latter could bypass security mechanisms that would normally protect data at rest.

This threat can be mitigated by: M1.4.

T3.4 Inject malicious container images As part of the CaaS platform, a container image repository must be maintained because it may not be efficient to always retrieve container images from remote locations. Access to the repository would usually also be controlled by the container management. An attacker can then use the unprivileged access to inject malicious container images. If an attacker can convince tenants that his image is an official release, he can obtain a persistent presence in all container instances that use the malicious image.

This threat can be mitigated by: M1.2.

3.3.4 Exploiting Access to Containers

Once an adversary has possession of a container on the CaaS platform, additional attacks become possible that could compromise the overall security of the CaaS platform. Thereby, an adversary does not have to exploit a vulnerability in a VNF, container or the container management. Instead, adversaries can pretend to be legitimate tenants and rent resources. This would provide them with access to the system without arousing suspicions. In combination with covert side-channel attacks, simply renting resources would represent an attack vector that is very difficult to detect and defend against.

In addition, adversaries can exploit weaknesses and vulnerabilities to compromise container isolation and to break out of container confinement. For example, attacker can use Linux kernel vulnerabilities to easily breach isolation and confinement because all containers on a host share a kernel [14]. Moreover, container runtimes can introduce vulnerabilities through implementation flaws (e.g. CVE-2015-3629[8] or CVE-2015-3630[9]). Finally, insufficient restrictions on capabilities and permissions can provide sufficient leverage to break out of containers [19, 14].

With access to a container on the CaaS platform, attackers also gain access to its network resources and services. Thus, malicious users can mount network attacks such as spoofing, eavesdropping, man-in-the-middle, and denial-of-services. As these attacks originate from within the CaaS platform, they can catch tenants off guard who neglect to secure their platform internal network traffic and services.

[8]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3629
[9]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3630

The following sections will provide additional details on a number of threats that attackers can exploit once they have access to a container.

T4.1 Disclose sensitive information through co-residence One inherent weakness of a CaaS platform and other cloud computing services is that multiple tenants share the underlying system resources. This introduces the problem of co-residence [20]. Adversary can use side effects resulting from a shared resource usage to deduce information from co-hosted containers. Security researchers have shown that it is possible to leak sensitive information (e.g. [20]), extract secret keys (e.g. [21, 22, 23]) and even manipulate memory (e.g. [24]) from co-hosted instances. In addition, residual data from previous users, for instance left on insufficiently wiped volumes, can be reconstructed [5].

Because these attacks use normal system operations like reading from disk and memory, they are difficult or impossible to detect. This allows adversaries to extract sensitive information and data covertly. As a result, tenants may remain unaware of a data breach until that data has been leaked.

This threat can be mitigated by: M1.3.

T4.2 Escape container confinement The greatest threat against the integrity of a CaaS platform is the escape from container confinement. A successful escape from container confinement makes it very likely that an attacker gains full control over the underlying host. In the case of containers, this would automatically comprise all co-hosted containers. In addition, adversaries would gain the ability to mount attacks against the core infrastructure of the CaaS platform.

There are multiple ways for attackers to escape container isolation and table 3.2 exemplifies a number of known CVEs from the past years. One way to escape containers is to exploit vulnerabilities in the Linux kernel. The Linux kernel enforces container isolation by employing namespaces and cgroups. However, if attackers gain kernel-level privileges through privilege escalation, they can circumvent isolation (e.g. CVE-2014-4699 and CVE-2016-3134). What is even more dangerous about Linux kernel exploits is the fact that all co-hosted containers share the same kernel. An attacker with a working kernel exploit automatically compromises all co-hosted containers of any tenant. Thus, the security omissions of a single tenant endanger all other tenants residing on the same host.

Another attack vector can leverage vulnerabilities in container runtimes. They are responsible for bootstrapping containers and properly configuring namespaces and cgroups. Unfortunately, this can be prone to mistakes (e.g. CVE-2015-3629, CVE-2015-3630 and CVE-2016-9962). For attackers, however, these become opportunities to escape containers.

As malicious tenants, attackers can also introduce vulnerabilities at will. For example, they can request resource based on specially crafted images that make use of known vulnerabilities (e.g. CVE-2015-3627 and CVE-2015-3630). Similarly, they can request a container instance where the configuration is intentionally less secure and would facilitate attacks to escape containers. There are a number of Linux kernel resources and capabilities that are known to be exploitable when a container instance gets unrestricted access to them [14]. Moreover, attackers could request privileged containers that share their users and groups with the host. Thus, the root user in the

CVE	Description
CVE-2013-1957	Bypass intended read-only property of a filesystem by leveraging a separate mount namespace
CVE-2014-4699	Gain privileges, or cause a denial of service through ptrace and fork system calls
CVE-2014-5206	Bypass intended read-only restriction and defeat certain sandbox protection mechanisms through use of a user namespace
CVE-2015-2925	Bypass intended container protection mechanism by renaming a directory
CVE-2015-3627	Gain privileges through a symlink attack in an image
CVE-2015-3629	Escape containerization and modify files on host when respawning a container
CVE-2015-3630	Modify host, obtain sensitive information, and perform protocol downgrade attacks through crafted images
CVE-2016-3134	Gain privileges or cause a denial of service (heap memory corruption) through setsockopt call in netfilter kernel subsystem
CVE-2016-5195	Gain privileges by leveraging incorrect handling of a copy-on-write feature to write to a read-only memory mapping
CVE-2016-9962	Escape containerization by using ptrace on processes joining a container

Table 3.2:
Overview of CVEs against container virtualization and container platforms. Descriptions are derived from Mitre's CVE (https://cve.mitre.org/) and NIST's NVD (https://nvd.nist.gov/).

container is also the root user on the host. This obviously plays into the hand of adversaries, as any container escape will automatically make them root on the host with all its security implications.

The ramifications of a successful container escape are significant. As mentioned before, an attacker gains full control over the underlying host. This compromises all co-hosted containers and tenants. In addition, the unrestricted access to the container runtime would allow starting new container instances that may not be accounted for within the CaaS platform. Thus, attackers could hide malicious containers. Similarly, they can subvert all container-related operations for example through man-in-the-middle attacks. Moreover, gaining access to the underlying VM provides new opportunities to exploit vulnerabilities in the core infrastructure of the CaaS platform. Successful attacks on the virtualization layer and core infrastructure can compromise the CaaS platform as a whole.

This threat can be mitigated by: M3.1, M3.2, M3.3, M3.4, M3.6.

T4.3 Spoofing and eavesdropping on network traffic Just like VMs, containers must be connected to the underlying network interface, which in our model is provided by a VM. There a different methods to connect containers. One way would be to give containers direct access to the network interface of the VM. However, this opens up many attack vectors. For one, containers would share the same network stack allowing other containers to intercept and spoof network traffic destined for co-hosted containers. Secondly, direct access to the underlying network would allow adversaries to gain valuable information on the internal network traffic. This would permit attacks against internal provider services.

Another way to connect containers to the underlying network is to create a virtual switch or a bridge on in the container runtime. Containers are then connected to this switch or bridge very similar to how the networking of VMs is connected to virtual switches in the hypervisor. Combined with network namespaces, this would provide isolation of the network stack between containers. However, containers now share a common network segment. In particular, bridge devices are susceptible to spoofing and eavesdropping attacks that exploit the fact that they share a common network segment.

This threat can be mitigated by: M3.2, M3.5.

T4.4 Attack internal network services In addition to attacking the network between containers, adversaries can also attack supporting services such as Kubernetes DNS service, which is only reachable from within the cluster network. The highly distributed nature of containers requires shared services for example for coordination and service discovery. An attacker can target these services to degrade services. For example, a denial-of-service against the service discovery infrastructure could prevent NFV services to react to changing resource requirements properly. Thus, the NFV service may no longer be able to scale appropriately to sudden demand spikes.

This threat can be mitigated by: M3.2, M3.3, M3.5.

4 Mitigation Strategies

Based on the description of possible threats and attacks against the CaaS platform, it becomes possible to devise strategies that can mitigate risks. In this chapter, we discuss these mitigation strategies. Thereby, we focus on providing a broad overview of useful directions with some prominent examples. We explicitly abstain from giving detailed instructions to realize the presented strategies because specific instructions may change over time and are not applicable to all use cases. Instead, we hope that our discussion will engage the reader's interest to further investigate presented strategies and adopt those that fit their specific requirements.

In addition to our discussion, a large number of useful guidelines and best practices exist that are relevant for container-based virtualization and cloud computing in general. Therefore, we provide them here collectively and highly recommend that readers consult them for additional references. First, the Cloud Security Alliance[1] publishes a wide range of reports and papers concerning all aspects of security in cloud computing e.g. [7]. Second, national and supranational security agency such as NIST[2] (e.g. [8, 9, 25]), BSI[3] (e.g. [26, 27, 28]) and ENISA[4] (e.g. [12, 29]) provide technical reports and guidelines in accordance with national cybersecurity recommendations and requirements. The Center for Internet Security[5] on the other hand publishes CIS Benchmarks that are a collection of security-enhancing configuration guidelines with conformity checks for specific software products. Among others, CIS Benchmarks exist for Docker [30], Kubernetes [31], and different Linux distributions (e.g. [32, 33, 34]). Finally, the NCC Group white paper by Aaron Grattafiori [14] compares the security model of three container engines and discusses how certain configurations can lead to vulnerabilities.

4.1 Securing VNFs

In the previous chapter, we have outlined possible attacks against VNFs and additional threats once an attacker gained access to a container. Thus, one important countermeasure is to secure VNFs. This primarily means to ensure that VNFs and software deployed in containers does not contain vulnerabilities. Moreover, once new vulnerabilities are discovered, processes should be in place to quickly update running containers.

Secondly, security measures and best practices to software development and system administration should be followed. This ensures that containers withstand attacks or at least make them more

[1] https://cloudsecurityalliance.org/
[2] https://www.nist.gov/
[3] https://www.bsi.bund.de/EN/
[4] https://www.enisa.europa.eu/
[5] https://www.cisecurity.org/

difficult. This would provide time and possibilities to identify ongoing attacks and initiate countermeasures. To this end, a monitoring and logging service should be operated and continuously reviewed. In addition, automatic warning systems can call attention to suspicious behavior.

While tenants must realize most of the following mitigation strategies, providers can positively reinforce better security practices. From a security perspective, this can become a positive feedback loop that is beneficial to both of them. Moreover, tenants and providers should cooperate on certain security practices to take full advantage of them.

M1.1 Employ a secure software development life-cycle Tenants should assume that their VNF software contains flaws and have appropriate processes in place to mitigate these cases. To this end, they can adopt a secure software development life-cycle (S-SDLC). A S-SDLC integrates security considerations into the normal software development life-cycle. This ensures that risks, threats and security mechanisms are formalized alongside the development of VNFs.

Another important part of S-SDLC is to define and implement good coding practices. This includes writing documentation, adopting secure programming guidelines, enforcing code quality metrics and perform software tests. In particular, code quality and adherence to coding standards can be checked automatically with tools, such as SonarQube[6]. Most importantly though is that these policies are mandatory and code that does not meet standards is rejected. With this approach, tenants reduce the chance of introducing easily exploitable vulnerabilities into their code.

While containers do not help tenants in writing more secure code, they do facilitate a more continuous testing. Modern continuous integration (CI) and continuous delivery (CD) systems can build new container images from code repositories, such as Git[7] automatically. Thus, every commit can be used to generate a new container image that is immediately deployed for testing in a development environment. In fact, containers support this streamlined process because they are self-contained software bundles. Because developers use the same container blueprint for development that are later used for testing, there are no conflicts in the software dependencies.

In addition, once a new software version passes all tests, it can be deployed automatically to the CaaS platform. This step should be safeguarded by an internal approval processes. For example, tenants can configure their CI/CD pipeline to only start deployment in production after quality assurance, security testing and operations have approved the new version.

The later step can be important to prevent sensitive data to be leaked in container images. During development, developers may include private keys and configuration files with passwords into their container images for convenience and testing reasons. However, this information should never be published in a production container image. Thus, one of important step before actually deploying any container image to the CaaS platform is to verify that no sensitive information is included. In case of Docker, recent versions allow to specify these image differences between development and operations by using multiple FROM commands, i.e., one for development and one production in the respective Dockerfile.

Containers also support a continuous update of software dependencies. Because everything is bundled in the container image, there is no conflict between versions available in development

[6] http://sonarqube.org
[7] https://git-scm.com

and in operations. Thus, as long as newer libraries do not generate conflicts, updates can be applied immediately for testing and quickly rolled out to production. This can reduce the time a vulnerable container is maintained running. Moreover, orchestrators like Kubernetes and Docker's swarm mode support different deployments mode that support a downtime free transition between outdated and updated software versions.

Another important advantage of the CaaS platform for NFV is that containers promote a micro-service architecture. In short, software is decomposed into single independent and replaceable components. A similar decomposition is envisaged for VNFs. The main benefit from a security standpoint is that these independent components are very loosely coupled and should be replaceable. Thus, any update in a single component should be deployable without causing conflicts.

This mitigation strategy can help to mitigate: T1.1, T1.2.

M1.2 Maintain a secure container life-cycle Container-based virtualization introduces containers as an immutable artifact of software development and deployment. The software development life-cycle generates a new container image for every new version of the software. As a result, tenants have to maintain a secure container life-cycle as well to ensure that new versions of a container are deployed and old ones are terminated. The software development and container life-cycle management are overlapping and development decisions influence container decisions and vice versa. In addition, containers have their own life-cycle once they are deployed.

Containers are built up from base images. These base images are the foundation of VNFs and tenants must place their trust into the provided libraries. As mentioned before, if an adversary is able to introduce malicious container base images into the development pipeline all derived images will be compromised. Therefore, tenants must ensure that the base image they use is from a trusted source.

To this end, Docker has introduced Notary and Content Trust[8]. Notary and Content Trust define mechanisms and processes that ensure that image metadata is signed and verified. Thus, if tenants download an image from a party they trust, they can verify the signature and the cryptographic hashes of the image files. Similar security mechanisms are also present in other container runtimes, such as rkt.

By using signed and verified base images, tenants secure their initial step in the container life-cycle. However, once a container image is generated and possibly deployed, a continuous verification of the container content is required. During the use of a container image, contained libraries and dependencies might prove vulnerable to attacks. In such cases, the vulnerable images must be replaced with updated once that contain the updated library versions. In order to identify whether a container image contains vulnerabilities, automated vulnerability scanners can be employed. For example, Docker Security Scanning[9] and CoreOS Quay Security Scanner[10] are enterprise-ready vulnerability scanners for container images. In addition, CoreOS maintains the open-source project Clair[11] which provides the underlying scanner for their Quay Security Scanner. These regularly scan images and generate warnings once vulnerable dependencies are detected.

[8] https://docs.docker.com/engine/security/trust/content_trust/
[9] https://blog.docker.com/2016/05/docker-security-scanning/
[10] https://coreos.com/blog/quay-security-scanner-now-powered-clair-10
[11] https://github.com/coreos/clair

Another security measure is to secure the actual deployment pipeline. So far, tenants ensured that container images are generated from secure and trusted base images and that their images do not contain vulnerable software. However, adversaries are still able to inject malicious images into deployment or force a roll back to a now vulnerable image. To prevent this, tenants can utilize tools such as Content Trust and Notary themselves. They can sign their own images before hosting them on the providers CaaS platform. During the deployment process, they can then specify rules to only allow images signed by themselves to be deployed. In addition, they can fix container image versions by using their cryptographic hashes. Thus, container images are uniquely identified by their digest. This is enabled as container registries nowadays use content addressable storage.

Finally, tenants must initiate regular cleanup operations. As container images are constantly generated, the container registry quickly fills with images. To prevent attackers from running outdated and vulnerable images, the container registry should be regularly pruned. This ensures that only up-to-date image versions are present to be executed. However, the cleanup operations also extend to the container instances. Container instances may have associated files such as temporary files and logs. These files must also be pruned if container instances are terminated. This prevents that attackers can extract valuable data from past instances.

This mitigation strategy can help to mitigate: T1.1, T1.2, T3.1, T3.4.

M1.3 Harden containers Ensuring that container images and VNFs are free of known vulnerabilities prevents easy exploitation by attackers. However, the risk of unknown or zero-day vulnerabilities remains. A mitigation strategy against these is to harden containers. Hardening is a process that minimizes the attack surface and mitigates common exploit vectors by applying security best practices and employing software protection mechanisms.

One hardening approach is to use specifically compiled binaries that utilize protection mechanisms against common exploits such as buffer, heap and stack overflows. The purpose of these mechanisms is not necessarily to prevent exploitation but to make it a deterrent against attackers. There are security focused GNU/Linux distributions available that distribute protected binaries and system libraries. For example, Alpine Linux[12] is one such distribution and is thus often used as a base image for production-ready containers. In addition, the discovered vulnerabilities due to co-residence has motivated the development of specifically designed libraries, especially cryptographic ones [22]. These libraries can prevent side-channel attacks between containers.

Additionally, tenants should use minimal systems. These systems only contain the software required for their function. This is actually encouraged by containers. One of the security best practices for containers is to run only one service per container. This reduces the attack surface because there are fewer parts that can be exploited. Moreover, it becomes easier to maintain the container images because each container image has a well-defined and small set of software packages installed. The one-service-per-container paradigm also facilitates independence between containers. It becomes easier to scale individual parts of a system that experience resource shortages.

The adoption of security best practices during hardening also ensures that software is configured properly. Hardening guidelines for specific software packages always include secure configuration defaults and highlights possible pitfalls. These guidelines however should always be adapted to

[12] https://alpinelinux.org

the specific use case for tenants. It is the responsibility of the tenant to be diligent about applying security guidelines and best practices.

This mitigation strategy can help to mitigate: T1.1, T1.2, T4.1.

M1.4 Perform logging, monitoring and alerting The continuous security of running VNFs and containers depends on the ability to identify attacks and reconstruct them. The earlier tenants identify suspicious behavior and possible attacks, the faster they can initiate countermeasures. In the best case, they are able to stop attacks severe before harm is done. In the worst case, they at least are able to reduce the attacks severity and collect important details on the attack itself. This knowledge is important to find and fix flaws and vulnerabilities to prevent similar attacks in the future. Therefore, a key activity is to perform logging, monitoring and alerting.

The importance actually increases because of the dynamic container deployment. Containers may be started when demand arises and then quickly be terminated once they are no longer required. Furthermore, the focused nature of containers and the application of a microservice architecture means that there are more systems to monitor. As a result, monitoring and logging must handle this dynamic landscape and be able to collect data from many systems at once.

It is important however to achieve a balance between minimal container images on the one side and sufficient monitoring on the other. One option is to use built in monitoring and logging facilities in container orchestrators. For example, Kubernetes contains a logging collection facility that automatically collects terminal output from containers. Thus, tenants could simply write system logs to console and have Kubernetes collect and enrich them with associations to the originating container and service.

Another option is to use dedicated containers solely for logging and monitoring. Other containers simply have an agent that emits logging message to this central container or container cluster. This is in line with the one service paradigm because the logging containers have only the purpose of logging. Moreover, they can be secured by not exposing them to the outside world and only allow containers to interact with them on a private network. This can also be extended to create logging for each service separately preventing information from different systems to be compromised.

In case persistent log files are required, tenants can use persistent volumes. These can be attached to container instances and logs are then written as files to these volumes. This can be extended to use encrypted file-systems on the used volumes. This would prevent attackers from reading logs they may have obtain through other means.

Finally, if tenants require remote access for some reason, there are again built-in mechanisms to attach to a container. This creates a connection to a shell running in a remote container. In Kubernetes, this connection is relayed to the user. In case of Docker, tenants would require first access to the host and would then be able to attach to a container using the interface to the Docker Engine. If providers do not want to provide this level of access to the container runtime, they can implement a forwarding mechanism similar to Kubernetes or provide a web based management console similar to ones available on modern server-class hardware equipment. In short, remote access should always be realized using the attach mechanisms for containers and then securely relaying the resulting terminal session.

This mitigation strategy can help to mitigate: T1.1, T1.2, T1.3, T3.3.

M1.5 Utilize secure configuration and secrets management One of the challenges with containers is the management of configuration data and secrets, such as passwords or credentials. Containers are immutable artifacts and in order to create customized instances, configurations must be injected upon instantiation. In addition, secrets and passwords should never be included into container images. For one, it creates a vulnerability because adversaries can extract them by obtaining a copy of the image and they are potentially shared with third parties. It also makes it more difficult to adjust them and would require multiple copies of the same images with just different secrets.

The better option is to inject configuration upon instantiation or to request them from a central management meta-data system during run-time, similar to the cloud-init[13] system wildly used in cloud computing. The idea behind a centralized configuration and secrets management is that configuration options are decoupled from their values. A container is defined with names for configuration options and locations of secrets. After the container is started it will request these informations from a central management service. Examples in the container world include the ConfigMap[14] and Secrets[15] concepts of Kubernetes. Thus, tenants can deposit different versions and map them to specific container instances. In addition, many systems ensure that only authorized container instances get access to their and only their secrets. Hence, even though secrets reside centrally this provides security if implemented correctly.

This mitigation strategy can help to mitigate: T3.1, T3.2.

M1.6 Take advantage of microservices and automated container scheduling Taking advantage of container and their benefits means to adopt a system architecture based on microservices. A microservice is an independent software component that performs a single, focused task. In addition, microservices have well-defined interfaces. This interface allows to create loosely-coupled and modular system composed of microservices. Thereby, each mircoservice can be maintained, scaled and replaced independently of the other microservices. This allows users to scale individual parts and services. The result is a highly scalable system.

Another benefit of containers is that they are very lightweight. They are processes and only require the time to prepare their environment, setup namespaces and cgroup, and start a new process. Thus, it becomes very efficient to start and terminate containers as the need arises. If the current system load is high, additional containers are spawned within seconds and are as easily terminated, once the load subsides. The adoption of a microservice architecture supports this rapid scaling, because components are independent.

Container cluster management systems such as Docker in swarm mode[16] and Kubernetes[17] leverage microservices and the rapid deployment of containers to facilitate an automated service scaling. They can monitor the system, detect faults and spawn a new container that replaces the faulty one. They can also detect increased system load and can auto scale the cluster within predefined limits.

[13]https://cloud-init.io
[14]https://kubernetes.io/docs/tasks/configure-pod-container/configmap/
[15]https://kubernetes.io/docs/concepts/configuration/secret/
[16]https://docs.docker.com/engine/swarm/
[17]https://kubernetes.io/docs/tasks/

For VNFs, this provides scalability and fault-tolerance. In addition, it can protect against certain attacks. For example, auto scaling of services can reduce the impact of denial-of-service attacks. Moreover, tenants can use the same features to safely roll out new software versions. This makes it possible to react quickly to discovered vulnerabilities and deploy bug fixes to production.

This mitigation strategy can help to mitigate: T1.1, T1.3.

4.2 Securing the Container Management

One of the entry points that enables attackers to gain broad access to the CaaS platform is through the container management (CM). Once attackers gain access, they can start new containers, manipulate existing ones, inject malicious code and ex-filtrate data. At the same time, the CM is an important interface for tenants to manage and orchestrate their NFV deployment. Therefore, securing the container management interface is paramount.

M2.1 Implement strong identity and access management The CM must support strong user authentication and authorization. For example, role-based access control can be applied. This would allow tenants to create roles with restrictive permissions that are tailored towards individual NFV services. Individual groups can be created that administer parts of a specific NFV service. Thus, a compromise would restrict the exploitability to one service.

To further secure user authentication and authorization, public key-based authentication and two-factor authentication should be supported. The former prevents tenants from using simple, easily guessable passwords – a common vulnerability [6]. Moreover, multi-factor authentication (MFA) can further secure security critical operations by requiring a user to provide a second authentication token. This would make it more difficult for attackers, because they would need to acquire possession of the additional token. Production-grade container management systems such as Kubernetes also support delegating the authentication to already existing systems using technologies such as OpenID Connect as well as a generic hook system to include further external authentication frameworks. Using an already in-place system, such as a corporate LDAP or Single-Sign On (SSO) has many advantages. First, password and credential policies can be managed in one central place. Second, access rights can be given to already existing users and groups and third, revoking access, i.e., in case an employee leaves the organization, is enforced immediately.

This mitigation strategy can help to mitigate: T2.1, T2.2.

M2.2 Secure communication with CM Another important mitigation strategy is to provide secure communication channels between tenants and the CM. The first step in providing secure communication channels starts with selecting state-of-the-art cryptographic algorithms. Newly discovered weaknesses and increasing compute power make some cryptographic algorithms obsolete. Likewise, other algorithms may require larger parameters to maintain the same level of security. Therefore, it becomes important to consult the recommendation regarding cryptographic algorithms and their parameter. These are published, for instance, by NIST [35, 36], ENISA [37] and

BSI [38]. A collected overview of recommendations from many organizations can be found at the BlueKrypt's Keylength.com[18] project.

Cryptographic algorithms also affect the creation of a digital certificate for the CM. With a trusted digital certificate, the tenants can validate the authenticity of the CM preventing man-in-the-middle attacks. Therefore, providers must choose a trustful certificate authority and generate state-of-the-art certificate. New certificates should no longer use the hash functions MD5 or SHA-1 in their signatures because they have known weaknesses. In particular, exploits against certificates with MD5-based signatures are known [39].

Finally, providers must configure the communication channels properly. This often requires disabling deprecated cryptographic algorithms. For example, communication over HTTP secured by Transport Layer Security (TLS) becomes vulnerable to a number of attacks if deprecated ciphers are not prohibited and downgrades attacks are possible [40]. For HTTP and TLS, SSL Labs[19] by Qualys provides the ability to test and grade ones certificate and server configuration.

This mitigation strategy can help to mitigate: T2.1.

M2.3 Employ secure life-cycle management for CM Providers should be prepared that the CM implementation and its API contains flaws. Therefore, an update strategy should be defined. In case a vulnerability is found, the provider should be able to quickly fix any problems and mitigate possible damages. To this end, providers can employ containers themselves to provide the CM. This allows them to quickly deploy new versions of the interface.

This mitigation strategy can help to mitigate: T2.2.

M2.4 High-Availability Because the CM is important for tenants to manage and orchestrate their NFV deployments, providers must ensure high-availability. Again, they can rely on the container platform themselves and deploy their CM software using containers. This would allow them to take advantage of the rapid scaling and fault-recovery features that their container environment provides.

This mitigation strategy can help to mitigate: T2.3.

4.3 Enforcing Container Isolation

Another big part of securing CaaS is to strengthen container isolation. Containers running on one host share a common Linux kernel. Therefore, any attack that gains kernel level execution privileges has a high chance of compromising other containers running on the same host. Providers have to be particularly rigorous in enforcing proper isolation.

The problem of enforcing container isolation is partly aggravated by the fact that the Linux kernel has a large interface. Current kernel versions expose more than 300 system calls that if exploitable

[18]https://www.keylength.com/en/
[19]https://www.ssllabs.com/

would allow an attacker to access and abuse kernel resources. Thus, one mitigation strategy is to minimize the interface between the host and containers.

Another important mitigation strategy is to restrict the damage attackers can do once they successfully escaped a container. Again, the Linux kernel provides a couple of security features that can be used to restrict user actions. Among them are Linux security modules providing mandatory access control and user namespaces themselves.

M3.1 Filter system calls One method to restrict the large system call interface is to use seccomp. The seccomp ('Secure Computing') integrates a BPF-like filter mechanism for system calls into the Linux kernel. This can be used to reduce the number of system calls that a container can effectively use. Moreover, the parameters to system calls can be sanitized. Thus, seccomp reduces one attack surface by reducing the number of exploitable system calls.

This mitigation strategy can help to mitigate: T4.2.

M3.2 Restrict root privileges In addition, the Linux kernel allows to restrict users' capabilities to perform specific operations. In particular, root users have a vast set of permissions. Linux capabilities permit a fine-grained partition of the capabilities bundled into the root user. Thus, instead of being root, different capabilities can be restricted. For example, a user can be root without having the permissions to change network settings. Currently, the Linux kernel has 37 capabilities. Unfortunately, not all capabilities have a specific, well-defined scope. For example, CAP_SYS_PTRACE permits the use of ptrace(), while CAP_SYS_ADMIN comprises many permissions such as changing the host and domain name, using mount() and umount(), or performing various configurations of storage and memory devices. Restricting capabilities assigned to container processes limits the operations an adversary can use on the host system.

This mitigation strategy can help to mitigate: T4.2, T4.3, T4.4.

M3.3 Increase protection through namespaces and cgroups Namespaces and cgroups are the two features that enable container-based virtualization in the Linux kernel. They provide isolation and separation of processes. However, their default configuration is often not as restrictive as possible to support many use cases. Thus, it is important to employ many namespaces and cgroups and tighten their configuration.

For example, the user namespace separates the user IDs on the host system from the user IDs of a container. A user within the container can be root, while on the host system the same user is unprivileged. As another example, the devices control group can restrict what device nodes can be created and be accessed. Other control groups limit the maximum resources a container can use. Thus, they prevent degradation or denial of services in a shared environment.

This mitigation strategy can help to mitigate: T4.2, T4.4.

M3.4 Utilize Linux Security Modules Linux security modules (LSM) are are software components that use a well-defined kernel interface to control and decline certain operations. In particular, LSMs providing mandatory access control (MAC) define rules that further restrict what files a process can access. Two well-known LSM MACs are AppArmor and SELinux. These MACs define policy rules about what a process can and cannot access. The resulting policies are then enforced by the kernel. Thus, adversaries are on the one side further limited on the resources that are exploitable. On the other hand, they have to invest additional time and resources to circumvent and disable the LSMs.

So far, these security mechanisms mostly protect the host from containers. If a container is compromised, an adversary still has to put in the additional effort to compromise the host. However, these protections may not circumvent exploits across containers on the same host. For example, containers on the Docker engine share a common user namespace and storage space for the container images. Thus, an adversary escaping one container may have access to the container images of another container. This can be used to manipulate images and introduce persistent threats or steal information. One safeguard against these cross-container exploitations are extended features of LSM MACs known as multi category security. Using MCS, every container gets a unique identity when it is initialized. Based on this unique identity, the kernel can perform more fine-grained access decisions because two process executing the same binary that would usually have the same policy are now distinguishable.

As a result, access to container specific resources can be restrict for each container.

This mitigation strategy can help to mitigate: T4.2.

M3.5 Configure firewall rules In addition to host and container security, it is important to secure remote access to the host and containers themselves. While not going into the details of full network security in a cloud environment, each container and host must be protected as part of the security-in-depth approach. On Linux systems, netfilter/iptables and ebtables can be used to create packet firewalls. They limit network access to exposed ports on the host and in the containers. Similar to virtual switches in hypervisors, containers can be connected by a Linux bridge device. Platforms such as Docker use a default bridge device for all container if no separate network is specified during container initiation. As a result, containers may share a common network segment, which can be exploited. Therefore, ebtables is recommended as an additional firewall configured for the bridge device.

This mitigation strategy can help to mitigate: T4.3, T4.4.

M3.6 Performing host kernel and system hardening Hardening of the host system reduces the attack surface of container platforms. If it becomes more difficult for an adversary to execute exploits, threats can be mitigated. Probably the most important hardening is with respect to the Linux kernel itself because it is shared by all containers. Common practices include activating additional security features during the compilation of the kernel and patching the kernel to lock down common attack vectors. The PaX and grsecurity are two set of prominent patches.

In addition to hardening the kernel, the OS can be hardened as well by common practices such as minimizing the number of running services and installed applications. A number of minimal

GNU/Linux distributions specifically design for container hosting are available such as CoreOS Container Linux[20], Red Hat Project Atomic[21], or Rancher Labs RancherOS[22]. They only come with the necessary services to host containers while integrating into an overall CaaS architecture.

This mitigation strategy can help to mitigate: T4.2.

[20] https://coreos.com/why/
[21] https://www.projectatomic.io/
[22] http://rancher.com/rancher-os/

5 Conclusion

The introduction of containers as a virtualization technology for NFV requires a new evaluation of security threats and mitigation strategies. We have described shortly the characteristics of container-based virtualization and emphasized the fact that containers are immutable software bundles. We have continued to present a generic CaaS platform that would provide the ability to run VNFs as containers and allows for dynamic provisioning, scaling and fault-tolerance. Based on this generic CaaS platform, we have outlined a threat model and defined common attack scenarios that we believe to be principal threats. Finally, we have argued that with the appropriate mitigation strategies threats can be mitigated and that containers can actually add security value.

We recognize that our discussion may not apply to every conceivable CaaS platform architecture. For example, in our architecture the network response and throughput required for certain NFV application may be limited. This is due to the layered design of hosting containers within VMs, which adds an additional layer. However, this architecture also provides advantages from an operational and a security perspective. From the operational perspective, it becomes easier to commission machines. In addition, providers may already have an existing IaaS based on VMs. Thus, extending it to support containers as suggested by our architecture may proof simpler.

Our architecture also adds additional security layers. The VM itself can be seen as a demilitarized zone separating the boundary between tenant and provider. This gives providers the ability to enforce strict security mechanisms at lower levels. It also simplifies the separation between tenants. As discussed, containers are limited in their ability to isolate and contain. VMs add an additional isolation and separation layer.

Finally, any suspicious behavior on the VM virtualization layer is a strong indicator for malicious activity to the provider. They can use it to initiate strong countermeasures before adversaries are able to break out. For example, our architecture would permit providers to terminate VMs and replace them in case a compromised VM is likely. This is possible because the upper container layer will ensure that redundancy and fault-tolerance will be recreated.

We believe that our general discussion provides an initial starting point in developing security mechanisms for a CaaS platform for NFV. We recognize that our discussion did not cover all possible attack scenarios and mitigation strategies. We plan to extend on these in the future and generate a corresponding catalog.

Bibliography

[1] ETSI. Network functions virtualisation. White paper, ETSI, 2012. 4

[2] Steffen Gebert, David Hock, Thomas Zinner, Phuoc Tran-Gia, Marco Hoffmann, Michael Jarschel, Ernst-Dieter Schmidt, Ralf-Peter Braun, Christian Banse, and Andreas Köpsel. Demonstrating the optimal placement of virtualized cellular network functions in case of large crowd events. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 359–360. ACM, 2014. 4

[3] ETSI. Network functions virtualisation (NFV); architectural framework. Group Specification ETSI GS NFV 002 V1.2.1, ETSI, 2014. 4

[4] ETSI. Network functions virtualisation (NFV); use cases. Group Report ETSI GR NFV 001 V1.2.1, ETSI, 2017. 4

[5] Keiko Hashizume, David G. Rosado, Eduardo Fernández-Medina, and Eduardo B. Fernandez. An analysis of security issues for cloud computing. *Journal of Internet Services and Applications*, 4(1):5, Feb 2013. 12, 18

[6] CSA. The treacherous 12: Cloud computing top threats in 2016. Technical report, Cloud Security Alliance, 2016. 12, 14, 15, 16, 17, 27

[7] CSA. Security guidance for critical areas of focus in cloud computing v3.0. Technical report, Cloud Security Alliance, 2011. 12, 21

[8] Wayne Jansen and Timothy Grance. Guidelines on security and privacy in public cloud computing. NIST Special Publication 800-144, NIST, 2011. 12, 21

[9] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. Cloud computing synopsis and recommendations. NIST Special Publication 800-146, NIST, 2012. 12, 21

[10] Monowar H. Bhuyan, D.K. Bhattacharyya, and J.K. Kalita. Surveying port scans and their detection methodologies. *The Computer Journal*, 54(10):1565, 2011. 13

[11] Zakir Durumeric, Michael Bailey, and J. Alex Halderman. An internet-wide view of internet-wide scanning. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 65–78, San Diego, CA, 2014. USENIX Association. 13

[12] ENISA and George Patsis. Information package for SMEs. Technical report, ENISA, 2007. 14, 21

[13] OWASP. OWASP top 10 - 2013: The ten most critical web application security risks. Technical report, The OWASP Foundation, 2013. 14

[14] Aaron Grattafiori. Understanding and hardening linux containers. NCC Group whitepaper, NCC Group, 2016. 15, 17, 18, 21

[15] Tejun Heo and Johannes Weiner. NYLUG Presents: Tejun Heo on cgroups and cgroups v2, 2015. https://www.youtube.com/watch?v=PzpG40WiEfM. 15

[16] Michael Kerrisk. Control Groups (cgroups). Presentation, LinuxCon Europe 2016, 2016. http://www.man7.org/conf/lceu2016/cgroups-LinuxCon.eu_2016-Kerrisk.pdf. 15

[17] Chris Down. cgroupv2: Linux's new unified control group hierarchy (FOSDEM 2017), 2017. https://www.youtube.com/watch?v=XKwBtDhZ2Gc. 15

[18] Andrej Yemelianov. Containerization Mechanisms: Cgroups, 2017. https://blog.selectel.com/containerization-mechanisms-cgroups/. 15

[19] Jesse Hertz. Abusing privileged and unprivileged linux containers. NCC Group publication, NCC Group, 2016. 17

[20] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017. To appear. 18

[21] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In *14th International Conference on Financial Cryptography and Data Security (Financial Crypto 2012)*, Lecture Notes in Computer Science. Springer, 2012. 18

[22] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM. 18, 24

[23] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 368–388, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 18

[24] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016. 18

[25] Murugiah Souppaya, John Morello, and Karen Scarfone. Application container security guide (2nd draft). NIST Special Publication 800-190, NIST, 2017. 21

[26] BSI. BSI standard 100-1: Information security management systems (ISMS). Technical Report Version 1.5, BSI, 2008. 21

[27] BSI. BSI standard 100-2: IT-Grundschutz methodology. Technical Report Version 2.0, BSI, 2008. 21

[28] BSI. Cloud computing compliance controls catalogue (C5): Criteria to assess the information security of cloud services. Technical Report Version 2.0, BSI, 2016. 21

[29] Thomas Haeberlen and Lionel Duprè. Cloud computing: Benefits, risks and recommendations for information security. Technical Report Rev.B, ENISA, 2012. 21

[30] CIS. CIS Docker Community Edition benchmark (v1.1.0). CIS Benchmark, Center for Internet Security, 2017. 21

[31] CIS. CIS Kubernetes 1.6 benchmark (v1.0.0). Cis benchmark, Center for Internet Security, 2017. 21

[32] CIS. CIS Red Hat Enterprise Linux 7 benchmark (v2.1.1). CIS Benchmark, Center for Internet Security, 2017. 21

[33] CIS. CIS SUSE Linux Enterprise 12 benchmark (v2.0.0). CIS Benchmark, Center for Internet Security, 2016. 21

[34] CIS. CIS Ubuntu Linux 16.04 LTS benchmark (v1.0.0). CIS Benchmark, Center for Internet Security, 2016. 21

[35] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A Revision 1, NIST, 2015. 27

[36] Elaine Barker. Recommendation for key management, part 1: General. NIST Special Publication 800-57 Part 1 Revision 4, NIST, 2016. 27

[37] ENISA. Algorithms, key size and parameters report – 2014. Technical report, ENISA, 2014. 27

[38] BSI. Cryptographic mechanisms. Technical Guidelines BSI TR-02102, BSI, 2017. 28

[39] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 55–69, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 28

[40] Y Sheffer, R Holz, and P Saint-Andre. Summarizing known attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, IETF, 2015. 28

Fraunhofer AISEC

The Fraunhofer Institute for Applied and Integrated Security AISEC under the responsibility of Prof. Dr. Claudia Eckert is one of the leading research institutions in Europe. Fraunhofer AISEC is focused on development of application-oriented security solutions and their precise and tailored integration into existing systems. Core competences of over 90 scientific and technical members of staff lie in the areas of hardware security and the security of embedded systems, product and intellectual property protection, network security, and security in cloud- and service-oriented computing. Fraunhofer AISEC's clients operate in a variety of industrial sectors, such as the chip card industry, telecommunications, the automotive industry, and mechanical engineering, as well as the software and healthcare industries. The main goal is to support and improve the competitiveness of our clients and partners in the manufacturing and service sectors as well as those in the public sector.

The Authors

Florian Wendland is a junior researcher who started working for Fraunhofer AISEC in 2016. His current research focuses on various models of virtualization and their impact on security. In the SENDATE-PLANETS[1] project he evaluates the use of container-based virtualization for Network Function Virtualization (NFV). In addition, he develops tools that enforce security requirements during container deployment. Aside from security topics in virtualization, Florian performs security tests of network equipment for the automotive and aerospace industry. In his remaining time, he is interested in static code analysis and develops tools to discover common weakness in C programs through the application of satisfiability modulo theories (SMT).

Christian Banse is the deputy head of the department *Service and Application Security* of Fraunhofer AISEC and has been employed at the institute since 2011. Within the department he is responsible for the topics of network and communication security, especially in cloud and Software-Defined Networking (SDN) environments. Christian is also in charge of the Network and Cloud Security Lab of Fraunhofer AISEC, a test environment which allows the simulation of different network and cloud deployments and is extensively used in the evaluation of proof-of-concept implementations. His main research focus lies within the field of SDN security but also extends to other cloud and service-related areas, such as cloud service certification. He is the author of several publications in the field of communication and network security.

[1] www.sendate.eu/sendate-planets/

Fraunhofer AISEC

Fraunhofer Institute for Applied

and Integrated Security

Parkring 4

85748 Garching, Germany

www.aisec.fraunhofer.de

Phone: +49 89 322 99 86 119

E-Mail: christian.banse@aisec.fraunhofer.de