SoK: Evaluating Container Security

Harleen Kaur (40232489) || Chris V (40232485)

Objective

The rate of adoption of software containers has witnessed an unprecedented increase over the last few years. As a result, understanding and evaluating the security of containers has become the need of the hour. Through this project, we aim to explore the various aspects of container security. We look at the major risks, vulnerabilities and possible exploits in container technologies that could hinder its widespread adoption. With a strong understanding of the many threats and attacks, we develop an attack tree using the principles discussed in class.

In Section I, we start with the basics of containers, how they work, and why they have revolutionized the software industry. In Section II, we explore and understand the security mechanisms that make it possible to isolate containers from each other. In Section III, we look at the many vulnerabilities and threats to container security. In Section IV, we use the threats identified in section III to develop an attack tree.

Section I: Containerization Basics

What are containers?

To understand what containers are, it might help to understand why we need containers in the first place. Developers build an application that works perfectly fine in their own development environments. But as soon as this same piece of code is deployed into production, it throws a bunch of new errors that they never saw coming. This mostly happens due to differences in the computing environments between development and production environments – different libraries, packages, versions, and dependencies. To solve this problem, containers package the code and all its dependencies together. This container can now be run on any host without running into a bunch of dependency issues, missing libraries, and conflicts. In the words of Docker, "*a container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.*" [1]

Are containers a big deal?

Yes, they are - but why?

Firstly, containers are extremely portable as just discussed. Secondly, containers are lightweight. They share the same OS kernel and resources. On the other hand, virtual machines require an OS per application. This makes containers more efficient and lightweight [1]. As a result, a container can be started up almost instantly while a virtual machine might take several seconds to start up [2].

These features become especially relevant when it comes to the micro-service architecture [3], where a whole application is an integration of several micro-services and components, each of which is developed, tested, and deployed independently. Analysis by Mordor Intelligence LLP indicates that application container market size is expected to grow from USD 4.23 billion in 2023 to USD 15.06 billion by 2028, at a CAGR of 28.89% during the forecast period (2023-2028) [4].

Is it all moonlight and roses?

Not really. Although containers outperform virtual machines by a mile, they are less secure than VMs. This lack of security derives from the very design feature that makes containers so lightweight in the first place - the sharing of host kernel OS [5]. A 2015 study conducted by Forrester Consulting on behalf of Red Hat reported that 53% of IT operations and development decision makers at enterprises in APAC, EMEA, and North America stated that security is their biggest concern when it comes to software containerization [6]. Given the performance benefits and capabilities that containers bring to the table, these security concerns ought to be addressed urgently so that more organizations can safely reap the benefits of this technology.

Section II: Understanding Container Security

How do containers contain?

The foundation of container isolation is built on two key Linux features, namely, cgroups and namespaces.

Control Groups: cgroups limits the set of resources (CPU, memory, etc.) that a group of processes can use. This allows containers to co-habitat the same host without some containers disproportionately using up resources and starving other containers.

Namespaces: Linux namespaces control what a set of process can see. It limits the resources that are visible to a container.

Apart from these, several other key features are available to strengthen container isolation. Some of these include Seccomp (restricts systems calls that an application can make), AppArmor (restricts what executable files are allowed to do) and SELinux (restricts interactions between processes and files) [7].

Do containers really contain?

Although container platforms like Docker offer multi-layered isolation as explained, this approach is not comprehensive. Through his article titled "Containers don't contain", D. Walsh explains why. Effective isolation in a virtual machine is achieved by not allowing the VM to communicate directly with the host kernel. Different virtual machines do not share the same host kernel OS. They have their own copy of it. As a result, a process running inside the virtual machine obtaining privileges over the host kernel is highly unlikely. On the other hand, containers share the host kernel OS and hence can obtain privileged access to some kernel subsystems that are not yet namespaces [5].

Section III: Container Attack Vectors

The Application Container Security Guide published by NIST divides container attack vectors into 5 major categories. We use this categorization to kickstart our attack tree. These categories are Vulnerabilities in (i) Images, (ii) Registries, (iii) Orchestrators, (iv) Containers, and (v) Operating Systems.

Contaminating Application Images

All the code that developers write along with the required libraries, configurations, tools and dependencies are packaged into an *image.* In this section, we will look at how images might be contaminated and the consequences of deploying contaminated application images.

Images are static archive files. They are not updated or changed once they are run as containers. As a result, as time passes by, security mechanisms embedded in the image become outdated and they become vulnerable to attacks that were not discovered at the time of their creation. Image misconfiguration is also a major source of attacks in containers. For example, not specifying which user to "run as" leads to containers running with root privileges by default [8]. To prevent such situations, developers must follow the least privilege policy. Applications that make use of SSH might lead to brute force attacks. These could by prevented by enabling sshguard [9].

Another major source of attacks is untrusted images. Attackers can break into trusted repositories and place poisoned images [10]. Also, developers might sometimes use images from unofficial sources for convenience. This might lead to introduction of malware and/or leakage of data.

Secret mismanagement in applications is another major attack vector. Embedding secrets directly into the image filesystem allows anyone with access to the image to learn these secrets. Hence, secrets should be stored outside of images and only accessed when required. Most orchestration platforms offer native secret management solutions. For example, *Docker Secrets* is Docker's native secret management solution. This can be used to store passwords, SSH private keys, SSL certificates etc. [11]

Attacking Registries

A container registry is a collection of repositories which contain all the images owned/used by an organization. While public registries are unsafe due to lack of access control, private registries could be unsafe due to implementation errors [12]. One such error could be allowing insecure connections or allowing external access from outside the organization to the private registry. Insecure connections can lead to MITM attacks which can not only reveal confidential information but also sacrifice the integrity of images stored in the registry. To prevent such scenarios, organizations should make sure that all connections to the private registry are encrypted [13]. External access to private registries should be blocked by implementing firewalls.

Another common issue is the presence of outdated images in the registry. Large organizations update their applications very often. As a result, the number of images (or image versions) are always on the rise. Improper versioning can lead to accidental deployment of older vulnerable versions. To combat this, sufficient versioning control mechanisms should be in place. Pruning policies should also be put in place which facilitate manual or automatic removal of old images that are not required anymore [14][15].

Container & Orchestrator Vulnerabilities

Escape Vulnerabilities: Runtime escape vulnerabilities allow adversaries to break out of the isolation mechanisms and obtain access to the host machine [16]. These exploits are very severe and could have catastrophic consequences since it lets the adversary obtain control of host machine privileges and resources, resulting in threat not to a single container or application, but all containers and applications hosted on the same node. [16] presents a seven-class taxonomy 28 CVEs by categorizing them based on cause and impact. The three major issues that can lead to container escapes as presented in this paper are (i) Mishandled File Descriptors, (ii) Runtime components missing access control, and (iii) Adversary controlled host-execution.

Insecure Networks: Another major attack vector is through networks. Weak network defaults could give rise to very critical threat vectors since they connect containers to each other and to the host [17]. If any single container is compromised, the network can be used by the adversary to obtain access to other containers or the host itself.

Most container orchestration platforms use a virtual overlay network to enable communication between different nodes as explained in [13]. Since the overlay network is managed by the container orchestrator, the content of these networks is not visible to the client's network security tools. Although this invisibility creates an impression of security, its impact is quite the opposite. Network filters, that would have otherwise captured malicious packets flowing between different hosts, are now unable to filter data as it is invisible to them. As a result, organizations are unable to observe data traffic in their own networks.

A more serious problem is the same virtual network being shared by different applications. The overlay network used by the orchestrator to connect different nodes is not application specific. As a result, different applications that have different sensitivity levels end up sharing the same virtual network. This enables attackers to target less secure applications and then use this shared network to obtain access to more secure applications.

Improper Access Control: In addition, improper implementation of access control on the orchestration software could open several threat vectors. It is important to carefully consider which teams have access to the orchestration software. Ineffective access control to the orchestration software can lead to inadvertent errors that might create security vulnerabilities.

Rogue Containers: Lastly, improper removal of containers (or daemon malfunction) could result ghost processes that may not visible in the inventory but still consumes resources. Another risk is test containers that were not removed on a timely basis. This could be due to developers testing on code the production environment or because of loose separation between development and production environments.

Operating System Vulnerabilities

Shared Kernel: Containers, unlike virtual machines, share the same kernel. As a result, kernel exploits executed inside the container can compromise the host kernel [18]. For example, CVE-2022-0492 is a vulnerability that targets a weakness in the handling of release_agent in cgroups. This exploit resulted in a container escape [19].

Another major Linux kernel vulnerability named *Dirty Pipe* (CVE-2022-0847) allowed attackers to write to read-only files. As a result, attackers gained the ability to modify images in the host from within the container [20]. Many other similar vulnerabilities have been discovered over the past few years.

Insecure access rights in container-specific OSs: Container-specific OSs often do not include multi-user capabilities. As a result, users directly log on to hosts to manage containers without an orchestration layer in the middle. Such access gives a wide range of administrative capabilities to the user to alter the state of the host and containers hosted on it [13].

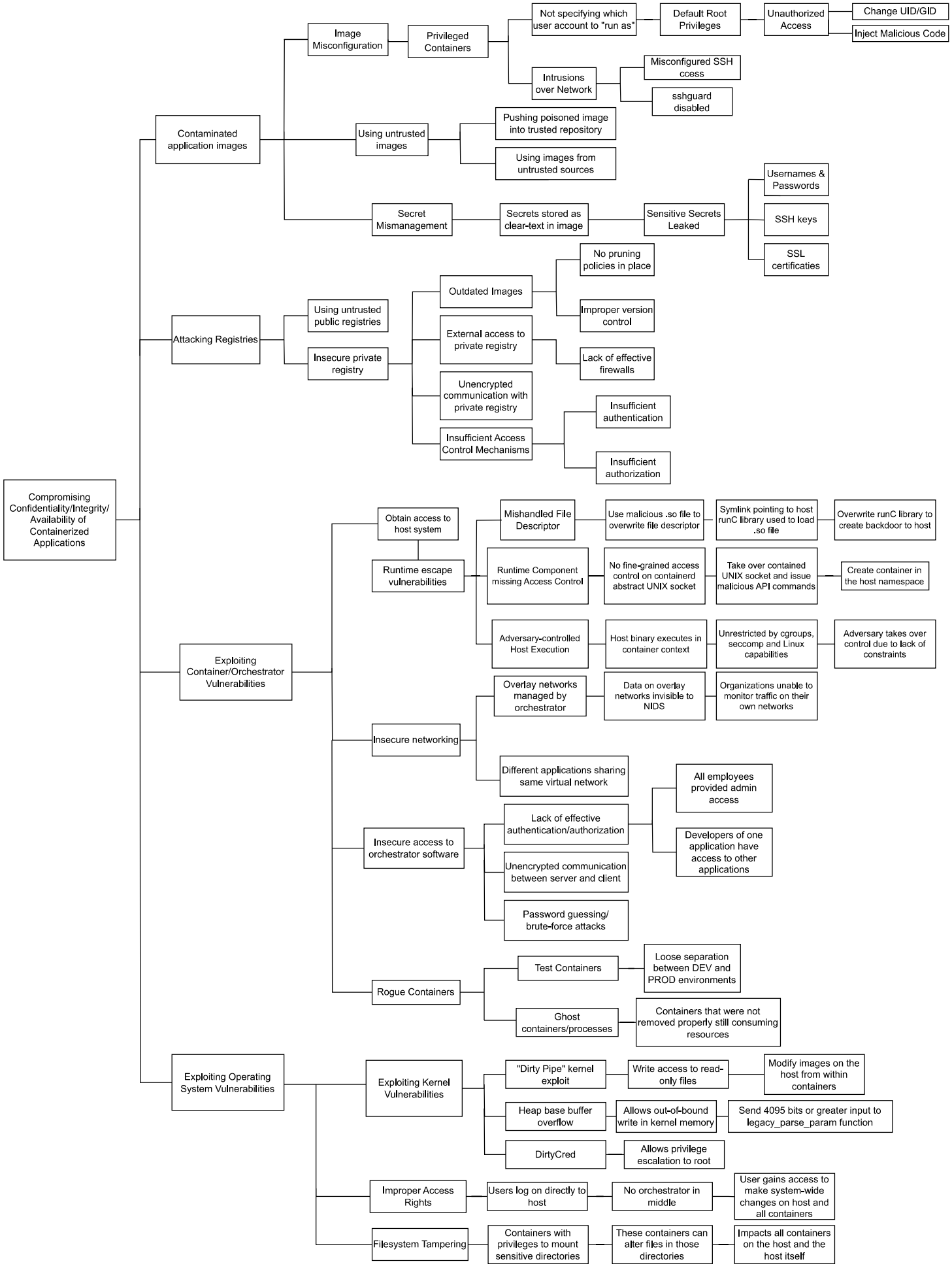
Filesystem Tampering: Containers that are not configured correctly might expose host volumes to file tampering. A container that has privileges to mount sensitive directories on the host OS can tamper files in those directories. Such tampering impacts all containers and the host itself.

Section IV: Attack Tree

Based on the threats, vulnerabilities and attacks discussed in section III, we developed an attack tree. This attack tree is not exhaustive, but includes most major threats and attacks on confidentiality, integrity and availability of containerized applications.

Conclusion

Through this project, we were able to develop a thorough understanding of various security issues in containerized applications. We looked at many different threats that could be used to compromise container security. Based on the various security threats we explored, we developed an attack tree. Through this exercise, we were able to further our understanding of different security evaluation methodologies as well as learn a lot about containerization.


Compromising Confidentiality/Integrity/ Availability of Containerized Applications
Contaminated application images
Image Misconfiguration
Privileged Containers
Not specifying which user account to "run as"
Default Root Privileges
Unauthorized Access
Change UID/GID
Inject Malicious Code
Intrusions over Network
Misconfigured SSH ccess
sshguard disabled
Using untrusted images
Pushing poisoned image into trusted repository
Using images from untrusted sources
Secret Mismanagement
Secrets stored as clear-text in image
Sensitive Secrets Leaked
Usernames & Passwords
SSH keys
SSL certificaties
Attacking Registries
Using untrusted public registries
Insecure private registry
Outdated Images
No pruning policies in place
Improper version control
External access to private registry
Lack of effective firewalls
Unencrypted communication with private registry
Insufficient Access Control Mechanisms
Insufficient authentication
Insufficient authorization
Exploiting Container/Orchestrator Vulnerabilities
Obtain access to host system
Runtime escape vulnerabilities
Mishandled File Descriptor
Use malicious .so file to overwrite file descriptor
Symlink pointing to host runC library used to load .so file
Overwrite runC library to create backdoor to host
Runtime Component missing Access Control
No fine-grained access control on containerd abstract UNIX socket
Take over contained UNIX socket and issue malicious API commands
Create container in the host namespace
Adversary-controlled Host Execution
Host binary executes in container context
Unrestricted by cgroups, seccomp and Linux capabilities
Adversary takes over control due to lack of constraints
Insecure networking
Overlay networks managed by orchestrator
Data on overlay networks invisible to NIDS
Organizations unable to monitor traffic on their own networks
Different applications sharing same virtual network
Insecure access to orchestrator software
Lack of effective authentication/authorization
All employees provided admin access
Developers of one application have access to other applications
Unencrypted communication between server and client
Password guessing/ brute-force attacks
Rogue Containers
Test Containers
Loose separation between DEV and PROD environments
Ghost containers/processes
Containers that were not removed properly still consuming resources
Exploiting Operating System Vulnerabilities
Exploiting Kernel Vulnerabilities
"Dirty Pipe" kernel exploit
Write access to read-only files
Modify images on the host from within containers
Heap base buffer overflow
Allows out-of-bound write in kernel memory
Send 4095 bits or greater input to legacy_parse_param function
DirtyCred
Allows privilege escalation to root
Improper Access Rights
Users log on directly to host
No orchestrator in middle
User gains access to make system-wide changes on host and all containers
Filesystem Tampering
Containers with privileges to mount sensitive directories
These containers can alter files in those directories
Impacts all containers on the host and the host itself

References

[1] What is a Container? | Docker

[2] K. Kaur, T. Dhand, N. Kumar and S. Zeadally, "Container-as-a-Service at the Edge: Trade-off between Energy Efficiency and Service Availability at Fog Nano Data Centers," in IEEE Wireless Communications, vol. 24, no. 3, pp. 48-56, June 2017, doi: 10.1109/MWC.2017.1600427.

[3] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni and C. Fetzer, "SGX-Aware Container Orchestration for Heterogeneous Clusters," 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2018, pp. 730-741, doi: 10.1109/ICDCS.2018.00076.

[4] Application Container Market Size & Share Analysis - Growth Trends & Forecasts (2023 - 2028) (reportlinker.com)

[5] D. Walsh, "Are Docker containers really secure?", opensoure.com, 2014 (Are Docker containers really secure? | Opensource.com)

[6] A. Bettini, "Vulnerability exploitation in docker container environments", FlawCheck, Black Hat Europe, 2015 (BHEU-Bettini-Docker-Exploitation (blackhat.com))

[7] L. Rice, "Container Security: Fundamental Technology Concepts that Protect Containerized Applications", published by O'Reilly Media, 2020

[8] R. Campuzano, “Why non-root containers are important for security”, docs.bitnami.com, 2018 (Why non-root containers are important for security (bitnami.com))

[9] Blocking SSH brute force attackes on Docker containers with sshguard (cmu.edu)

[10] Kubernetes Attacks: How Untrusted Docker Images Fail Us | Optiv

[11] Manage sensitive data with Docker secrets | Docker Docs

[12] What is a Container Registry security? | Snyk

[13] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," NIST Special Publication, vol. 800, p. 190, 2017.

[14] Registry Risks: Gaining Visibility into NIST SP 800-190 Part 4 | Optiv

[15] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities," Technical report, BanyanOps, Tech. Rep., 2015.

[16] M. Reeves, D. J. Tian, A. Bianchi and Z. B. Celik, "Towards Improving Container Security by Preventing Runtime Escapes," 2021 IEEE Secure Development Conference (SecDev), Atlanta, GA, USA, 2021, pp. 38-46, doi: 10.1109/SecDev51306.2021.00022.

[17] A. Grattafiori, “Understanding and Hardening Linux Containers”, NCC Group, 2016

[18] Docker Security - OWASP Cheat Sheet Series

[19] New Linux Kernel Vulnerability: Escaping Containers by Abusing Cgroups (aquasec.com)

[20] Dirty Pipe Linux Vulnerability: Overwriting Files in Container Images (aquasec.com)