Container Security : Enhancement of Integrity and Availability of a Containerized Environment

Jugal R Patel
department of computer science and engineering
National Institute of Technology Calicut
Kozhikode, Kerala 673601, India
jugal_m220275cs@nitc.ac.in

Priya Chandran
department of computer science and engineering
National Institute of Technology Calicut
Kozhikode, Kerala 673601, India
priya@nitc.ac.in

***Abstract*—Container Configuration Monitoring Containerization is the best virtualization practice for maximizing the utilization of a machine's hardware (such as RAM, Storage, CPU, etc.) over virtual machines for the development and deployment of software applications. However, despite its advantages, it also presents certain security threats and drawbacks, which raise concerns about using containerization techniques. Therefore, to mitigate these security risks and enhance the security of containerization, identifying and addressing these vulnerabilities is imperative. This paper presents literature on various container vulnerabilities, native protection mechanisms for containers, and issues related to container volumes. Additionally, we have proposed CHM (Container Hash Monitoring) and CCM (Container Configuration Monitoring) solutions for detecting and preventing these vulnerabilities from harming containers. The code for these solutions can be found at the https://github.com/JugalPatel01/container_security link.**

***Index Terms*—virtualization, containers, container runtimes, security attacks, container security mechanisms, container volumes.**

I. INTRODUCTION

As the demand for scaling social media applications continues to rise, many are transitioning from monolithic architecture to microservices architecture. However, prior to the widespread adoption of virtualization technology, the traditional approach to implementing microservices involved running applications on individual servers and each server was allocated to a specific services to achieve the highest level of security. However, dedicating a new server for every new service can be costly and inefficient, as it fails to fully utilize the server's resources. As a solution, virtualization has emerged to address this inefficiency.

Virtualization is the process of creating isolated virtual instances of a physical computer, allowing multiple operating systems to run on a single physical machine. This is achieved through the emulation of both hardware and software. There are **two types** of virtualization. **1) hypervisor-based virtualization and 2) container-based virtualization** [1]**.**

While container-based virtualization is highly efficient for deploying applications, yet it is often considered less secure due to its shared kernel, weaker isolation, potential for privilege escalation, and complex internal networking among multiple containers [6]. Consequently, security is the primary obstacle to the widespread adoption of containers. Therefore, studying vulnerabilities in containerization environments and implementing effective solutions is of paramount importance for security. In this regard, we discuss virtualization and containerization in Section II. In the Section III, we explore various vulnerabilities and their categories in containerized environments. Furthermore, in the Section IV, we discuss the available Linux native solutions for safeguarding against these attacks. In Section V, we examine container volumes. Finally, we explore future research directions in this area, followed by our conclusion in the last section.

This work is supported by Ministry of Education and SERB, Government of India, through M.Tech stipends and CRG respectively.

II. BACKGROUND

In this section, We provide background information on virtualization and containers. Section II-A offers details about virtualization and hypervisor-based virtualization, while Section II-B provides information on container-based virtualization.

A. Hypervisor-based virtualization

In the Section I, we explored the significance of virtualization and how hypervisor-based virtualization addresses various challenges. Let's now take a closer look at how this hypervisor-based virtualization works.

Type 1 hypervisors (bare-metal hypervisors)

In Type 1 hypervisor-based virtualization, a Type 1 hypervisor is directly installed on the underlying hardware, typically on servers without the presence of any pre-existing operating system [2] (as shown in Figure 1 [4]). Examples of Type 1 hypervisors include VMware ESXi, Citrix XenServer, and Microsoft Hyper-V. This hypervisor serves as a lightweight, low-level software layer that takes on the role of a Virtual Machine Monitor (VMM). It is responsible for creating, managing and running virtual machines. Above the hypervisor, multiple virtual machines (VMs) are deployed, each equipped with its dedicated operating system, ensuring a strong level of isolation between VMs. Applications are executed on top of the operating systems, resulting in a layered and efficient computing approach.

Type 2 hypervisors (hosted hypervisors)

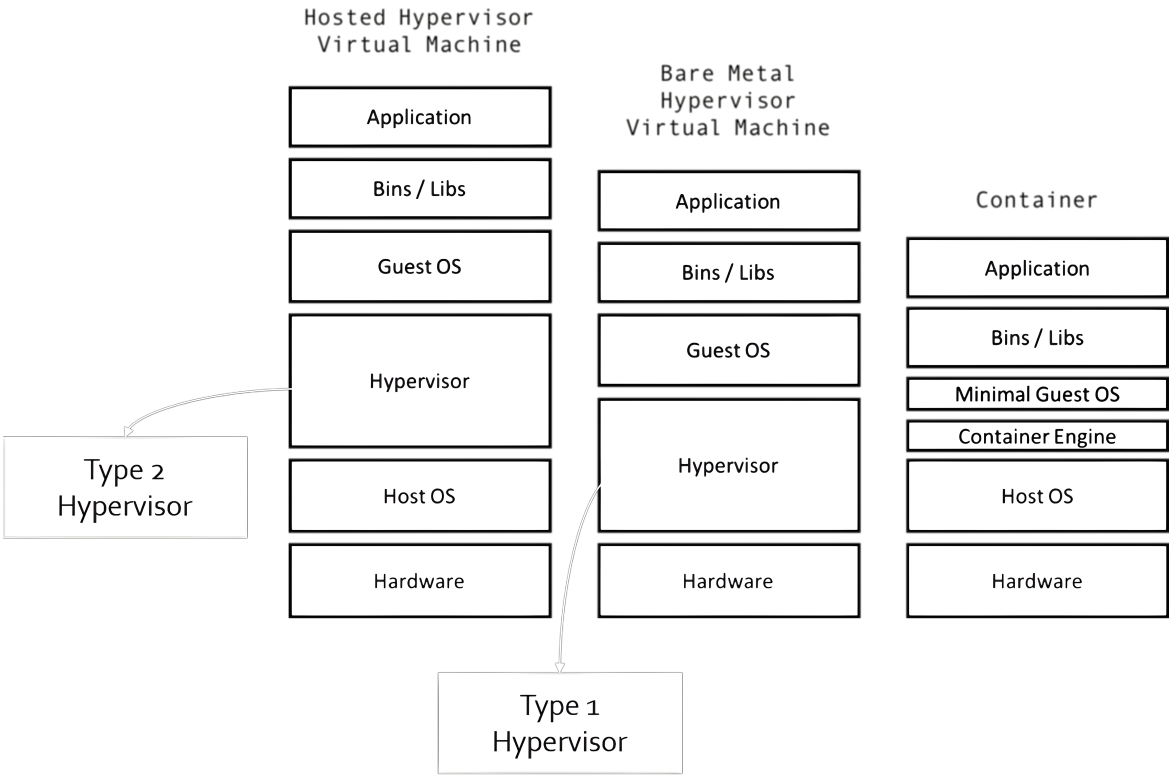

Fig. 1. Comparison between Type 1, Type 2 hypervisors and containers.

Type 2 hypervisors are designed to run on top of an existing host operating system (OS) rather than directly on the hardware [2]. Examples of Type 2 hypervisors include Oracle VM VirtualBox, Microsoft Virtual PC, and VMware Workstation. In this configuration, The host OS serves as an intermediary layer between the hardware and the hypervisor, providing an interface for managing resources, as shown in Figure 1. This hypervisor, situated atop the host OS, takes on the role of a Virtual Machine Monitor (VMM). While Type 2 hypervisors offer the advantage of being easy to set up and use on existing hardware, they do introduce some performance overhead and security concerns due to the presence of the host OS. This additional layer can potentially diminish VM performance and security compared to Type 1 hypervisor-based virtualization [7]. These hypervisors are often used on regular servers and desktop computers for testing, development, or virtualizing non-production environments.

Benefits and drawbacks of hypervisor-based virtualization

Virtual machines (VMs) are an excellent choice for running legacy applications that rely on specific OS versions or configurations that might not be compatible with the host OS. They also provide portability, allowing for easy migration to another physical machine, and facilitate disaster recovery [2]. However, VMs have their drawbacks. They can be resource-intensive, consuming significant disk space, RAM, and CPU power. This results in slower startup times due to the presence of complete operating systems within each VM [8] [10]. Additionally, for running applications utilizing microservices (such as social media applications), there's often no need for multiple operating systems for each service. To overcome the challenges associated with running multiple operating systems and to improve resource utilization, containers offer a promising solution.

B. Container-based virtualization (containerization)

As we have previously discussed the need for container-based virtualization in the "Drawbacks of Virtualization" subsection, let's now explore containerization.

container-based virtualization provides a superior alternative to virtual machines. While VMs require a dedicated copy of the operating system kernel for each instance, containers allow multiple deployed application to share a common OS kernel, resulting in faster startup times and more efficient resource utilization [3]. Examples of container technologies include LXC, OpenVZ, Linux-Vserver, and Docker.

Before containerization, developers had to install and configure most services directly on their local machine's operating systems for development. Containers provide a superior solution to this problem. They eliminate the need to install services directly on the host OS by providing each container with an isolated operating system layer using a Linux-based image [9]. This eliminates conflicts and allows running the same application with different versions. Containers bundle everything an application's service needs to run, making them highly portable and easily shareable. Unlike virtual machines,

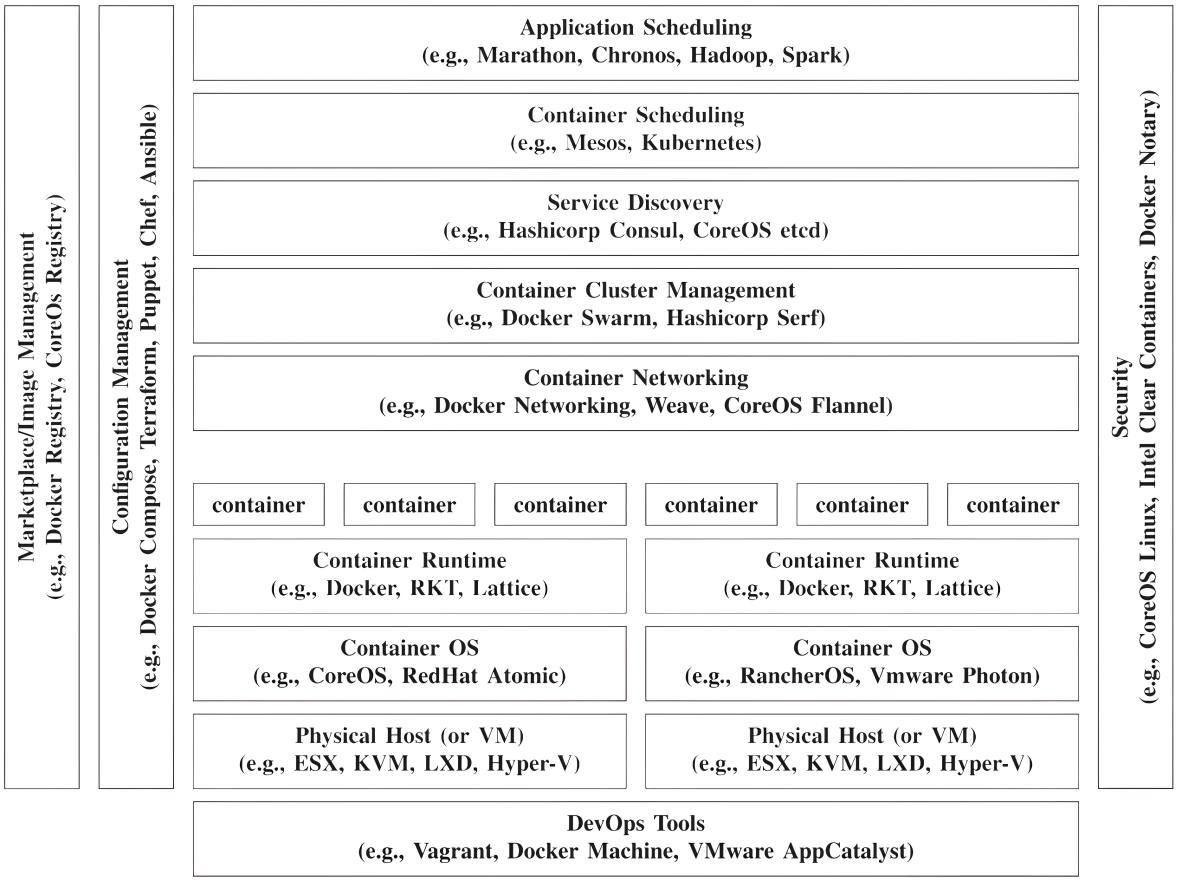

Fig. 2. Container stack components and realization technologies

which simulate entire machines, containers are self-contained applications bundled with all the necessary files, configurations, and dependencies. This portability and isolation offer efficiency in both development and deployment processes.

Containerization begins with the underlying hardware, and on top of that, there is the host operating system. Instead of using a hypervisor, containers utilize a container engine, also known as container runtime or container plateform [4]. There are four types of container runtimes :

1) **Low-level container runtimes:** This level deals directly with the Linux kernel to create and manage containers. Runc, crun, and railcar are the most common examples of low-level container runtimes [5].
2) **Middle-level container runtimes:** This level extends beyond the features of low-level runtimes by providing additional functionalities such as image distribution, storage management, and network configuration. Containerd, CRI-O, and rkt are widely used middle-level runtimes that serve as an interface between higher-level orchestration platforms and the operating system.
3) **High-level container runtimes:** This level provides user-friendly interfaces and tools, simplifying containerization and application deployment. Docker Engine and Podman are popular high-level container runtimes.
4) **Sandboxed/Virtualized container runtimes:** This level enhances security by running containers within lightweight VMs. Kata Containers, gVisor, and runV are popular virtualized container runtimes. They can utilize either containerd or runc as their OCI runtime, depending on the configuration, while leveraging VMs for stronger security boundaries [5].

This container engine is responsible for running and managing containers, performing functions like creating, starting, and stopping containers on a host system. It unpacks container image files and hands them off to the core of the operating system called **"kernel"**. Multiple containers are deployed on top of this engine, and each container has its local OS layer, as shown in Figure 1. Container engines interact with the host kernel to allocate resources and enforce isolation between containers through two crucial mechanisms [6] :

1) **Namespaces:** Namespaces restrict a container's access and visibility to other resources on the system, ensuring that each container has its isolated environment [6]. Key namespace types include Mount, PID (Process Identifier), Network, User Namespaces. Further details on namespaces are provided in Section IV-A(a).

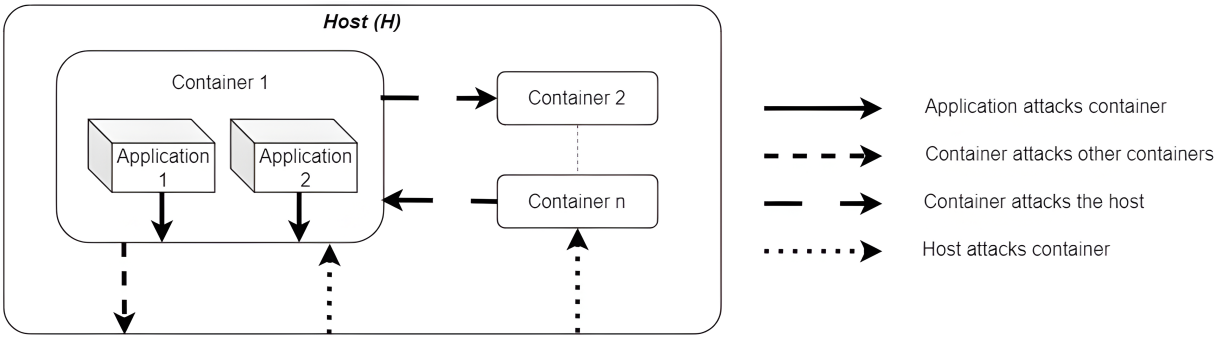

Fig. 3. Container-Host level security threats.

2) **Control Groups (cgroups):** Control Groups are a feature of the Linux kernel that isolates and controls the resource usage for user processes. These processes can be put into namespace, which are collections of processes sharing the same resource limitations. Each namespace has specific resource properties enforced by the kernel, allowing for fine-grained control of resources like CPU and RAM [4]. cgroup isolation is a higher level of isolation that makes sure processes within a cgroup namespace are independent of processes in other namespaces. Further details on cgroups are provided in Section IV-A(b).

This combination of container engines, namespaces, and cgroups provides the foundation for container isolation, making sure that containers operate independently and securely on the same host. Each container has its own isolated environment, allowing applications to run consistently and securely across different computing environments. **But host OS kernel sharing introduces many security issues, which make them less secure than VMs [12]**.

The panoramic perspective of container technologies is important as deployment of the applications relies on various components of it. Figure 2 [3] provides an overview of container stack components and associated technologies available in the market for facilitating microservices deployment, derived from the architecture depicted in [11].

Researcher Peinl *et al.* surveyed the tools mentioned in Figure 2, which are used for container management [13]. They classified various solutions found in both academic and industry literature, mapping them to requirements based on the case study they provided. Through that analysis, they identified gaps in these tools and integration requirements. To address these deficiencies, they proposed their own tools. As indicated in their research paper [13], gaps in container architecture exist, and the adoption of evolving container architecture may raise security concerns.

In containerized applications, various services like front-end, back-end, and databases are divided as per microservices' requirements and deployed on specific containers. When there is a higher server load on one service, an additional container can be deployed for that specific service, and load balancing can be performed. This scalability is a primary advantage of containerization. However, it also introduces security challenges. Containers share the same host OS kernel, leading to numerous security breaches that adversaries can exploit to potentially take over the host machine or steal passwords. Therefore, identifying these threats and vulnerabilities, studying them, and resolving them is imperative. The following section provides a threat classification for containers.

III. THREAT CATEGORIZATION

The security of any organization begins with three principles: confidentiality, integrity, and availability [14]. These three principles are referred to as the security triangle, or CIA, which has served as the standard for systems security since the first computer systems.

The principle of confidentiality states that only an authorized user can access sensitive information and functions. Example: encrypting data to prevent unauthorized access. The principles of integrity claim that only authorized individuals and resources can modify, add, or remove sensitive information and functions, and it guarantees that information is accurate and hasn't been tampered with. Example: using a digital signature to ensure the authenticity of software updates. Availability principles claim that systems, functions, and data must be available on demand upon agreed parameters based on SLA (service level agreement). Example: Implementing redundancies to ensure website uptime even during server maintenance.

To enhance security, we need to safeguard above-mentioned triad from malicious activities. Securing this triad for containers by creating an exhaustive list of possible threats and following that list would be frustrating and challenging to track. Therefore, Sari Sultan *et al.* [3] categorized all threats listed into [15] into 4 different host-container level categories, as shown in Figure 3 and below :

1) Protecting a container from applications inside it
2) Inter-container protection
3) protecting the host (and the applications inside it) from containers
4) protecting containers from host

According to different categories, there are multiple threats and attacks possible on the containerized environment. Some

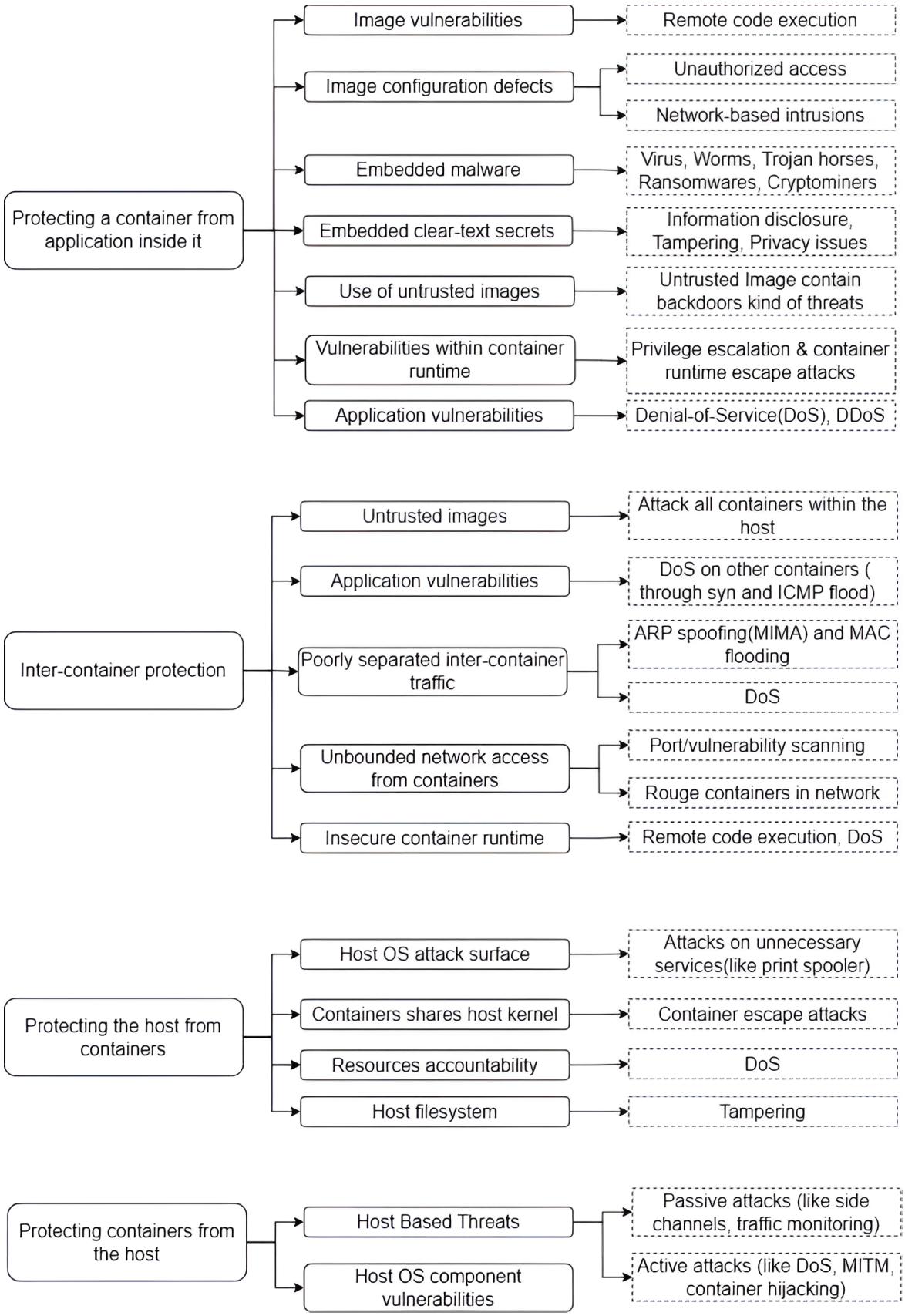

Fig. 4. Detailed categorization of Container-Host level security threats.

of them are shown in Figure 4. Souppaya *et al.* [15] have provided a detailed explanation of these threats and suggested precautions and solutions to mitigate these vulnerabilities. In the next section, we will explore how some native Linux solutions and other solution help to resolve some of the above issues.

IV. CONTAINER PROTECTION MECHANISMS

Containers are just processes [16]. Therefore, they inherit all the security features available to processes through the Linux kernel, such as linux security features and modules. In the Figure 5, we can observe the process level architecture of container :

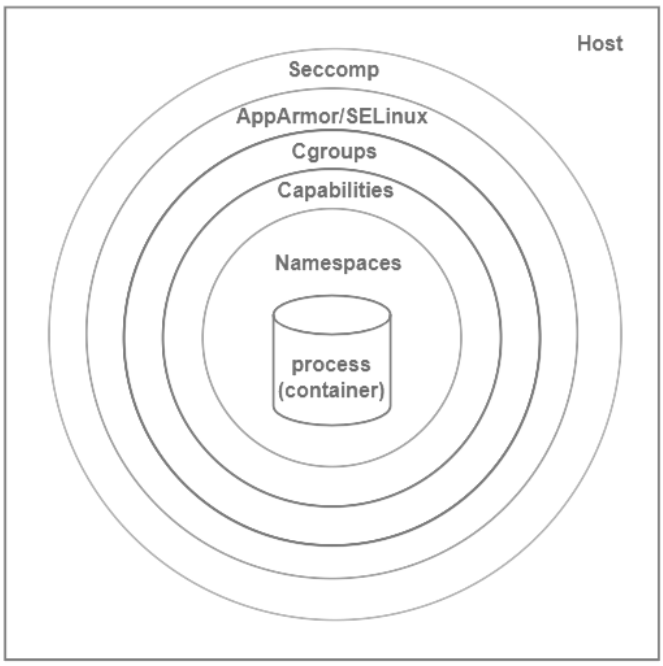

Fig. 5. Process level architecture of container.

To mitigate vulnerabilities and attacks on containers, there are two primary mechanisms are available :

1) Software based protection mechanisms :
   - Linux security features
     a) Namespaces
     b) Control groups
     c) Capabilities
     d) Secure computation mode
   - Linux security modules
     a) AppArmor
     b) Security-Enhanced Linux (SELinux)
     c) LoadPin
     d) YAMA
     e) Smack
     f) TOMOYO
2) Hardware based protection mechanisms :
   - Hardware Security Modules(HSMs)
   - Trusted Plateform Modules(TPMs)
   - Secure Enclaves
     a) Intel SGX
     b) ARM TrustZone

A. Software-based protection mechanisms :

Linux kernel's security features :

a) Namespaces :

Namespaces wraps global resources to provide resource isolation and virtualization(isolation of processes) for a collection of processes [17] [3]. They function as dividers of the identifier tables and other structures associated with kernel global resources into isolated instances. Linux currently supports a total of 8 namespace types, as shown in Figure 6 [17], and each namespace has multiple instances on a single system. Every process resides in one namespace instance of the 8 different namespace types. Hence, each container can have a unique view of resources. Namespaces ensure the isolation of processes running in a container, preventing them from observing other processes running in a different container [19]. However, one issue with namespaces is that some resources, such as input and output devices, are still not namespace aware [18].

Namespace	Constant used for system call APIs	It isolates
Mount(mnt)	CLONE_NEWNS	Set of mount points
UNIX time-sharing(uts)	CLONE_NEWUTS	System hostname & NIS domain name
Inter-process communication(ipc)	CLONE_NEWIPC	System V IPC(message queues (MQs), semaphores, shared memory), POSIX MQs
Process id(pid)	CLONE_NEWPID	Process Id number space
Network(net)	CLONE_NEWNET	IP addresses, IP routing tables, netfilter(firewalls),socket port-number space, abstract UNIX domain sockets
User id(uid)	CLONE_NEWUSER	User & Group ID number spaces
Control groups(cgroups)	CLONE_NEWCGROUP	cgroups(manage & allocate system resources like CPU, memory)
Time	CLONE_NEWTIME	time-related resources(CPU time, timers)

* All the namespaces are listed in the order they were introduced in the LINUX kernel

Fig. 6. Types of Namespaces

Use of namespaces in containers :

- ***Mount namespace*** isolates a set of mount points, allowing each individual containers to have its own private filesystem tree.
- ***UTS namespace*** isolates the nodename (system hostname) and domainname for each container. Container configuration scripts might utilize these hostnames and domain names. For example, a container might broadcast its hostname to DHCP to obtain a unique IP address.
- ***PID namespace*** isolates the Process ID number space, enabling processes inside containers to maintain the same PIDs when the container is migrated to a different host. It also allows for a per-container init process (PID 1) that manages container initialization. Additionally, by using PID namespace, a container can be restricted to see only its own processes rather than the host's processes.
- ***Network namespace*** isolate system resources associated with networking, enabling each container to have its own virtual network device. Routing rules of the host system can direct network packets to these virtual devices, facilitating networking between different containers.
- ***UserID namespace*** isolate user and group ID number space. So, container processes have different UIDs and GIDs inside and outside the user namespace. For example, outside the container may have a non-privileged UID, while inside the container, its UID is 0, making it a superuser within the container. This mechanism helps restrict the privileges of containers.

Solutions that use namespaces :

- Jian *et al.* [20] investigated container escape attacks in Docker. They introduced a defense technique aimed at mitigating escape attack by inspecting namespace statuses during process execution. Their goal is to identify compromised containers that have gained root access and attempt to modify the namespace. As a solution, they proposed a method that detects namespace statuses and flag any changes that could indicate a compromised container.
- Gao *et al.* [21] explored information leakage (side) channels within containers. They address channels that could leak host information, which allow adversaries to launch advanced attacks against cloud service providers. So, the authors proposed a two-stage defence system. They implemented a power-based namespace to offer fine-grained control at the container level regarding resource consumption.

b) Control Groups :

Cgroups are Linux features that control the accountability and limitation of resource usage, such as CPU runtime, system memory, I/O, and network bandwidth [3]. A cgroup constitutes a collection of processes bound to specific limits established through the cgroup filesystem. A subsystem, also known as a resource controller, modifies the behavior of processes in a cgroup. Various subsystems have been implemented for tasks like limiting CPU time and memory, tracking CPU time usage, and controlling the freezing and resuming of process execution [22].

The cgroups for a controller are organized in a hierarchy, which is defined by creating, removing, and renaming subdirectories within the cgroup filesystem. At each level of the hierarchy, attributes (e.g., limits) can be defined. The limits, control, and accounting provided by cgroups generally have effect throughout the subhierarchy underneath the cgroup where the attributes are defined. Therefore, for example, the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups.

Use of cgroups in containers :

- As Cgroups constrain permitted resource usage, containers are unable to exceed the designated resource limit. This restriction helps protect other containers from various attacks, such as Denial-of-Service (DoS) attacks [23].
- Cgroups give the host the power to not only limit resource usage but also account for how much of the resource is used. This capability facilitates the implementation of usage quotas, which is a crucial aspect in the cloud computing delivery model, Container as a Service (CaaS). Some cloud providers, such as AWS, utilize cgroups as a layer for tracking resource usage [24].

Solutions that use cgroups :

- Chen *et al.* [23] introduced a real-time protection mechanism against DoS attacks called ContainerDrone. Initially, the system utilizes CGroups to allocate a specific set of cores to each task for CPU DoS protection. It also uses Docker features to restrict a process from raising its priority. However, regarding memory DoS protection, the authors found through experimentation that CGroups are not entirely effective. Although they can restrict the allocated memory bandwidth, malicious applications could still exploit intensive memory access within that limit. Therefore, to protect memory from DoS attacks, the researchers used a MemGuard Linux kernel module to restrict memory access for each CPU core.

c) Capabilities :

For the purpose of performing permission checks, UNIX implementations categorize processes into two groups: privileged (superuser or root) and unprivileged. Privileged processes bypass all kernel permission checks, while unprivileged ones undergo full permission checking based on credentials such as effective UID, effective GID, and supplementary group list. However, in the context of containers, this categorization can be troublesome. For example, a web server (e.g., Nginx) needs to bind on a specific port (e.g., TCP port 80). To accomplish this task, the web server process should have root access, which poses significant security risk because if it gets compromised, an attacker will be able to control the entire system.

To address this issue, from Linux 2.2, Linux divides superuser privileges into distinct units called capabilities, which

can be independently enabled and disabled [25]. This changes the privileged and unprivileged dichotomy into fine-grained access control [19]. Consequently, containerized processes within containers typically do not require full root privilege, as long as there is an available capability for the required tasks. The Linux kernel supports close to 40 different capabilities [26].

Capabilities play an essential role in protecting containers from malicious applications running inside them. As mentioned above in the web server example, a container can be assigned the process the CAP_NET_BIND_SERVICE capability (used for binding a socket to a privileged port ¡ 1024), allowing it to run a version of the web server without requiring full privileged root access. Suppose the web server contains vulnerabilities that are exposed to an adversary; in that case, the adversary can exploit only the capability assigned to the container. Now, suppose a container process becomes malicious, posing a direct risk to the host. So, by setting capabilities to a container, it can reduce the container's privileged root operations. However, restricting capabilities alone does not guarantee security. For example, suppose we give a KILL capability to a container to stop some application. However, if this container becomes malicious, it can affect the availability of other containers.

d) Secure Computation Mode (Seccomp) :

The seccomp system call functions within the Secure Computing (seccomp) state of the calling process. It filters system calls to the kernel. Seccomp offers fine-grained control by allowing different seccomp profiles to be applied to different filters. This feature helps decrease the number of potentially risky system calls made by containers. For example, the default seccomp profile in Docker disables around 44 out of 300+ system calls, providing a sensible default for running containers with seccomp [27]. These proactive measures can further reduce possible threats, as many attacks exploit kernel vulnerabilities through system calls.

Wan *et al.* [28] introduced a solution for sandboxing containers through automatic testing. During the testing phase, the proposed solution extracts the system calls utilized by the container during execution. Using seccomp, the solution creates a profile for each application based on the observed system calls during testing. However, a potential issue arises if vulnerable containers access exploitable system calls observed during the testing phase. In such cases, generating a profile for these vulnerable containers becomes meaningless.

Linux Security Modules :

Linux security modules(LSMs) enable a wide variety of security models to be implemented on Linux kernel as loadable modules [29]. This flexibility allows users to choose their preferred security model instead of being forced to use the default one that comes with the operating system. LSMs aim to fulfill the requirements for implementing Mandatory Access Control (MAC) with minimal changes to the kernel itself. It's important to note that LSMs are shared among all containers, meaning that each container cannot have its own specific LSM [30].

a) AppArmor :

AppArmor is a MAC (mandatory access control)-style security extension for the Linux kernel. It implements a task centered policy, where task "profiles" are created and loaded from user space [33]. By confining container processes within predefined policies, AppArmor enhances container security, limiting their access to system resources. It enforces the principle of least privilege, preventing unauthorized actions and mitigating potential vulnerabilities.

b) Security-Enhanced Linux (SELinux) :

SELinux provides administrators with a comprehensive mandatory access control mechanism for containers, offering greater access granularity compared to existing Linux Discretionary Access Controls (DAC) [34]. For example, instead of having read, write, and execute permission, SELinux allows specifying unlink, append only, move a file, and other permissions.

c) LoadPin :

LoadPin is a Linux Security Module that ensures all kernel-loaded files (modules, firmware, etc.) originate from the same filesystem, with the expectation that such a filesystem is backed by a read-only device like dm-verity or CD-ROM [35]. This enables systems with verified and/or unchangeable filesystem to enforce module and firmware loading restrictions without need to individually sign the files. LoadPin enhances container security by providing a tamper-proof seal, detecting any unauthorized access or tampering during transit.

d) YAMA :

Yama is a Linux Security Module that collects system-wide discretionary access control (DAC) security protections that are not handled by the core kernel itself [36]. It enhances container security by restricting the ptrace system call, preventing the debugging of processes outside the container.

e) Smack :

Smack (Simplified Mandatory Access Control Kernel) enhances container security by enforcing strict access controls based on labels, preventing unauthorized access to system resources [38]. It isolates containers from each other and the host system, reducing the risk of privilege escalation and unauthorized data access.

f) TOMOYO :

TOMOYO is a name-based MAC extension (LSM module) for the Linux kernel [37]. It restricts process based on defined policies. Tomoyo offers auditing capabilities, providing insights into container activity for improved security monitoring and incident response.

Solutions that use LSMs :

- Loukidis-Andreou *et al.* [31] introduced Docker-sec, an automated system designed to enhance Docker container security by leveraging the AppArmor LSM. Docker-sec

establishes a static set of access rules during image creation based on parameters and enhances them during runtime to further restrict threats. The authors claim that Docker-sec can automatically protect against zero-day vulnerabilities with minimal performance overhead [3].
- MP *et al.* [32] studied the impact of various LSMs and LSFs on Docker containers. They examined SELinux, AppArmor, and TOMOYO and demonstrated, through a proof of concept, that these LSMs are effective in mitigating several risks, including malicious code and bugs in namespace code.

B. Hardware-based protection mechanisms :

Virtual Hardware Security Modules(HSMs) :

A Hardware Security Module (HSMs) is a physical computing device designed to securely stores and manage cryptographic keys for various security purpose, including authentication (access control mechanisms), authorization (data signing), data confidentiality (encryption and decryption) and data integrity [39]. However, with the rise of cloud computing and virtualization, challenges such as scalability, agility, and adaptability have emerged for hardware HSMs [40]. As a result, the need for a virtual hardware security module (vHSM) arises. A vHSM is a software-based solution that replicates the functions of HSM in a virtual environment. It provides key management and cryptographic services in a containerized environment and can be utilized to run a broad range of trusted applications. Yehuda has presented detailed information about vHSMs [40].

Virtual Trusted Platform Modules(vTPMs) :

Trusted Platform Modules (TPMs) are hardware components designed to serve as cryptographic processors. TPMs provide support for various functions such as attestation, data sealing, secure boot, and algorithm acceleration. A detailed survey about TPMs is presented by Kenneth *et al.* [41]. As the use of cloud computing, virtualization, and CaaS increases, the need for virtual Trusted Plateform Modules(vTPMs) arises because these virtualization techniques need to make the TPM available for all the instances at the same time.

Use of vTPMs in containers :

- Hosseinzadeh *et al.* [42] conducted the first study implementing vTPMs for containers. They propose two methods: 1) integrating vTPM into the operating system kernel, and 2) placing vTPM in a separate container. They also mention ongoing research on native support for virtualization on hardware TPMs.
- Souppaya *et al.* [15] suggested attestation mechanisms, beginning with secured/measured boot, followed by providing a verified system platform. This is further extended by building a chain of trust rooted in hardware, extending it to the bootloader, OS kernel, system images, container runtime, and container images.

Secure Enclaves :

A secure enclave is a protected area of a computer's memory that is isolated from the rest of the system, ensuring that sensitive data and processes are kept secure even from other parts of the operating system [43]. Intel SGX (Software Guard Extensions), AMD SEV (Secure Encrypted Virtualization), ARM Trust Zone, and IBM SecureBlue++ are some of the examples. These enclaves utilize hardware-based security mechanisms to provide a high level of protection against various vulnerabilities, including side-channel attacks and unauthorized access.

Use of secure enclaves in container technology :

- Arnautov *et al.* [43] introduced Secure Linux Containers with Intel SGX (SCONE) to protect containers from attacks. SCONE's design emphasizes a smaller trust base, low performance overhead, and supports asynchronous system calls and user-level threading.
- Hunt *et al.* [44] developed Ryoan, which utilizes Intel SGX to establish a distributed sandbox. It uses enclaves to protect sandbox instances from malicious computing platforms and allows it to process secret data while preventing leaking secret data. The authors evaluated Ryoan across various tasks, such as email filtering, health analysis, image processing, and machine translation. It's important to note hardware limitations of SGX, such as the inability to run unmodified applications, should be considered.

V. PROPOSED SECURITY APPROACH

As discussed in the Section III, various threats affect containerized environments, and the Section IV covers their native security solutions. Vulnerabilities such as image vulnerabilities (compromised images for cryptominers [45]) and application vulnerabilities can directly impact the availability and integrity of containers during operation. These vulnerabilities allow adversaries to manipulate container configuration files and behavior. For example, attacks like DDoS and cryptojacking can exhaust CPU resources, rendering the container unavailable for legitimate requests. Researcher Jeeva Chelladhurai *et al.* proposed solution for DDoS kind of attacks [46] but sometimes it's also not working because of vulnerablities. Therefore, detecting unusual container behavior is crucial for protecting against such attacks and vulnerabilities.

For protecting integrity and availability of the container, We have suggested one procedure in which the first step involves computing the SHA-256 hash of all container configuration files and continuously monitoring these hashes for discrepancies. If an adversary alters any container properties, the hash will change, triggering a notification to the user about the modification. Upon detection, several actions can be proposed: stopping the container, detaching it from the network, or alerting other containers in the network to enhance their security and reject requests from the compromised container.

The second step involves continuously monitoring the actual resource usage of the container to ensure it does not exceed the limits specified in the configuration file. While cgroups and capabilities provide some level of control, misconfigurations or

vulnerabilities can lead to excessive memory and CPU usage. This additional check directly on the host machine ensures the container remains available and helps detect attacks such as DDoS or crypto jacking. If resource usage exceeds the specified limits, the user is notified to take appropriate action.

VI. EXPERIMENT AND RESULTS

As an experiment, we ran several Docker containers on a machine with an Intel® Core™ i5-10500 CPU @ 3.10GHz × 12 processor, 8GB of RAM, running Ubuntu 22.04.4 LTS 64-bit OS. The running containers are shown in Figure 7. Then, we executed a script that implements the procedures described in Algorithms 1 (CHM) and 2 (CCM).

Algorithm 1 Container Hash Monitoring

Input: HASH_FILE
Output: Container security monitoring
Initialization :
Initialize HASH_FILE, CURRENT_HASH_FILE
function CheckHashes()
  Initialize processed_containers as empty list
  Compute current hashes and save to CURRENT_HASH_FILE
if HASH_FILE differs from CURRENT_HASH_FILE **then**
    Get changed files using diff
  for all changed files **do**
    Extract file path and container ID
    if file exists **then**
      Get container name and last modified user
      if container already processed **then**
        Log the change and continue
      end if
      Log the change and prompt user for action:
      Disconnect container
      Stop container
      Inform other containers
      Execute user action and log compromised container
    else
      Log file is deleted
    end if
  end for
else
    Log no changes
end if
end function
Main Loop :
while true **do**
  Call CheckHashes()
end while=0

In the first step, as outlined in Algorithm 1, the process begins by retrieving the details of all containers and verifying their integrity through SHA-256 hashing of each container's configuration files. If a hash mismatch is detected, the system suggests actions to be taken, which are flexible and not limited to predefined responses. In the second step, according to Algorithm 2, the system monitors real-time CPU and memory usage of the containers against configured limits. If usage exceeds these limits, the system alerts the user and proposes appropriate responses to mitigate the anomaly.

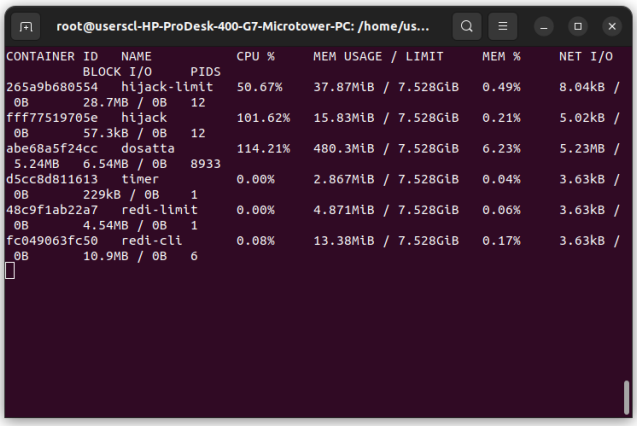

Fig. 7. Running container's details.

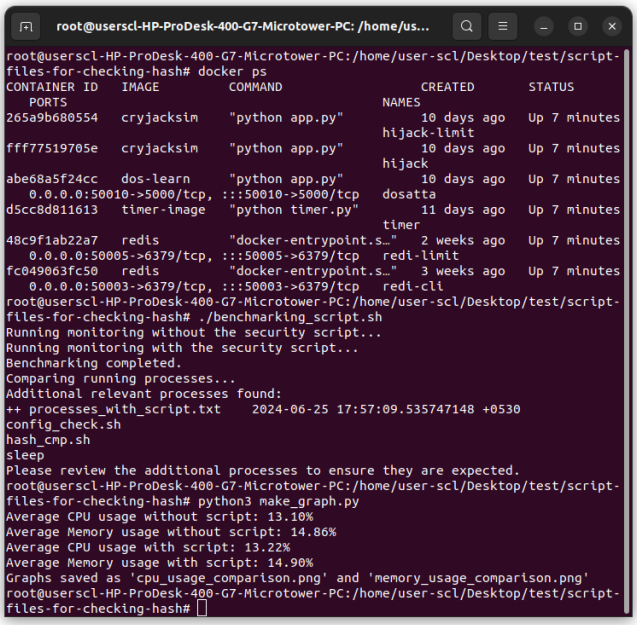

Fig. 8. Proof of transparency for execution of suggested solution

For example, if an adversary initiates an attack causing CPU usage to surpass the specified limits in the configuration file, the system promptly notifies the user. The user can then investigate whether the container is under attack and take necessary measures. This proactive approach facilitates early detection and mitigation of potential attacks, thereby ensuring continuous availability and integrity of the containers.

After executing the CHM and CCM solutions, We conducted benchmarks to assess its impact on memory and CPU

usage, Figure 7 shows a nominal 0.12% increase in CPU usage, while Figure 8 indicates 0.04% rise in memory usage following the execution of the CHM and CCM script. These minor increases demonstrate minimal impact on the machine's overall performance.

Algorithm 2 Container Configuration Monitoring

Input: no-input
Output: Container security monitoring
Initialization :
function check_cpu_limits(container_id, nanocpus, container_name)
    Get CPU usage from Docker stats
    Calculate CPU limit from cgroup
if CPU usage exceeds limit **then**
        Log exceeded usage and prompt user for action
        Disconnect container
        Stop container
        Inform other containers
        Execute user action
end if
end function
function check_memory_limits(container_id, memory_set_limit_from_json, container_name)
    Get memory usage from cgroup
if Memory usage exceeds limit **then**
        Log exceeded usage and prompt user for action
        Disconnect container
        Stop container
        Inform other containers
        Execute user action
end if
end function
function check_container_configs(container_id)
    Set paths to config files
if config files not found **then**
        return error
end if
    Extract container details
if container is running **then**
        Call check_cpu_limits(container_id, nanocpus, container_name)
        Call check_memory_limits(container_id, memory_set_limit_from_json, container_name)
end if
end function *Main Loop* :
while true **do**
    for all container_id in container_ids **do**
        Call check_container_configs(container_id)
    end for
end while=0

VII. CONTAINER DATA VOLUMES

Containers are most commonly used with stateless applications because their filesystems are temporary in nature. Changes made to a container's environment are lost when the container stops, crashes, or is replaced [47]. Therefore, to enable stateful container applications, we need to attach storage volumes to containers. Storage volumes provide persistent storage that is independent of individual containers and exists outside the containers. These volumes are mounted onto filesystem paths within containers. When containers write to a path beneath a volume mount point, the changes are applied to the volume instead of the container's writable image layer. This data remains available even after the container's lifetime, as volumes are stored separately on the host. However, if volumes are stored on a malicious host, such as a container as a service(CaaS) provider or registry administrators, they could potentially access these volumes. To protect this data from unauthorized users, encryption of data volumes is essential. From a data security perspective, the most promising approach is the use of the secure enclaves such as Intel SGX or AMD Trust Zone.

A. Solutions for protecting container storage volumes

1) Giannakopoulos *et al.* [48] have introduced and illustrated a mechanism for secure Docker image manipulation across its lifecycle. They employ mechanisms for encryption/decryption at-rest and on-the-fly to maintain confidentiality during the creation, storage, and utilization of a Docker image.
2) SCONE [43] offers filesystem shields that prevent low-level attacks, including those where the OS kernel manipulates pointers and buffer sizes passed to services. It ensures the confidentiality and integrity of the application data transmitted through the OS. With the filesystem shield, users can encrypt files or verify access to them. Additionally, SCONE extends support for encrypting and authenticating the filesystem of temporary containers.
3) Richter *et al.* [49] suggested utilizing Intel SGX enclaves for disk encryption and showcased its resilience against single-level and multi-level authentication threats. However, their solution imposes a significant overhead compared to traditional disk encryption methods when applied to encrypting data volumes and images.

VIII. FUTURE SCOPE

1) Utilizing generative artificial intelligence facilitates the detection of anomalies in containerized environments and enables the simulation of attacks to examine vulnerabilities mentioned above. It can also aid in creating dynamic policies utilized in seccomp.
2) To ensure the security of container storage volumes on CaaS, encryption is essential. Therefore, finding a lightweight encryption-decryption algorithm suitable will help reduce the time required for accessing the data efficiently.
3) Blockchain technologies can be employed to track user engagement on social media platforms for digital forensics purposes, enhancing transparency and accountability.

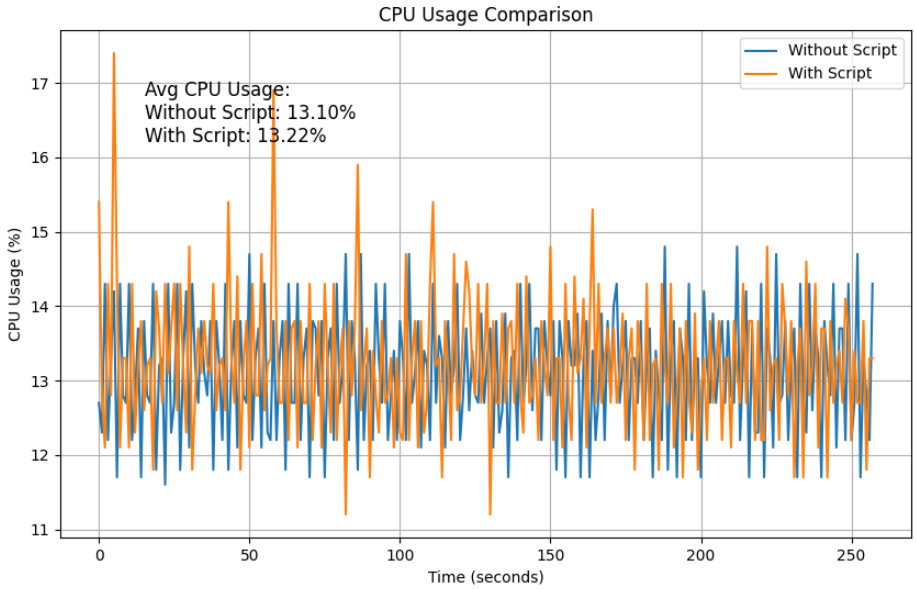

Fig. 9. CPU usage comparison between with script and without script.

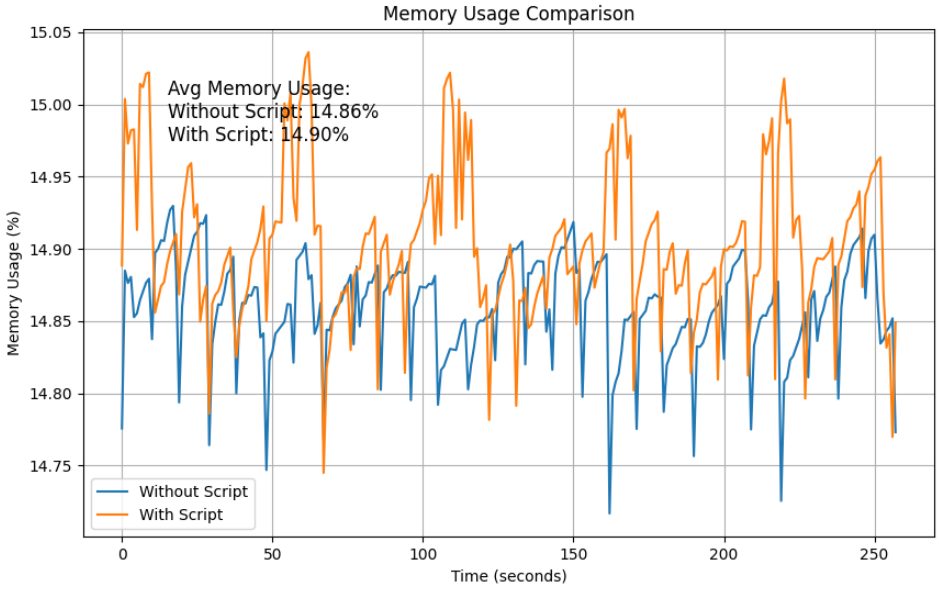

Fig. 10. Memory usage comparison between with script and without script.

IX. CONCLUSIONS

With the increasing adoption of microservices for development, containers have become an essential component for deploying these microservices due to their lightweight characteristics. However, the direct involvement of the host kernel poses security risks and confidentiality concerns when using Container as a Service (CaaS). Therefore, this paper highlights the necessity of containers, container ecosystems, and available security mechanisms to address security breaches in containers and storage volumes. This paper aims to provide future researchers with a foundational understanding of the terminology necessary for conducting further research in this field.

REFERENCES

[1] S. Singh and N. Singh, "Containers & Docker: Emerging roles & future of Cloud technology," 2016 2nd International Confer-

ence on Applied and Theoretical Computing and Communication Technology (iCATccT), Bangalore, India, 2016, pp. 804-807, doi: 10.1109/ICATCCT.2016.7912109.

[2] Sharma Srinarayan, Park Young. (2011). VIRTUALIZATION: A REVIEW AND FUTURE DIRECTIONS Executive Overview. American Journal of Information Technology. 1. 1-37.

[3] S. Sultan, I. Ahmad and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," in IEEE Access, vol. 7, pp. 52976-52996, 2019, doi: 10.1109/ACCESS.2019.2911732.

[4] Will Wang, "Demystifying Containers 101: A Deep Dive Into Container Technology for Beginners".Accessed: Sep. 13, 2023. [Online]. Available: https://www.freecodecamp.org/news/demystifying-containers-101-a-deep-dive-into-container-technology-for-beginners-d7b60d8511c1/

[5] Amit sheps, "What Is a Container Runtime?". Accessed: Apr.15, 2024. [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-security/container-runtime/

[6] Watada, Junzo & Roy, Arunava & Kadikar, Ruturaj & Pham, Hoang & Xu, Bing. (2019). Emerging Trends, Techniques and Open Issues of Containerization: A Review. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2945930.

[7] Rehman, A. U. & Aguiar, Rui & Barraca, João. (2019). Network Functions Virtualization: The Long Road to Commercial Deployments. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2915195.

[8] D. Walsh. (2014). Are Docker Containers Really Secure?. Accessed: Oct. 27, 2023. [Online]. Available: https://opensource.com/business/14/7/docker-security-selinux

[9] B. M. Abbott, "A security evaluation methodology for container images," M.S. thesis, Brigham Young Univ., Provo, UT, USA, 2017.

[10] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," IEEE Cloud Comput., vol. 3, no. 5, pp. 54–62, Sep./Oct. 2016.

[11] M. Stuart. (2015). Evolving Container Architectures. Accessed: Sept. 20, 2023. [Online]. Available: https://wikibon.com/evolving-container-architectures/

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance Evaluation of container-based virtualization for high-performance computing environments," in Proc. 21st Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process., Feb./Mar. 2013, pp. 233–240.

[13] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud-survey results and own solution," J. Grid Comput., vol. 14, no. 2, pp. 265–282, Jun. 2016.

[14] Li, Yuchong & Liu, Qinghui. (2021). A comprehensive review study of cyber-attacks and cyber security; Emerging trends and recent developments. Energy Reports. 7. 10.1016/j.egyr.2021.08.126.

[15] M. Souppaya, J. Morello, and K. Scarfone, Application Container Security Guide, vol. 800. Gaithersburg, MD, USA: NIST, 2017, p. 190.

[16] Rory McCune. "Containers are just processes". Accessed: feb. 16, 2024. [Online].https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-1/\#containers-are-just-processes

[17] Michael Kerrisk(2019). "Containers unplugged:Linux namespaces"Accessed: feb. 24, 2024. [Online].https://man7.org/conf/ndctechtown2019/Linux-namespaces-NDC-TechTown-2019-Kerrisk.pdf

[18] Chandramouli, R., & Chandramouli, R. (2017). Security assurance requirements for linux application container deployments. Gaithersburg, MD, USA: US Department of Commerce, National Institute of Standards and Technology.

[19] D. S. Team. Docker Security. Accessed: March 13, 2024. [Online].Available: https://docs.docker.com/engine/security/security/

[20] Zhiqiang Jian and Long Chen. 2017. A Defense Method against Docker Escape Attack. In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy (ICCSP '17). Association for Computing Machinery, New York, NY, USA, 142-146. https://doi.org/10.1145/3058060.3058085

[21] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis and H. Wang, "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds," 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, 2017, pp. 237-248, doi: 10.1109/DSN.2017.49.

[22] Michael Kerrisk. cgroups(7)-Linux manual page" Accessed: feb. 29, 2024. [Online]. Available : https://man7.org/linux/man-pages/man7/cgroups.7.html

[23] J. Chen, Z. Feng, J.-Y. Wen, B. Liu, and L. Sha. (2018)."A Container-based DoS Attack-Resilient Control Framework for Real-Time UAV Systems" Accessed: feb. 29, 2024.[Online]https://arxiv.org/abs/1812.02834

[24] Massimo Re Ferre & Samuel Karp, "How Amazon ECS manages CPU and memory resources" Accessed: march 1,2024.[Online]. Available: https://aws.amazon.com/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources/

[25] Michael Kerrisk. capabilities(7)-Linux manual page" Accessed: March 5, 2024. [Online]. Available:https://man7.org/linux/man-pages/man7/capabilities.7.html

[26] Siemens, "Linux capabilities" Accessed: March 5, 2024. [Online]. Available: https://docs.eu1.edge.siemens.cloud/develop_an_application/developer_guide/industrial_edge_platform/docker_and_security/02_01_03_Linux%20Capabilities.html

[27] Docker docs. "Seccomp security profiles for Docker" Accessed: March 7, 2024. [Online]. Available: https://docs.docker.com/engine/security/seccomp/

[28] Wan, Z., Lo, D., Xia, X., Cai, L., & Li, S. (2017). Mining sandboxes for linux containers. In A. Memon, Y. Nishi , I. Schieferdecker, & H. Washizaki (Eds.), Proceedings - 10th International Conference on Software Testing, Verification and Validation, ICST 2017: 13-17 March Tokyo, Japan (pp. 92-102). IEEE, Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/ICST.2017.16

[29] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in Proc. USENIX Secur.Symp., Aug. 2002, pp. 17-31.

[30] J. Johansen. "Making Linux Security Modules Available to Containers". Accessed: Mar. 14, 2024. [Online]. Available: https://archive.fosdem.org/2018/schedule/event/containers_lsm/

[31] Loukidis-Andreou, Fotis, Ioannis Giannakopoulos, Katerina Doka and Nectarios Koziris. "Docker-Sec: A Fully Automated Container Security Enhancement Mechanism." 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) (2018): 1561-1564.

[32] Mp, Amith Raj, Ashok Kumar, Sahithya J Pai and Ashika Gopal. "Enhancing security of Docker using Linux hardening techniques." 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT) (2016): 94-99.

[33] AppArmor. Accessed: March 15, 2024. [Online]. Available: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/apparmor.html#documentation

[34] Selinux Userspace. Accessed: March 15, 2024.[online]. Available: https://github.com/SELinuxProject/selinux

[35] LoadPin. Accessed: March 16, 2024.[online]. Available: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/LoadPin.html

[36] Yama. Accessed: March 16, 2024.[online]. Available: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/Yama.html

[37] Tomoyo. Accessed: March 16, 2024.[online]. Available: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/Tomoyo.html

[38] Smack. Accessed: March 16, 2024.[online]. Available: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/smack.html

[39] https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/Yama.html

[40] Yehuda Lindell. "The unbound nextgen VHSM" Accessed: 19 March, 2024. [Online]. Available: https://www.fintechfutures.com/files/2020/09/vHSM-Whitepaper-v3.pdf

[41] Ezirim, Kenneth, Wai Khoo, George Koumantaris, Raymond Law and Irippuge Milinda Perera. "Trusted Platform Module - A Survey." (2014).

[42] Hosseinzadeh, Shohreh, Samuel Laurén and Ville Leppänen. "Security in Container-Based Virtualization through vTPM." 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC) (2016): 214-219.

[43] Sergei, Arnautov., Bohdan, Trach., Franz, Gregor., Thomas, Knauth., Andre, Martin., Christian, Priebe., Joshua, Lind., Divya, Muthukumaran., Dan, O'Keeffe., Mark, Stillwell., David, Goltzsche., David, Eyers., Rüdiger, Kapitza., Peter, Pietzuch., Christof, Fetzer. (2016). SCONE: secure Linux containers with Intel SGX. 689-703. doi: 10.5555/3026877.3026930

[44] Hunt, Tyler, Zhiting Zhu, Yuanzhong Xu, Simon Peter and Emmett Witchel. "This Paper Is Included in the Proceedings of the 12th Usenix Symposium on Operating Systems Design and Implementation (osdi '16). Ryoan: a Distributed Sandbox for Untrusted Computation on Secret Data Ryoan: a Distributed Sandbox for Untrusted Computation on Secret Data." .

[45] Aviv Sasson, 20 Million Miners: Finding Malicious Cryptojacking Images in Docker Hub. Accessed: April 16, 2024.[online]. Available: https://unit42.paloaltonetworks.com/malicious-cryptojacking-images/
[46] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from denial of service (DOS) attacks" in Proc. IEEE Int. Conf. Services Comput. (SCC), Jun./Jul. 2016, pp. 856-859.
[47] James walker, Docker volumes. Accessed: March 20, 2024.[online]. Available: https://spacelift.io/blog/docker-volumes
[48] Giannakopoulos, Ioannis & Papazafeiropoulos, Konstantinos & Doka, Katerina & Koziris, Nectarios. (2017). Isolation in Docker through Layer Encryption. 2529-2532. 10.1109/ICDCS.2017.161.
[49] Richter, L., Götzfried, J., & Müller, T. (2016). Isolating Operating System Components with Intel SGX. Proceedings of the 1st Workshop on System Software for Trusted Execution.